UNIVERSITÉ DE NICE SOPHIA ANTIPOLIS

École Doctorale STIC
Sciences et Technologies de l'Information et de la Communication

THÈSE

pour l'obtention du titre de

Docteur en Sciences

Mention Informatique

présentée et soutenue par

Justine Rochas

Execution Support for Multi-threaded Active Objects: Design and Implementation

Thèse dirigée par Ludovic Henrio

Soutenue le 22 Septembre 2016

Jury

Rapporteurs	Einar Broch Johnsen	University of Oslo
	Lionel Seinturier	Université de Lille
Examinateurs	Florian Kammüller	Middlesex University London
	Johan Montagnat	CNRS
	Fabrice Huet	Université de Nice Sophia Antipolis
Directeur de thèse	Ludovic Henrio	CNRS

Résumé

Pour aborder le développement d'applications concurrentes et distribuées, le modèle de programmation à objets actifs procure une abstraction de haut niveau pour programmer de façon concurrente. Les objets actifs sont des entités indépendantes qui communiquent par messages asynchrones. Peu de systèmes à objets actifs considèrent actuellement une exécution multi-threadée. Cependant, introduire un parallélisme contrôlé permet d'éviter les coûts induits par des appels de méthodes distants.

Dans cette thèse, nous nous intéressons aux enjeux que présentent les objets actifs multi-threadés, et à la coordination des threads pour exécuter de façon sûre les tâches d'un objet actif en parallèle. Nous enrichissons dans un premier temps le modèle de programmation, afin de contrôler l'ordonnancement interne des tâches. Puis nous exhibons son expressivité de deux façons différentes: d'abord en développant et en analysant les performances de plusieurs applications, puis en compilant un autre langage à objets actifs avec des primitives de synchronisation différentes dans notre modèle de programmation. Aussi, nous rendons nos objets actifs multi-threadés résilients dans un contexte distribué en utilisant les paradigmes de programmation que nous avons développé. Enfin, nous développons une application pair-à-pair qui met en scène des objets actifs multi-threadés. Globalement, nous concevons un cadre de développement et d'exécution complet pour les applications hautes performances distribuées. Nous renforçons notre modèle de programmation en formalisant nos contributions et les propriétés du modèle. Cela munit le programmeur de garanties fortes sur le comportement du modèle de programmation.

Abstract

In order to tackle the development of concurrent and distributed systems, the active object programming model provides a high-level abstraction to program concurrent behaviours. Active objects are independent entities that communicate by the mean of asynchronous messages. Compared to thread-based concurrent programming, active objects provide a better execution safety by preventing data races. There exists already a variety of active object frameworks targeted at a large range of application domains: modelling, verification, efficient execution. However, among these frameworks, very few of them consider a multi-threaded execution of active objects. Introducing a *controlled* parallelism within active objects enables overcoming some of their well-known limitations, like the impossibility of communicating efficiently through shared-memory.

In this thesis, we take interest in the challenges of having multiple threads inside an active object, and how to safely coordinate them for executing the tasks in parallel. We enhance this programming model by adding language constructs that control the internal scheduling of tasks. We then show its expressivity in two ways: first in a classical approach by developing and analysing the performance of several applications, and secondly, by compiling another active object programming language with different synchronisation primitives into our programming model. Also, we make multi-threaded active objects resilient in a distributed context through generic engineering constructs, and by using our programming abstractions. Finally, we develop a peer-to-peer application that shows multi-threaded active objects and their features in action. Overall, we design a flexible programming model and framework for the development of distributed and highly concurrent applications, and we provide it with a thorough support for efficient distributed execution. We reinforce our programming model by formalising our work and the model's properties. This provides the programmer with guarantees on the behaviour of the programming model.

Remerciements

Merci à mes deux rapporteurs pour le temps qu'ils ont accordé à la compréhension de mes travaux, ainsi qu'aux membres de mon jury de soutenance.
Merci à mon directeur de thèse Ludo avec qui j'ai eu la chance de travailler en collaboration et qui a fait bien plus qu'une mission d'encadrement.
Merci à Fabrice qui m'a présentée à l'équipe et qui a toujours su me conseiller dans toutes les facettes que cette thèse a pu avoir.
Merci à Oleksandra pour son support quotidien et son enthousiasme débordant.
Merci aux autres membres de l'équipe, tout particulièrement à Fabien, Françoise et Éric avec qui j'ai pu passer d'agréables moments de tous les jours.
Merci à ma collaboratrice Sophie et à Laurent pour son aide précieuse à mon arrivée dans l'équipe.
Merci à Ludwig, mon plus fidèle auditeur, avec qui j'ai la chance de partager mes intérêts professionnels et extra-professionnels.
Merci à mes parents qui m'ont fait confiance pour en arriver là et à mon frère toujours présent pour nos sorties sportives et compétitives.
Merci à mon maître de Taekwondo Pierre grâce à qui j'ai pu ponctuer mes semaines d'une évasion indispensable.
Enfin, merci à tous mes professeurs de l'université de Nice Sophia Antipolis qui m'ont donné le goût de l'informatique sous tous ses aspects.

Table of Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

Contents

1.1 Motivation

At the age of e-commerce, ubiquitous chats, collaborative on-line editors, synced cloud storage, real-time games and connected objects, each and every of our digital habits involve simultaneous multi-party interactions. Such now common usages are inherently supported by concurrent and distributed computer systems that have been evolving through the multi-core era. However, the languages for programming these systems have not evolved as fast as the hardware and applications, although the purpose of programming languages is to map the applications to the hardware. This is especially true for the programming languages that are well established in the industry. In general, existing programming models and languages lack of support for handling the concurrency of parallel programs and for writing distributed applications. As such, parallel and distributed systems have coped for decades with programming languages that barely provide support for parallelism.

This support is embodied by so-called *threads*, an abstraction for independent, asynchronous sequences of instructions. Threads represent the oldest parallel programming abstraction that is given in the hands of the programmer, yet they are still omnipresent. Currently, all programming languages provide at least an API to create and start threads. The problem in the programmer manipulating bare threads is that he must have a precise knowledge of the right way to use and synchronise threads, which is not trivial. Synchronisation in concurrent systems is what can lead to two well-known categories of concurrency bugs: deadlocks, where there exists a circular dependency between tasks, and data races, where several threads access a shared resource with no synchronisation. Typically, in a shared-variable concurrency model, one must analyse where data protection is required, most often by the mean of *locks* [And99]. Not doing this analysis exposes the application to the risk of non-deterministic bugs [Lee06]. In short, with threads, the burden of parallelism is put on the shoulders of the programmer.

On the other hand, as programming languages evolved, modularity of programs has become a major leitmotiv. A representative of this organisational change is object-oriented programming, which represent a reality-like abstraction for programs. The fact that data structures and procedures could be manipulated together as a whole was first explored in the 'algorithm language' ALGOL 60 [Bac+63]. Later on, Simula 67 [DN66], an extension of ALGOL 60, introduced the main concepts of object-oriented programming in term of classes, and in term of objects as instance of classes. The creator of SmallTalk [Kay93] introduced the term of object-oriented programming and the pervasion of objects was carried to the extreme in further versions of SmallTalk [GR83], where each and every entity is seen as an object. Since then, multi-paradigm programming languages have been perpetually developed, and almost all of them include an object layer that is strongly inspired from the SmallTalk approach. Object-oriented programming is now borne by the programming language triptych - C++, Java, C# - to support production code in the industry.

As said earlier, programmer-side support for threads is massively embedded in mainstream programming languages. Thus, object-oriented programming and multi-threaded programming are often mixed in the same language, although they are conceptually orthogonal. Indeed, objects encapsulate a state which should only be modified through messages, seen as object methods. By allowing several threads of control to execute these methods, the encapsulation of objects becomes broken. Moreover, this architectural inconsistency is worsen by the use of locks within a class: locks force an exposure of the object's variables on which the synchronisation applies. Finally, using threads in a distributed context is not adapted, because threads that are settled on different computers cannot communicate through shared-memory. A global shared-memory in distributed architectures is rather difficult to obtain, and often comes at a too high performance cost. Alternatively, it can be efficient if full data consistency is not guaranteed, but approximate distributed systems are out of our scope. The current programming languages clearly lack of parallel and distributed abstractions that fit the currently developed applications. They do not enforce by default a safe parallel execution nor offer a programmer-friendly way to program concurrently.

1.2 Objectives

Already, we have given many reasons why threads, coupled with object-oriented programming, cannot be the right approach for implementing parallel and distributed systems. In order to write applications with higher-level constructs, inherently concurrent and distributed, one has to let the developers focus on the application business, and for that, they need well-integrated abstractions for concurrency and distribution which they can trust and rely on. They must be given frameworks that provide convenient constructs and thread safety guarantees, and not at the expense of the application's performance. A complete stack of technologies can help the programmer in this objective. My work focuses on the elements of the stack that are the closest to the programmer, and that are directly used by him: the programming models, the programming languages, and their APIs. Crafting a programmer experience that is as pleasant as possible while keeping application's safety and performance is the guideline of this thesis. On one hand, low-level abstractions for programming parallel and distributed systems must be given up, first because they are not user-friendly, and second because they do not scale with the intricacy of programs. One the other hand, high-level abstractions are often incomplete or lack efficiency, which also cuts off scalability. In particular, there is a need for unifying the abstractions that are given for parallelism: we need to redefine concurrent abstractions so that they do not only work in the case of shared-memory architectures, if we aim at raising them to a distributed execution.

In this thesis, we want to offer a parallel and distributed programming model and runtime that balances the ease of programming and the performance of the application. We offer the abstractions that the programmer needs in order to effectively and safely program modern systems, that are necessarily parallel and distributed. In particular, we focus on the active object programming model, because it gives the building blocks for safe parallelism and distribution. The active object programming model also reconciles object-oriented programming and concurrency. A characteristic of active objects is the absence of shared memory, which makes them adapted to distributed execution as well. Each active object has its own local memory and cannot access the local memory of other active objects. Consequently, objects can only exchange information extra-memory through message passing, implemented with requests sent to active objects. This characteristic makes object accesses easier to trace, and thus it is easier to check their safety. Active objects enable formalisation and verification thanks to the framework they offer: the communication pattern is well-defined as well as the accesses to objects and variables. This facilitates static analysis and validation of active object-based programs. For all these reasons, the active object model is a very good basis for the construction of programming languages and abstractions that match the influx of concurrent and distributed systems.

In this thesis, our goal is to promote holistic approaches when considering active object languages. We aim at studying both the implementation of the languages and their semantics. The gap between language semantics and language implementation is often not enough considered, and our objective is to show that this gap can nevertheless be fulfilled with the right approaches and the right tools.

1.3 Contributions

The global contribution of this thesis is to come with a global approach for tackling the development and the controlled execution of concurrent and distributed systems. We build a complete framework based on multi-threaded active objects.

We use it for bridging a gap between active object languages, and for developing realistic applications. More precisely, the contribution of this thesis comes in three aspects, summarised below. More details on the content of the chapters are available in Section 1.4.

Local scheduling for multi-threaded active objects. The first contribution extends the multi-threaded active object programming model with a set of annotations that controls the local scheduling of requests. This contribution allows the programmer to manage the internal execution of multi-threaded active objects, while keeping the safety ensured by the automated interpretation of declarative annotations.

From modelling to deployment of distributed active objects. The second contribution is an encoding of cooperative active object languages into a precise configuration of multi-threaded active objects. We provide one fully implemented translator and prove its correctness. This general study about active object languages provides the users and the designers of active object languages with a precise knowledge of the guarantees given by active object programming models and frameworks.

Efficient execution support for multi-threaded active objects. A third contribution of this thesis is to improve the execution support of the multi-threaded active object programming model. We provide a post-mortem debugger for its applications. It helps the developer in seeing the effects of annotations on the application's execution. It makes the development iterations easier and faster by providing a comprehensive feedback on an application's execution. We also come with a preliminary fault tolerance protocol and implementation for multi-threaded active objects. It provides the developer with a recovery mechanism that automatically handles crashes in distributed executions of multi-threaded active objects.

Finally, to illustrate the programming model and to show the effectiveness of our framework, we use fault tolerant multi-threaded active objects in the realistic scenario of a peer-to-peer system. We provide a middleware approach to implement a robust distributed broadcast.

1.4 Outline

The contributions are structured around four core chapters. Each chapter is ended with a conclusion that summarises the content of the chapter, along with a comparison to the related works for this chapter. We summarise the chapters of this thesis below.

Chapter 2 presents the context of this thesis which is the active object programming model. We introduce there the key notions that are related to this programming model and organise them into a classification, that we bring from a general study of active object languages. We then give an overview of active object languages regarding the classification. We also introduce multi-threaded active objects, that represent the basis of this thesis. More precisely we introduce the multiactive object programming model in details, the formal calculus that is associated to it, MultiASP, and finally its implementation in ProActive, the Java library that is the main technology used to implement the different works presented in this thesis.

Chapter 3 introduces the first contribution of this thesis, published in [3], which offers advanced scheduling controls in the context of multi-threaded active objects. The idea is to empower the programmer with safe constructs that impact on the priority of requests and their allocation on available threads, in the context of multiactive objects. The mechanisms are introduced in a didactic manner and presented as they are implemented in the ProActive library. The properties of the priority specification are studied in details and evaluated with micro-benchmarks. The chapter is concluded with related works on application-level scheduling in active object programming languages.

Chapter 4 promotes the work published in [4]. In this work, we use the multiactive object framework to encode another active object language, ABS. First, we present a backend for this language that automatically translates ABS programs into ProActive programs. Secondly, we use MultiASP, the calculus of ProActive, to formalise the translation and prove its correctness. The associated proofs are presented in appendix of this thesis. We show in the thesis the relevant elements of the proofs, the lemmas on which we rely, the equivalence relation, and the restrictions that apply. We give an informative feedback on the outcome of this translation in the conclusion of this chapter.

Chapter 5 highlights the latest works done around multiactive objects. In particular, it is split into two major parts. The first part presents a visualiser of multiactive object executions, that helps in debugging multiactive object-based applications. We review the debugger tool and we present two use cases in which it proves to be useful. The second part deals with the fault tolerance of multiactive object. To enter this subject in details, we first introduce the fault tolerance protocol that was developed for active objects in ProActive. We then present our generic adaptation of the protocol for multiactive objects. We give as well the current limitations of the new protocol and some possible solutions and directions to improve it in a future work.

Chapter 6 exposes an application that we have developed with the multiactive object programming model and that is set in the context of peer-to-peer systems. We start by introducing the Content-Addressable Network (CAN) together with our contribution, published in [2], dealing with the challenge of efficiently broadcasting information in a CAN. We then move to the developed application that consists in making the efficient broadcast in CAN robust, thanks to fault tolerant multiactive objects. This chapter actually brings together several notions that were introduced in the previous chapters.

Chapter 7 concludes the thesis in two parts. It first summarises the thesis by recalling the methodology that was employed throughout the works, and by going over the main contributions of each chapter. Finally, it presents the perspectives that have been opened by this thesis.

Chapter 2

Background and Related Works

Contents

In this chapter, we introduce the ecosystem of this thesis, which starts from the active object programming model. We first introduce the context in which active objects were created, and the models that inspired active objects. We then propose a classification of active object-based languages based on implementation and semantic aspects. After that, we offer an overview of active object-based languages and frameworks, taking into account the previous classification. Then, we focus on multi-threaded active objects, that represent the specific context of this thesis. Finally, we relate the content of this thesis with the ecosystem presented before and justify its relevance there.

2.1 The Active Object Programming Model: Origins and Context

The active object programming model, introduced in [LS96], has one global objective: facilitate the correct programming of concurrent entities. The active object paradigm derives from actors [Agh86]. Actors are concurrent entities that communicate by posting messages to each other. Each actor has a mailbox in which messages arrive in any order. An actor has a thread that processes its messages sequentially. Only the processing of a message can have a side effect on the actor's state. This is a way to inhibit the effect of preemption on operating system threads. Since actors evolve independently, they globally execute concurrently. Yet, actors ensure the absence of *data races* locally. A data race exists if two parallel threads can access the same memory location. As actors encapsulate their state, an actor's state cannot be modified by other threads than the unique thread associated to the actor.

Once a message is posted to an actor, the sender continues its execution, without knowing when the message will be processed by the other actor. This is called an *asynchronous* communication pattern. In the original actor model, actors do

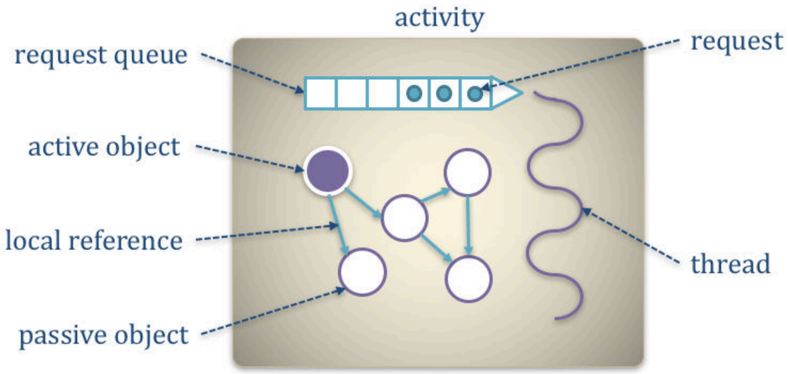


Figure 2.1 – Structure of an activity.

not guarantee the order in which messages are processed. If another computation is dependent on the result of the processing of a message, then a new message should be issued to the initiator of the first message. We call this new message a *callback* message.

Active objects are the object-oriented descendants of actors. They communicate with asynchronous method invocations. Asynchronous communication between active objects is First In First Out (FIFO) point-to-point, which means that the messages are causally ordered, preserving the semantics of two sequential method calls. In this sense, active objects are less prone to race conditions than actors, because their communications characterise more determinism. Like an actor, an active object has a thread associated to it. This notion is called an *activity*: a thread together with the objects managed by this thread. An activity contains one active object and several encapsulated objects that are known as *passive objects*. Figure 2.1 pictures the structure of an activity.

When an object invokes a method of an active object in another activity, this creates a *request* on the callee side. The invoker continues its execution while the invoked active object processes the request asynchronously. A request is dropped in the active object *request queue*, where it awaits until it can be executed. Contrarily to the messages of actors, the method invocations of active objects return a result. And since method invocations can be asynchronous, the result of the invocation might not be known just after the invocation. In order to have room for the prospected result, and to allow the invoker to continue its execution, a place holder is created for the result of the asynchronous invocation. This place holder is called a *future*. A future represents a promise of response. It is an empty object that will later be filled by the result of the request. We commonly say that a future is *resolved* when the value of the future is computed and available. Futures have seen their early days in functional languages: in MultiLisp [Hal85] and in ABCL/1 [YBS86]. They have been later on formalised in ABCL/f [TMY94], in [FF99], and, more recently, in a concurrent lambda calculus [NSS06], and in Creol [BCJ07]. A future is a particular object that will eventually hold the result of an asynchronous method invocation. In the meantime, a future allows the caller to continue its execution even if the result has not been computed yet, until the value of the future is needed to proceed.

In practice, actors and active objects have fuzzy boundaries, one being the historical model that inspired the other model. When actor frameworks are implemented using object-oriented languages, they often mix the characteristics of actors and active objects, that were established in the original models. For example, some actor frameworks (detailed in Paragraph 2.3.7), like Salsa, are implemented using method invocations to post messages. Others, like Scala actors and Akka, allow actors to synchronise on futures, which is not strictly part of the original model proposed in [Agh86]. Actors and active objects share the same objectives and have the same architecture, so there is not point in distinguishing the two models, apart from a comparative point of view on specific instances. This is the reason why, in the rest of the manuscript, we might use interchangeably actor or active object terms depending on the context. We also use sometimes the terms ‘request’ and ‘message’ interchangeably, whether we take the point of view of active objects or actors. These different denominations have no impact on their actual meaning.

In summary, active objects and actors enforce decoupling of activities: each activity has its own memory space, manipulated by its own thread. This strict isolation of concurrent entities makes them suited to distributed systems. Furthermore, the notion of activity provides a convenient abstraction for implementing non functional features of distributed systems like components, built on top of objects, or like process migration, group communication, and fault tolerance.

More recently, active objects have been playing a part in service-oriented programming, and especially in mobile computing [CVCGV13; Gör14], thanks to their loose coupling and to their execution safety. However, the strict active object model suffers from obsolescence from a particular perspective: it processes requests in a mono-threaded way. Thereby, many extensions and adaptations of the active object programming model have been proposed through the past ten years, in order to fit with new usages, new technologies, and new hardware. These enhancements have scattered the active object paradigm such that each new branch of active objects is adapted to a particular environment and targets a particular objective.

2.2 A Classification of Active Object Languages

In this section, we present the aspects under which active object languages can be studied, and the choices that must be made in order to implement them. In particular, there are crucial questions to answer when implementing an active object language. Answering these questions often determines the application domain to which the language is the most adapted. We will see that those aspects will have a large impact on the contributions of this thesis. Throughout the following subsections, we briefly mention corresponding related works in order to illustrate the point without going much into details. Related works are presented in details in the next section.

2.2.1 Object Models

All active object-based languages define a precise relation between active objects and regular objects. Considering the object models in active object languages boils down to answering the question “how are objects associated to threads?”, or more precisely, in active object words, “how are objects mapped to activities?”. In existing active object languages, we can find three categories of object models.

Uniform Object Model. In this model, all objects are active objects with their own execution thread, and with their own request queue. All communications between objects are made through requests. The uniform object model is convenient to formalise and reason about, but can lead to scalability issues when put into practice. Indeed, in the case where the model is implemented with as many threads of control as objects, the performance of the application becomes quickly poor when executed on commodity computers. Alternatively, the model can be implemented by associating each object to a logical thread. Logical threads are threads that are seen, and that can plainly be manipulated as threads, at the application level, but that might not exist at the operating system level. In this case, several logical threads are mapped to one operating system thread. Although this solution would scale in number of threads managed by a computed, it has two main drawbacks. Firstly, logical thread are sometimes difficult to implement considering the underlying execution platform. Secondly, even with logical threads, the whole active object baggage is carried anyway: the active object queue, the request scheduler, and in general all the non-functional structure of the active object, which can also be problematic at some point. Nevertheless, the concern of scalabilty might not be relevant considering the target of the language. Creol, detailed in Section 2.3.1, is representative of this active object model and has a compositional proof theory [BCJ07].

Non Uniform Object Model. This model involves active and passive objects. A passive object is only directly accessible (in term of object reference) locally and cannot receive asynchronous invocations. In this sense, passive objects are only accessible remotely through an active object, i.e. they are part of the state of the active object. ASP, the foundation stone of this thesis, presented in Section 2.3.4, features a non uniform object model and restricts the accessibility of a passive object to a single active object. In this case, a passive object exists in a single activity, which prevents concurrent state alterations. In general, an activity contains one active object and several passive objects. The non uniform object model is scalable at a large scale as it requires less remote communications and runs with less concurrent threads. Reducing the number of activities also reduces the number of references that must be globally accessible in a distributed execution. Thus, a large number of passive objects can still be instantiated. However, this model is tougher to formalise and reason about than the uniform object model.

Object Group Model. The object group model is inspired from the notion of *group membership* [BJ87] that applies to the partitioning of processes in distributed systems. However, with group membership, the groups are dynamically formed, whereas in the object group model, groups are assigned only once. In this model, an activity is made of a set of objects sharing the same execution thread and the same request queue. The difference with the non uniform object model is that all objects in the group can be invoked from another activity, there is no notion of passive object. Compared to the two previous object models, this approach has a balanced scalability, due to the reduced number of concurrent threads, and a good propensity to formalisation, because there exists only one category of objects. Nevertheless, this object model presents a major drawback in distributed settings: all objects are accessible from any group. This fact implies that an object-addressing layer, that applies on top of shared-memory, must be created and maintained. Thus, in distributed setting, such a model does not scale as all objects created by the program must be registered in a global directory in order to be further retrieved for invocation. ABS, detailed in Section 2.3.3 and often referred to in this thesis, is an example of active object-based language that features the object group model.

Other hybrid kinds of object models have been released in actor models and languages. The E programming language [MTS05] is an actor-based programming language that features inter object communication through message passing. E uses many kinds of object references in order to identify suspect accesses, which is called isolating *coordination plans*, and handles object state hazards. Other works on the actor model attempt to release the constraint of having message-passing as an exclusive communication pattern. Some of these works make actors partially communicate through shared memory. In [DKVCD12], the actor model is extended to provide actors with a dynamic safe communication through shared memory, based on the notions of object *domains* and of *synchronisation views*. This is done by an inversion of control in which it is the user of the resource - and not its owner - that has an exclusive access to the resource. In [LL13], the performance of the actor model is optimised specifically for the single-writer multiple-reader case. In this case only, data is communicated through shared memory instead of message passing.

2.2.2 Scheduling Models

Active objects are tightly related to threads through the notion of activity. Latest developments in the active object programming model have led to the emergence of new request execution patterns. This brings to the question "how requests" are executed in active objects. Although we divided here the classification into three categories based on threading aspects, the focus of our analysis is on the interleaving of request services.

Mono-threaded Scheduling Model. This threading model is the one supported in the original active object programming model. It specifies that an active object is associated to a single thread of execution. In addition, it specifies that the requests of an active object are served sequentially up to completion, which is, without interleaving possibilities. In this case, data races and race conditions, as defined in Section 2.1, are completely avoided. ASP features this scheduling model. The drawback of this model is that deadlock are likely to arise in the presence of reentrant requests. So the application must be designed according to this constraint.

Cooperative Scheduling Model. A major development in active object request scheduling, introduced first by Creol, consists in having the possibility to release the single thread of control of an active object, explicitly, at the application level. Cooperative scheduling represent a solution to the need for a controlled preemption of requests. The cooperative scheduling model is based on the idea that a request should not block the progression of another request if it temporarily cannot progress itself. For that, a running request can explicitly release the execution thread before completion, based on some condition or unconditionally, in order to let another request progress. In this model, requests are not processed in parallel, but they might interleave. Consequently, data races are avoided, as a single thread is running at a time, but race conditions can occur because the execution of requests interleaves. Here, contrarily to the mono-threaded scheduling model, the result of the execution does not only depends on the request execution order, but also on the order in which futures are resolved. As a result, much more execution possibilities exist in the cooperative scheduling model. The possibilities are also increased depending on the request activation mechanism, whether the scheduler favours starting or resuming requests, or if it activates one them not deterministically. The cooperative scheduling model takes its inspiration in the concept of coroutines, that enable multiple entry points for suspending and resuming execution. Coroutines are opposed to subroutines in the sense that subroutines have a single entry point at the beginning and a single exit point at the end of the routine. Coroutines were first introduced in [Con63] and first implemented in a programming language in Simula 67 [DMN68] for concurrent system simulation, using a single thread of control. Since then, many modern languages offer a native support for coroutines, such as C#, Erlang, Haskell, JavaScript and Python, but not all of them: for example Java. In this case, implementing cooperative scheduling is made more difficult.

Multi-threaded Scheduling Model. As computer processors were becoming multi-cores, the hardware started to be adapted to several simultaneous flows of control. In return, high-level programming models have to adapt to this evolution in order to map to the hardware at best. In the multi-threaded scheduling model for active objects, several threads are executed in a parallel manner inside a same active object. In this category, we distinguish two kinds of multi-threading. In the first kind of multi-threading, contrarily to the cooperative scheduling model, multiple requests can be allowed to progress at the same time. Each request runs to completion without yielding the thread they have been assigned. In this case, data races and race conditions are possible if the requests processed in parallel manipulate the same part of the memory (e.g. the active object fields). However, data races can only occur within an activity because activities are still isolated from each other. MultiASP, a key component of this thesis, presented in details in Section 2.4.3, is based on the multi-threaded scheduling model. Hereafter, we call the active objects featuring this scheduling model *multiactive objects*.

> The second kind of multi-threading that can apply in active object languages is when data-level parallelism is allowed inside a request. In this case, the requests may not execute in parallel, but a single request processing may use more than one thread. This kind of multi-threading can also lead to data races, but only within a single request. Encore, presented in Subsection 2.3.6, features this particular kind of multi-threading.

In the context of actors, many works extend the actor model in order to enable parallelism within an actor. Parallel Actor Monitors (PAM) [STDM14] offer a parallel message processing approach in which an actor can be associated to a monitor that decides if messages can be processed in parallel. The parallelisation decision can be user-defined to enable fine-grain filtering. It is claimed that PAM save the programmer from having to rewrite data parallelism into task parallelism, as it should be done with regular actors. Other works propose the unification of the actor model with other parallel models to achieve intra-actor parallelism. In [IS12], the `async-finish` task-parallel model is embedded in an actor so that it can be used in the body of a function that processes a message. This reunification enables strong synchronisation at the `finish` stages, where bare actors cannot easily synchronise due to the non deterministic order on the messages. In [Hay+13; HSF15], several parallel message processing strategies that mix transactional memory with the actor model prove to increase message throughput, either without failing the actor semantics (in an optimistic approach for low contention workloads) or by avoiding transactional memory rollbacks if inconsistent snapshots only include readers (in a pessimistic approach for high contention workloads).

2.2.3 Transparency

Whether the programmer is aware of concurrency aspects and whether the programmer is able to manipulate them through language constructs is a key point of comparison of active object languages. Besides, the transparency of the language is the aspect that impacts the most the end users of active objects: the programmers. The active object programming model is designed such that it integrates well with object-oriented programming. Nevertheless, the level of integration can substantially vary depending on the active object language design and of the underlying object-oriented paradigm. In case of fully transparent active object language, the programmer manipulates active objects as regular objects, without any difference. In this case, points of desynchronisation and resynchronisation are completely hidden from the programmer by the active object runtime. In the other cases, some language constructs are given to the programmer to explicitly mark either desynchronisation or resynchronisation, or both. In this case, the programmer is responsible for a good placement of those language constructs. We review below concrete situations where the transparency of an active object language can be discussed.

Transparency of asynchronous method calls. In active objects, remote method invocations are the points of desynchronisation in an execution flow, where the invoker continues its execution concurrently with the request processing. In some active object languages, like Creol, such points of desynchronisation are explicitly placed in the program by the programmer. In this case, a special syntax is used: where usually a dot is the symbol of synchronous method invocation, separating the object instance from the called method, another symbol is used to mark an asynchronous method call. For example, Creol and ABS use the exclamation mark (`!`) in order to distinguish asynchronous method calls from synchronous ones. On the other hand, some other active object languages offer transparent asynchronous method calls: the same syntax is used both for synchronous and asynchronous method calls. In this case, the active object runtime distinguishes the two kinds of method calls thanks to the nature of the invoked objects: if it is an active object, then the method call will be asynchronous; if it is a passive object, then the call will be synchronous. Further local optimisations can be made from this basic rule. ASP is an example of active object language that features transparent asynchronous method calls.

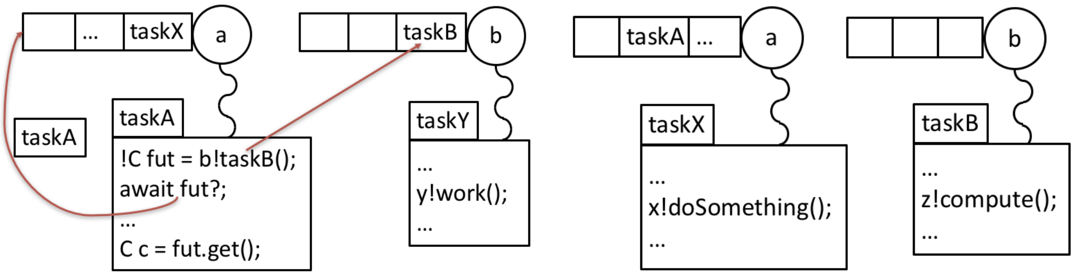
Transparency of futures. In all active object languages, the asynchronous invocation of a method is coupled with the creation of a future. And, in most active object languages, a future is implemented as an object. However, active object languages differ whether they show to the programmer the future object, through a dedicated type, or if they hide the future thanks to inheritance techniques. Often, when futures are explicit, a dedicated type exists and it is parametrised with the enclosed type, as in `Fut<Int>` (ABS syntax). Additionally, when futures are statically identified, they can also be accessed and polled explicitly. A dedicated operator, like `.get` in ABS, allows the programmer to retrieve the value of the future. If the future is not yet resolved when accessed through the operator, the execution flow is blocked until the future's value is computed. Explicit futures also allow the programmer to explicitly release the current thread if a future is not resolved. This is implemented with a particular keyword, usually named `await`. This possibility is strongly tight to the cooperative scheduling model described in Subsection 2.2.2 above. On the other hand, in active object languages, futures can also be implicit: the programmer does not see them in the program, as futures are only identifiable at runtime. In case of implicit futures, the expressions that need the future's value to be computed (as knows as *blocking* operations) automatically trigger a synchronisation: the program is blocked at this point until the future is resolved. This behaviour is known as *wait-by-necessity*. In some active object implementations, futures can be given as parameters of requests without resolving them before. Futures that have this ability are called *first class* futures. In practice, explicit or implicit futures can be implemented as objects that can be globally accessed through proxies. All activities to which the future has been passed can then access the future through its proxy.

Transparency of distribution. As not all active object languages offer a support for distributed execution, we focus here on the ones that can execute their program across distributed resources connected through a network. In this case, a major question that remains for the design of a distributed active object language is whether the programmer is aware of distributed aspects, and if distribution appears in the program. This is not specially related to active objects but to the level of abstraction of the distributed layer. For example, if the program must deal with Internet Protocol (IP) addresses in order to contact a remote active object, then the transparency of distribution of the language is very low. On the contrary, if active objects are addressed like regular objects (by various means: distributed global memory, abstract deployment layer, indirection, etc.) even if they lie on the other side of the world, then the active object languages has a transparent distribution. With such active object languages, there is almost no difference between a distributed program and usual objects. This is what ProActive (detailed in Subsection 2.3.4) offers. Transparent distribution is an important point to take into account as abstracting away distribution greatly facilitates the implementation of advanced features of active objects, such as automatic transmission of future references and automatic future updates (related to first class futures). Defining a level of abstraction for distribution also defines the semantic options offered by the language. For example, several active object languages that target distributed execution forbid synchronous method calls on remote active objects (e.g. AmbientTalk, presented in Subsection 2.3.5).

Overall, the active object languages that are not transparent facilitate static verifications on the program, for example in order to statically detect deadlocks. Also, having explicit constructs for the aspects that are related to active objects makes the programmer aware of where futures are created. They offer to the programmer a better control on execution, but also assume that the programmer has experience with the key notions of active objects. On the other hand, transparent active object languages are handier and more accessible to programmers, as parallel sections are spawned automatically, and as joining phases hold automatically as well. In return, transparent active object languages must be equipped with a sophisticated runtime, in order to detect the points in the program where the behaviour should differ from standard object-oriented execution, for example when a wait-by-necessity must be triggered out of a simple object access expression, or when an asynchronous method call must take place instead of a synchronous one.

2.3 Overview of Active Object Languages

In this section, we review the main active object languages that have arisen in the past ten years, and that have inspired the newest active object languages. Voluntarily, the presented active object languages are instances of the various



(a) Step 1. (b) Step 2.

Figure 2.2 – Cooperative scheduling in Creol.

kinds of active objects that we have introduced in the classification of Section 2.2. As mean of syntactic comparison, we show a same code snippet for most of the presented active object languages, in order to emphasise their features. The given example gives room for using most of the language constructs that are important in the context of this thesis. Finally, we present major actor systems because they are also a relevant source of inspiration for the design of active object languages.

2.3.1 Creol

Creol [JOY06] is an uniform active object language in which all objects are active objects: they all come with a dedicated execution thread. The full semantics of Creol with minor updates is presented in [BCJ07]. Creol objects are perfectly outlined so that no other thread than the object's execution thread can access the methods (and the fields) of the object. Consequently, objects can only communicate through asynchronous method calls and futures; no data is shared between objects. In Creol, asynchronous method calls and futures are explicit, but futures are not first class. Creol is based on a cooperative scheduling of requests, which softens its strict object partitioning. Figure 2.2 shows an example of execution of a Creol program in two steps. The example involves a cooperative scheduling of requests by using the `await` language construct.

The `await` Creol keyword can be used by the programmer in a method's body in order to yield the execution thread if the future on which the `await` is applied is not resolved. In Figure 2.2a, object `a` does an asynchronous method invocation on object `b`, and then awaits for the result. When the `await` instruction actually releases the execution thread because the future is not resolved, another request for this object can start (if it is in the request queue) or can resume (if it was deactivated by a previous call to `await`). In Figure 2.2b, while `taskA` is put back in the queue of object `a`, `taskX` gets activated. In Creol, when a request is deactivated or finished, the choice for the next request to start or resume is not deterministic. Finally, a `.get()` call can be used to eventually retrieve a future's value, like in Figure 2.2a at the end of `taskA`. Contrarily to `await`, `get` is a blocking construct, so the current request is never deactivated upon a `.get()` call and also, no other request is activated if the future is still not resolved at this point.

As mentioned in Subsection 2.2.2, Creol avoids data races thanks to its cooperative scheduling of requests offered by the `await` language construct. But in practice, interleaving of requests in a Creol program can be quite complex, as release points are highly needed to avoid the deadlocks that arise from circular request dependencies. Overall, the goal of Creol is to 'reencapsulate' the object state that can be infringed in multi-threaded environments. It achieves this goal by having a rich and precise active object language. However, the drawback of Creol is that its safety of execution strongly depends on how well the programmer places the release points in the program. Indeed, not enough release points would lead to deadlocks whereas too many release points would lead to complex interleaving of requests, which could violate the semantic integrity of the object's state.

2.3.2 JCoBox

JCoBox [SPH10] is an active object programming model, together with its programming language implementation in Java. JCoBox mixes the non uniform and the object group model presented in Subsection 2.2.1). The objects are organised into groups, called CoBoxes, such that each CoBox is responsible for maintaining its own coherent concurrency model. At each time, each CoBox has one single thread that is actively processing a request for any of the active objects contained in the CoBox. However, there might exist multiples other threads (plain Java threads) in the CoBox, but in this case, all they can do is wait for an unresolved future. A CoBox contains two kinds of objects: first, active objects, called *standard objects*,

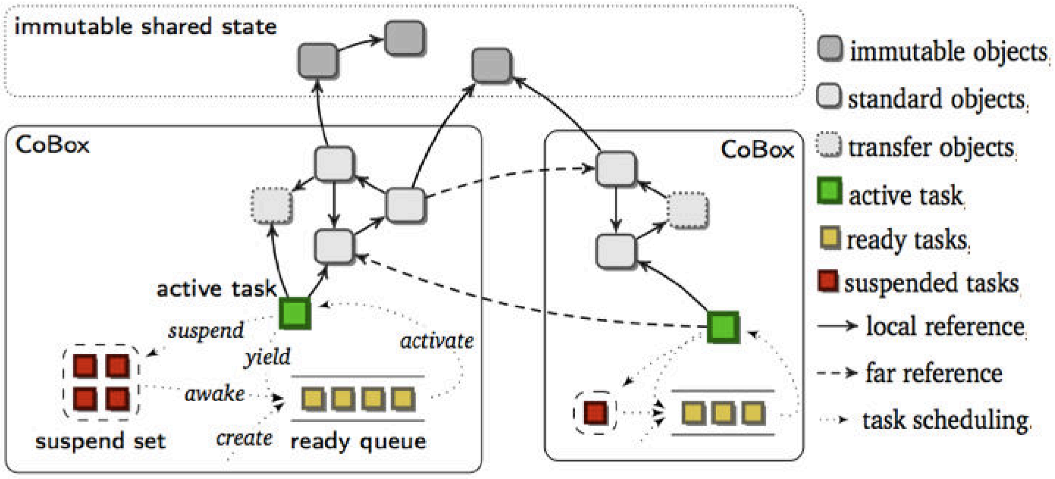


Figure 2.3 – Organisation of JCoBox objects.

that are accessible from other CoBoxes via asynchronous method calls, and second, passive objects, called *transfer objects*, that are local to a CoBox. In addition, JCoBox offers a third kind of objects: *immutable objects*. Immutable objects are passive objects that can safely be shared between active objects. Figure 2.3, taken from [SPH10], represents all objects kinds and their permitted communications, as well as the cooperative scheduling featured in each CoBox. Like in Creol (Subsection 2.3.1 above), asynchronous method calls and futures are explicit. Requests are executed according to the cooperative scheduling model. The methods `get()` and `await()` are used on future variables to respectively retrieve a future's value in a blocking way and to release the execution thread in the case where a future is not resolved. Additionally, JCoBox defines a static `yield()` method in order to unconditionally release the execution thread. The `yield()` method can optionally take as parameter a minimum waiting time before being rescheduled. To summarise, JCoBox interleaves the request execution of all the objects lying in a same CoBox, where Creol only interleaves the requests for a single object. However, contrarily to Creol, that executes requests not deterministically, JCoBox enforces a FIFO requests execution policy, where incoming requests and ready-to-resume requests are put together in a *ready queue*. From an implementation point of view, both blocked (by a `get()`) and descheduled (by an `await()` or a `yield()`) requests are put in the ready queue once they satisfy the resuming condition. However, de-

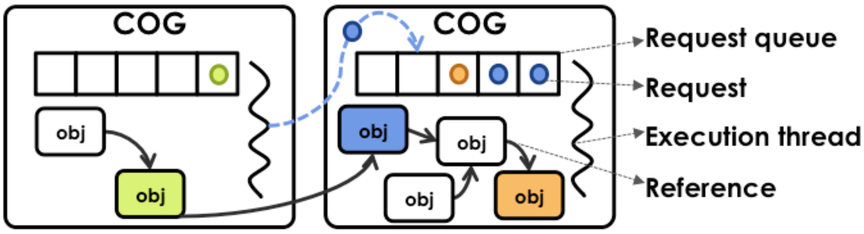


Figure 2.4 – An example of ABS program execution

scheduled requests are appended whereas blocked requests are prepended to the ready queue, in order to enforce the desired semantics.

JCoBox has a prototyped runtime for the support of distributed execution using Java Remote Method Invocation [WRW96] (RMI) as the communication layer. In this case, the unit of distribution is the CoBox. Generally, JCoBox better addresses practical aspects than Creol: it is integrated with Java, its design tolerates distribution, and the object group model improves the scalability in number of threads. The interleaving of request services in JCoBox is similar to the one of Creol; request services can even be more interleaved due to object groups. Thus, the request scheduling of JCoBox has the same advantages and drawbacks as Creol.

2.3.3 ABS

The Abstract Behavioral Specification language (ABS) [Joh+11] is an object-oriented modelling language based on active objects. ABS takes its inspiration in Creol for the cooperative scheduling and in JCoBox for the object group model. ABS is intended at modelling and verification of distributed applications. The object group model of ABS is based on the notion of Concurrent Object Group (COG), that partition the objects of an application into several COGs. A COG manages a request queue and a set of threads that have been created as a result of asynchronous method calls to any of the objects inside the COG. The COG granularity ensures that only one thread among the managed threads is executing (also said *active*) at a time. Figure 2.4 shows an ABS configuration with a request sending (dotted line) between two COGs.

In an ABS progam, new objects can be instantiated in a new COG with the `new` keyword. In order to instantiate an object in the current COG, the `new local` keyword combination must be used. As in Creol and JCoBox, ABS features an `await` language construct, that unschedules the thread if the specified future is unresolved and a `get` language construct, that blocks the thread execution until the specified future is resolved. In ABS, `await` can also be used to pause on a condition. In addition, there is a `suspend` language construct (counterpart of `yield` in JCoBox) that unconditionally releases the execution thread. Contrarily to JCoBox, ABS makes no difference on the objects' kind: all objects can be referenced from any COG and all objects can be invoked either synchronously or asynchronously. Moreover, all kinds of object invocation fall in the request queue of the enclosing COG. Asynchronous method calls and futures are explicit.

```
BankAccount ba = new local BankAccount(459818225, Fr);
...
TransactionAgent ta = new TransactionAgent(ba);
WarningAgent wa = new WarningAgent(ba.getEmail(), ba.getPhone());
...
Transaction dt = new DebitTransaction(42.0, Eur);
Fut<Balance> bfut = ta!apply(dt);
await bfut?;
Balance b = bfut.get;
wa!checkForAlerts(b);
...
b.commit();
```

Listing 2.1 – Bank account program example in ABS.

Listing 2.1 shows an ABS code snippet. The example consists of an (active) agent that handles the transactions on a bank account, and another (active) agent that is responsible for triggering alerts, based on the transactions that happened. As the transaction agent and the warning agent are supposed to be highly loaded, they are instantiated in their own COG (`new` is used instead of `new local`). Note that, in this program, we could wait and retrieve the future variable after the call of `checkForAlert`, passing the variable `bfut` instead of `b`. But for that, the `checkForAlerts` method signature must be changed from `checkForAlert(Balance)` to `checkForAlert(Fut<Balance>)`, which might not be convenient for other usage of `checkForAlerts`. This code snippet would be similarly implemented in Creol and JCoBox, this is why we did not present it in the respective subsections.

ABS comes with numerous engines[1] for verification of concurrent and distributed applications and their execution. Below is a list of the verification tools provided for ABS:

- A deadlock analyser [GLL15] allows the programmer to statically detect deadlocks in ABS programs, thanks to an inference algorithm that is based on a behavioural description of methods.
- A resource consumption analyser [JST15] enables the comparison of application-specific deployment strategies
- A termination and cost analyser, COSTABS [Alb+12], that has two parts. The first ability of this tool is to find deadlocks in an ABS program, in a different way than [GLL15] by the use of a solver. The second purpose of COSTABS is to evaluate the resources that are needed to execute an ABS program. In this case, computer resources are abstracted away through a model with quantifyable measures. COSTABS has also been generalised for concurrent object-based programs [Alb+14]
- A program verifier, KeY-ABS [DBH15; Din+15], allows the specification and verification of general, user-defined properties on ABS programs. KeY-ABS is based on the KeY reasoning framework [BHS07].

In addition to verification tools, ABS tools also comprise a frontend compiler and an Integrated Development Environment (IDE) support through an Eclipse plugin for ABS programs. In addition, several backends translate ABS programs into various programming languages, including into the Maude system, Java, and Haskell [BB16]. The Java backend for ABS translates ABS programs into local Java programs that enforce the ABS semantics. The Haskell backend for ABS performs the translation into distributed Haskell code. The ABS semantics is preserved thanks to the thread continuation support of Haskell, which is not supported on the JVM. A Java backend for ABS based on Java 8 [Ser+16] is currently under development and experiments different approaches to encode thread continuation

[1] ABS related tools can be found at: http://abs-models.org/

on the Java Virtual Machine (JVM). In Chapter 4, we present another backend for ABS that specially targets distributed High Performance Computing (HPC) with ProActive (Subsections 2.3.4 and 2.4.2). This backend is fully implemented and, moreover, the correctness of the translation is formally proven. The implementation of the ProActive backend for ABS is compared with the design of the Java 8 backend for ABS in [1].

2.3.4 ASP and ProActive

ASP [CHS04] is an active object programming language tailored for distributed computing. ASP has proven properties of determinism, and particularly fits the formalisation of object mobility, groups, and componentised objects [CH05]. ASP follows a non uniform active object model with high transparency: active and passive objects are almost always manipulated in the program through the same syntactic constructs. Contrarily to the cooperative scheduling model of Creol, JCoBox, and ABS, in ASP a request cannot be punctuated of release points. Once a request starts to be executed, it runs up to completion without ever releasing the thread. Synchronisation is also handled automatically: futures are implicitly created from asynchronous remote method calls. In practice, future types are dynamically created and inherit the return type of the method that was called asynchronously. Futures are, as such, completely transparent to the programmer. ASP features the wait-by-necessity behaviour upon access to an unresolved future: the program execution automatically blocks when a future's value is needed to pursue the program execution. A wait-by-necessity is triggered when the future is an operand of a strict operation, that is when, not only the reference of the future is needed, but also its value. ASP has first class futures, which means that futures can be passed between activities without being resolved beforehand. This happens when futures are not part of strict operations, for example when a future is a parameter of a local or remote method call. Indeed, as futures are transparent, they are also transparently passed between activities. When the future is resolved, its value is automatically updated for all activities it has been passed to. Several future update strategies have been explored in ProActive for this purpose [Hen+11]. ASP ensures causal ordering of requests by enforcing a *rendez-vous* upon all asynchronous communications. When a caller asynchronously invokes method on a callee, the caller is only allowed to continue its execution when the request has been successfully put in the queue of the callee. Thus, two successive asynchronous method calls are always put in the request queue of the recipient in a causal order. This characteristic enforces FIFO point-to-point request channels, and makes the semantics of ASP programs very close to the one of a sequential execution. This latter point allows the programmer to predict the behaviour of the program more easily.

ASP forms the theoretical foundation of ProActive [Bad+06], the implementation of ASP in Java. The active object model [LS96] and the distributed object model of Java (Java Remote Method Invocation (RMI)) [WRW96] were released the same year in seminal papers. ProActive has combined those two models and has implemented the semantics of ASP in a Java library that offers full support for distributed execution. As the active object model of ASP is transparent, ProActive active objects follow as much as possible the syntax of regular Java objects. The only difference is when an active object is created: a static method of the library, named `newActive`, must be used for that. Passive objects are created as standard Java objects with the `new` keyword.

```
BankAccount ba = new BankAccount(459818225, Country.FR);
...
TransactionAgent ta = PAActiveObject.newActive(
                              TransactionAgent.class, new Object[]{ba}, node);
Object[] warningParams = {ba.getEmail(), ba.getPhone()}
WarningAgent wa = PAActiveObject.newActive(
                              WarningAgent.class, warningParams, node);
...
Transaction dt = new DebitTransaction(42.0, Currency.EUR);
Balance b = ta.apply(dt); // dt is deeply copied
wa.checkForAlerts(b); // if b is a future it is passed transparently
...
b.commit(); // a wait-by-necessity is possible here
```

Listing 2.2 – Bank account program example in ProActive. *node* is not defined here.

Listing 2.2 shows the transaction agent example written in ProActive. Again, the transaction agent and the warning agent have their own thread of control, and in ProActive, they can be settled on different machines. The transaction agent and the warning agent are two active objects created through the call to `PAActiveObject.newActive` with three parameters: the class to instantiate, the parameters of the constructor, and optionally, the node on which the active object must be deployed (ProActive supports remote instantiation of active objects). As opposed to ABS, we can notice in the example that first class futures are completely transparent: line 11 will proceed even if `b` is an unresolved future. Thus, the ABS and ProActive programs have in fact a slightly different semantics. In Chapter 4, we will see that an updated version of ASP, namely MultiASP, introduced in Subsection 2.4.3, allows us to encode ABS programs in ProActive, giving exactly the same semantics.

In ProActive, when an active object is created, it is registered in the Java RMI registry, RMI being the main communication layer used in ProActive. ProActive uses the RMI registry in order to have active objects accessible through the network and identified by Uniform Resource Locator (URL). When an RMI URL is requested, the RMI registry returns a Java object that is a proxy to the requested active object. The proxy encapsulates network communication and serialisation mechanisms, in order to remotely invoke methods on the active object. Proxies enable transparent distributed program execution. In practice, ProActive active objects are always manipulated through proxies that delegate all asynchronous invocations to the active object. In the example given above, the references returned by the calls to `newActive` are references to the local proxies of the remote active objects.

One aspect of ProActive is also dedicated to components [BHR15]. ProActive active objects form a programming model that is suitable for component-based composition of distributed applications through the GCM[2] model. The Vercors platform [HKM16] enables the design, specification and verification of ProActive components through an Eclipse plugin, similarly to the verification abilities of ABS.

ProActive is intended for distribution, it forms a complete middleware that supports application deployment on distributed infrastructures such as clusters, grids

[2] Grid Component Model

and clouds. The deployment mechanism of ProActive, named GCM Deployment, is based on the concept of virtual nodes: a virtual node is an aggregation of physical machines that are declared in configuration files (XML files). Such virtual nodes can be denoted in a ProActive program through identifiers. This allows the programmer to choose indirectly a location on which to deploy an active object. The ProActive middleware has proven to be scalable and suitable for distributed HPC [Ame+10]. It is the main technology used as a basis to implement the works that are presented in this thesis. This first description of ProActive is enriched in Section 2.4, where the multi-threaded extension of ASP, MultiASP, is presented together with its implementation in ProActive.

2.3.5 AmbientTalk

AmbientTalk [Ded+06; Cut+07] is a distributed actor-based programming language targeting distributed execution in mobile ad hoc networks. Although AmbientTalk would rather work with message passing instead of Remote Procedure Call (RPC), we liken it to an active object language because the concurrency layer is highly entangled with object-oriented notions, and because it makes an advanced use of futures, specially for dealing with network failures. So, in the following, as we speak of active object, AmbientTalk usually use the term actor. AmbientTalk is inspired from the E actor-based programming language [MTS05] for most of the concurrency layer. AmbientTalk follows both a non uniform model and an object group object model (see Subsection 2.2.1): it features active and passive objects. Asynchronous invocations are explicit and return dynamically typed futures (object typing is dynamic in AmbientTalk). Only passive objects that are owned by the same active object can communicate through synchronous message passing (with syntax `o.m()`). AmbientTalk follows a mono-threaded scheduling model (see Subsection 2.2.2), where request processing is atomic: requests are executed one after the other and always run up to completion. Yet, an AmbientTalk program execution never leads to a deadlock because the execution flow never stops. The reason for deadlock-freedom in AmbientTalk is that a future access is a non-blocking operation: accessing an unresolved future results in an asynchronous call that returns another future. In our context, this is the most atypical aspect of AmbientTalk.

This way, even unresolved futures can receive messages intended to the not yet computed future's value. In order to implement this behaviour, futures are implemented like active objects, where the messages are accumulated in a queue until the future becomes resolved. Moreover, a callback can also be attached to a future in order to do something with the future's value as soon as it is available. In a sense, callbacks on futures can be likened to the first class futures of ASP.

```
when: contactFut becomes: { |contactInfo|
  // execution is postponed until future is resolved
  system.println("Found item, contact: " + contactInfo);
} catch: { |exception| ... };
// code following when: is processed immediately
```

Listing 2.3 – AmbientTalk when:becomes:catch clause example.

Listing 2.3, taken from [Cut+07], shows an example of a future's handler, where the callback is located in the `becomes:` block. The event-based execution of the different activities makes sequences of actions difficult to enforce in AmbientTalk. However, an AmbientTalk program is always partitioned into separate event handlers that maintain their own execution context; this is known as an *inversion of control*. Thus, reasoning on AmbientTalk programs is a bigger challenge compared to the other active object languages presented in this thesis.

```
def ba := BankAccount.new(459818225, Fr);
...
def ta := actor: {
  def myBankAccount := ba;
  def apply(debitTransaction) {
    ...
  };
};
def wa := actor: {
  def email := ba.getEmail();
  def phone := ba.getPhone();
  def checkForAlerts(balance) {
    ...
  };
};
...
def dt := DebitTransaction.new(42.0, Eur);
def b = ta<-apply(dt);
wa<-checkForAlerts(b);
...
b.commit();
when: b becomes: { |resB|
  system.println("Balance is computed. Commit can proceed.");
} catch: { |e|
  system.println("Exception: " + e);
};
```

Listing 2.4 – Bank account program example in AmbientTalk.

Listing 2.4 represents the bank account example written in AmbientTalk. The active objects (characterised with the `actor:` definition in the program) are declared inline: they are created at the same time as their behaviour is defined.

As ProActive, AmbientTalk is primarily made for distributed execution. Unlike JCoBox, the support for deployment on distributed infrastructures is much experienced. In particular, AmbientTalk features a dynamic active object discovery mechanism, based on the *publish/subscribe* pattern, that allows unexpected resources to be a part of the application. AmbientTalk also deals with unexpected disconnections of distributed resources by the mean of leasing, that denotes a limited access in time to an object. The lease is automatically renewed when objects communicate. Considering its resilient approach, AmbientTalk is probably the active object language that is the most representative of active objects as a service. Finally, AmbientTalk programs run on the JVM and, as such, must comply to the JVM constraints. In return, AmbientTalk can hit a potentially large audience thanks to this fact.

2.3.6 Encore

Encore [Bra+15] is an active object-based parallel language that is currently under development. Encore is inspired from the works carried out around the Joëlle programming language [Cla+08], oriented towards a theory and practice of object ownership. The philosophy of Encore is to provide a programming language that is parallel *by default*, and that relies on the active object programming model mixed with other parallel patterns. Encore is essentially based on a non uniform object model and a cooperative scheduling model (see Subsections 2.2.1,2.2.3). Encore features active and passive objects. Method calls are transparently asynchronous or synchronous, depending whether the called object is active or passive. Although futures are typed dynamically, their value must be explicitly retrieved via a get construct. Unlike the other presented active object languages, Encore natively includes two forms of asynchronous computation: asynchronous method calls (like the other languages) and asynchronous parallel constructs inside a request. Internal parallelism can be explicit through async blocks, or it can be implicit through *parallel combinators*, an abstraction that spawns Single Instruction Multiple Data (SIMD) tasks and joins them automatically. What is special about Encore is that all parallel constructs are unified with the use of futures for handling asynchrony [FRCS16]. In Encore, active objects encapsulate passive objects, in terms of *ownership*, but unlike ProActive and AmbientTalk, passive objects can be shared by reference across the activity boundaries. In this context, in order to prevent concurrent modifications on passive object, the programmer gives to a passive object a *capability* type, that defines both the accessible interface of the object and the level of accessibility of this interface. The capability possibilities include not exhaustively: exclusive access from one control thread, optimistic and pessimistic sharing, and unsafe sharing. The capability system of Encore has been formalised in [CW16]. Regarding request scheduling, Encore features the same cooperative scheduling as Creol and ABS with the await and suspend language constructs.

```
let ba = new local BankAccount(459818225, Fr); // BankAccount is a passive class
in {
  let
    ta = new TransactionAgent(ba);
    wa = new WarningAgent(ba.getEmail(), ba.getPhone());
    ...
    dt = new DebitTransaction(42.0, Eur);
  in {
    let bfut = ta.apply(dt)
    in {
      await bfut;
      let b = get bfut
      in {
        wa.checkForAlerts(b);
        ...
        b.commit();
      }
    }
  }
}
```

Listing 2.5 – Bank account program example in Encore.

Listing 2.5 shows the bank account example written in Encore. Here, we suppose that the BankAccount and DebitTransaction classes are passive, and that TransactionAgent and WarningAgent are classes declared without special keyword (and in this case, ta and wa are automatically active objects). The manipulation of futures is similar to what can be found in Creol or ABS programs. But in addition, Encore provides a chaining operator $\leadsto$ that adds a callback to a future, in order to execute it when the future is resolved. This chaining operator is somehow similar to the callbacks of AmbientTalk. Cooperative scheduling and future chaining mix explicit and automatic synchronisation, which might save the programmer from the burden of precisely placing all the release points in the program.

An Encore program is compiled through a source-to-source compiler, written in Haskell, that produces C code complying to the C11 standard. Consequently, all tools that apply to C programs can also be applied to a compiled Encore program. In conclusion, Encore is oriented towards massively parallel execution, but its design makes it not adapted for high performance distributed execution, due to the predominance of object sharing.

2.3.7 Actor Languages and Frameworks

Like active objects, since the publication of the original actor model in [Agh86], many implementations of actors have emerged, along with the needs of the programmers throughout the years. These implementations have various characteristics that differentiate them from each other, and that makes them adapted to particular contexts. An informative study [KSA09] compares several actor frameworks that execute on the JVM platform and discusses their guarantees. We review below the actor languages and frameworks that execute on various platforms and whose characteristics are remarkable in our context.

Rebeca

Rebeca [Sir+04] is an actor-based programming language featuring classical asynchronous message passing without reply between actors. Rebeca aims at using actor-based concepts for the specification and model-checking of reactive systems. The only difference of Rebeca against the original actor model is that Rebeca actors preserve the order of messages that are sent between two actors. This communication model is known as FIFO point-to-point communications, as in ASP. However, contrarily to ASP, message sending is non blocking. Consequently, upon reception, messages can be reordered depending on their timestamps. Each Rebeca actor encapsulates its own variables, so there exists no data sharing between actors. In addition, the execution model of Rebeca actors is mono-threaded, which makes Rebeca programs intrinsically data race free. As ABS, Rebeca has many frontend verifier tools, and several backend translators into various languages. One of the backends translate Rebeca models into the Erlang programming language, that has a solid background and support for concurrent programming [VWW96]. Rebeca also supports componentised model-checking [Sir+05], and an extension of Rebeca enables verification of Rebeca models in the presence of timed constraints [Kha+15].

Scala and Akka actors

Scala actors [HO09] are pioneers and ones of the most successful actors that participated to their popularisation. Their impact on the developement of support for concurrency and on newly designed technologies is nowadays as much visible in education as in industry [Hal12]. A more recent implementation of actors in Scala is available in the Akka library [Inc12], that mainly improves the performance of distributed actors on the JVM. This is mainly due to a better implementation of serialisation compared to standard Java serialisation. Akka actors also offer a facilitated way to distribute the execution of actors, compared to Scala actors, by making the deployment more transparent. Both Scala and Akka actors support the use of futures for the convenience of the programmer, although this makes a significant difference with respect to the original actor model.

Kilim

Kilim [SM08] is an actor implementation that provides lightweight threads on top of Java. To this end, actor's threads are cooperatively scheduled, which makes the execution of Kilim actors slightly different from the execution of actors in the original model. Also, Kilim allows the message content to reference objects that are outside the actor's boundaries. Such object sharing can lead to inconsistent state. It is then on the shoulders of the programmer to realise a copy of the shared object, or to protect it accordingly. An extension of Kilim [GB13] enables ownership-based isolation of objects in order to better structure object accesses. Another aspect of Kilim is that, in order to drop a message to an actor, one must get the local reference of the actor's mailbox. This has the drawback of breaking the actor's encapsulation and of relying on local references to target an actor. All of these properties make Kilim handy and efficient but not yet adapted to distribution.

SCOOP

The Maude model and the C runtime of SCOOP [Mor+13], that stands for Simple Concurrent Object-Oriented Programming, provides a concurrency model based on the notion of *handlers*. A handler is an autonomous thread of control that is able to execute actions on an object. In SCOOP, method calls on an object reference are performed asynchronously if they are placed in a special `separate` block, that mentions this object reference. What distinguishes SCOOP from other actor models is that a whole sequence of actions can be registered to an object's handler. For that, the actions just need to lie in a same `separate` block. The sequence of actions that are in a `separate` block is ensured to be delivered in order. Additionally, these actions are ensured to be executed atomically, which is, without interleaving with actions from other `separate` blocks. Although SCOOP is well studied and optimised for local concurrency [WNM15], it also starts to be adapted to distributed execution [SPM16].

Salsa

Salsa [DV14] is an actor-based programming language that targets distributed computing above all goals. Salsa is implemented in Java and can be used as follows. A Salsa program must be first compiled into Java code with a dedicated compiler, and then compiled into Java bytecode through a standard Java compiler. Although both Kilim and Salsa are made for running actors on the JVM, they pursue a radically different goal. Indeed, Kilim is oriented towards efficient multicore computing and Salsa is oriented towards distributed execution. Salsa allows no shared state between actors and offers a location-transparent distribution of actors, based on the notion of universal naming. It targets grid computing, mobile computing and internet-based computing. This illustrates well how the objective of the language can lead to completely different implementations of the actor model.

Java 8-based actors

New language constructs based on functional programming have been introduced lately in the version 8 of Java. Even though this cannot change the possibilties of the underlying execution platform of Java, namely the JVM, recent works [NB14; Ser+14; Ser+16] investigate whether those new constructs make Java more adapted to the implementation of actor frameworks, and make optimistic conclusions. Before Java 8, a lot of high level parallel constructs, such as lightweight asynchronous tasks, barrier, phasers and transactions, required the modification of the compiler, hence to have a language-based approach. The Java 8 runtime allows most of these constructs to be directly embodied without intricate underlying implementation.

Habanero Java actors

Habanero-Java (HJ) [Cav+11] is a parallel programming language built on top of Java and based on the X10 programming language [Cha+05]. HJ is primarily oriented toward portable multicore computing through the provision of parallel constructs and of lightweight tasks that are missing from Java. A new implementation of HJ, HJ-lib, has been written using Java 8 [IS14]. In particular, this new version uses closures through Java 8 lambda expressions for the safe implementation of the former parallel constructs, and also led to an implementation of actors.

The execution of HJ-lib actors deviate from the original actor model in the sense that an actor can be multi-threaded if a message contains parallel constructs. For example, it is possible with HJ-lib to have an actor processing a message using an `async` call.

Table 2.1 summarises the active object languages that have been presented in this section. Their main features are highlighted and placed according to the classification given in Section 2.2. Many mixed combinations of preponderant features have been experimented through each language, especially for the transparency of asynchronous method calls and futures. Yet, through this table we can notice the emergence of categories of language. Creol, JCoBox, and ABS share a lot of common features in the scheduling and transparency aspects. ProActive and AmbientTalk have the same object model and full support distribution, they belong to a same category of language where their models that are adapted to distribution. Finally, Encore would rather join the cooperative-based languages but its wide range of possibilities makes it unfitted to any of the categories.

2.4 Focus on Multi-threaded Active Objects

In this thesis, we mainly contribute to the multi-threaded active object model, this is why we propose in this section a particular focus on multi-threaded active objects, hereafter multiactive objects. Firstly, we present the multiactive object programming model and then, the way it is implemented in the ProActive library (Subsection 2.3.4). Afterwards, we present MultiASP, the multi-threaded extension of ASP (Subsection 2.3.4), that formalises the implementation of multiactive objects in ProActive.

2.4.1 Multiactive Object Model

As mentioned in Subsection 2.2.2, controlled multi-threading is gaining attention in the actor and the active object communities, in order to embrace the raise of multi-core computer architectures. Multiactive objects [HHI13] are a multi-threaded extension of the active object programming model. The principle of multiactive objects is to enable the execution of multiple requests of an active object

	Creol	**JCoBox**	**ABS**	**ASP ProActive MultiASP**	**AmbientTalk**	**Encore**
Object model	uniform	• object group • non uniform	• object group • uniform	non uniform	non uniform	• non uniform • multiple styles
Scheduling model	cooperative	cooperative	cooperative	mono-threaded (multi-threaded with MultiASP)	mono-threaded	• cooperative • parallel constructs
Transparency	• explicit async method calls • explicit futures • no distribution	• explicit async method calls • explicit futures • early distribution	• explicit async method calls • explicit futures • distribution through backends	• implicit async method calls • implicit futures • full support for distribution	• explicit async method calls • implicit futures • full support for distribution	• implicit async method calls • explicit futures • no distribution

Table 2.1 – Summary table highlighting the main features of active object languages

in parallel, while having control on the concurrency. Such a parallelism proceeds at the request level inside the active object: several requests can be executed at the same time, where the original active object programming model advocates the processing of one request at a time. The request scheduling of multiactive objects differs from cooperative scheduling in the sense that cooperative scheduling cannot make an active object take advantage of multi-core architectures, because it has only one single thread that is active at a time. On the opposite, multiactive objects feature not only multi-tasking but also true parallelism of tasks. An alternative for benefiting from multi-cores could be to have as many threads running as the number of cores of the machine on which they are deployed on. And, for the frameworks that implement logical threads, like Encore and ABS, many more active objects than running threads can fit on the same machine. Indeed such a solution would load all the cores of the machine. Nonetheless, this does not remove the communication overhead that exists when active objects communicate, due to the presence of remote method calls. This is a limitation that multiactive objects overcome thanks to shared-memory between threads.

In order to keep a notion of safety inside a multiactive object, the requests that execute in parallel must be acknowledged by the programmer: the programmer must say beforehand which requests are *compatible* regarding data race freedom and execution ordering. The compatibility of two requests is declared statically (when writing a class), but may depend on dynamic parameters. So, in multiactive objects, the execution safety is partially handed over to the programmer. Nonetheless, it is more accessible to the standard programmer to specify compatibilities rather than manipulating low-level concurrency construct, such as protecting each shared variable and critical section with a lock. This way, the multiactive object programming model preserves the ease of programming of the active object programming model.

2.4.2 Implementation in ProActive

ProActive implements the multiactive object programming model as an extension to the active object implementation presented in Subsection 2.3.4. ProActive offers to the programmer a specification language that allows him to declare compatibility of requests, and thus to have multiactive objects. The specification language used for multiactive object business is based on the Java annotation mechanism, that allows the programmer to add metadata to a Java program. Java annotations can be either intended to compile-time tools or to run time libraries. In the case of ProActive, annotations related to multiactive objects are processed at runtime, which enables dynamic parameters.

In practice, a Java class can be annotated with the multiactive object annotations that are available in the ProActive library. In this case, when an active object of such a class is created (with the static method `PAActiveObject.newActive`), it is interpreted as a multiactive object. This basic rule implies that, if no annotation is found in the class of an active object, then it remains a 'basic' active object; otherwise, it is a multiactive object. ProActive allows the programmer to define request compatibility in three steps:

- First, a `@Group` annotation must be declared on top of a class to define a group of requests. A group is meant to gather requests that have the same concerns (semantic partitioning) and/or the same compatibility requirements (practical partitioning). Methods belonging to the same group must share the same compatibility rules.

- Second, `@MemberOf` annotations can be defined on top of method definitions, in order to make them belong to a group (previously defined).

- Third, `@Compatible` annotations must be used to specify the groups that are compatible, so, in extension, to specify which requests can be run in parallel safely.

Declaring compatibilities of groups instead of methods increases the abstraction level of the specification of request parallelism. If a method is not assigned to a group, then it is compatible with nothing. Also, if the group names specified in the `@MemberOf` and `@Compatible` annotations do not correspond to any group, a warning message is produced, but the execution proceeds anyway. As an example, consider a distributed peer-to-peer system implemented with multiactive objects. A class `Peer` is defined, with methods that deal with: joining a peer in the network, adding a value in the system, retrieving a value, and monitoring the peer. Such a class is displayed on Listing 2.6.

```
public class Peer {

  public JoinResponse join(Peer other) { ... }

  public void add(Key key, Value value) { ... }

  public Value lookup(Key key) { ... }

  public void monitor() { ... }

}
```

Listing 2.6 – Peer class

When creating `Peer` objects with `PAActiveObject.newActive` in ProActive, all of such objects already run concurrently and can receive requests for any of the methods they define. Without further specifications, for each peer all requests are processed sequentially. However, it might be interesting to parallelise the operations that are done within a peer execution. For example, let us assume that monitoring a peer can be done at the same time as any other requests, since it does not modify the peer and it does not rely on the peer's state. Similarly, adding values in parallel should be feasible as long as it does not apply on the same key. On the opposite, joining a peer in the network must be a strict atomic operation, otherwise the peer-to-peer network architecture could be corrupted. All of those synchronisation notions can be expressed with multiactive object annotations, applicable on the `Peer` class.

```
@DefineGroups({
  @Group(name="atomic", selfCompatible=false),
  @Group(name="concurrentRW", selfCompatible=true, parameter="Key",
                                                          condition="!equals"),
  @Group(name="monitoring", selfCompatible=true)
})
@DefineRules({
  @Compatible({"atomic", "concurrentRW"}, condition="!isLocal"),
  @Compatible({"concurrentRW", "monitoring"}),
})
public class Peer {

  @MemberOf("atomic")
  public JoinResponse join(Peer p) { ... }

  @MemberOf("concurrentRW")
  public void add(Key k, Value v) { ... }

  @MemberOf("concurrentRW")
  public Value lookup(Key k) { ... }

  @MemberOf("monitoring")
  public void monitor() { ... }

}
```

Listing 2.7 – Peer class with multiactive object annotations.

Listing 2.7 shows how to map those notions with the ProActive multiactive object annotations. The annotations previously introduced are used to partition the methods of the `Peer` class into three groups of requests, named respectively `atomic`, `concurrentRW`, and `monitoring`. All the methods of the `Peer` class are assigned to a group and, because any not specified combination leads to an incompatibility, `join` requests are forbidden to be executed in parallel with `monitor` requests. The `@Group` and `@Compatibility` annotations define more parameters than just group names. The second parameter of the `@Group` annotation specifies whether the requests of this same group are compatible. For example, the requests of the `atomic` group are not compatible, because we do not want to have two peers joining the same peer at the same time. The `condition` parameter of the `@Group` and `@Compatible` annotations enables dynamic compatibilities: two considered requests are compatible if the evaluation of the specified condition method returns true. The `parameter` parameter of the `@Group` annotation specifies the parameters to be taken into account for the evaluation of the condition method. Here, it is specified that the parameters of the requests that have type `Key` must be taken as parameters of the condition method. For example, in our case we allow `join` requests to be executed in parallel with `add` requests only if the peer must not add the content to its own storage, but just route it to the next peer (the behaviour of `isLocal` is not shown is the code snippet). Evaluation conditions and their parameters enable very flexible and fine grained parallelism, provided that the programmer is able to determine it. In conclusion, a few multiactive object annotations can greatly improve request processing throughput with a low overhead compared to low-level optimisations for parallelism [HHI12]. Overall, programming with multiactive objects can be summarised with the following principles:

- Without multiactive object annotations, a ProActive active object is mono-threaded without any local parallelism nor race condition.
- If some parallelism is desired, compatibility must be declared between groups of requests that can be safely executed at the same time and, for which the execution order do not matter. Compatibility can be statically declared or decided *dynamically* depending on invocation parameters and/or on the object's state.
- If even more parallelism is required, an expert programmer can still declare more methods as compatible and manually protect the access to the shared variables, using a lower-level synchronisation mechanism.

The multiactive object runtime is in charge of interpreting the multiactive annotations. In order to ensure maximum parallelism, multiactive objects enforce a FIFO policy with possibility to overtake. More precisely, within a multiactive object request queue, a request is executed if it is compatible with requests that are already executing and with older requests in the queue. The first condition prevents data races and the second condition preserves the ordering of non-compatible requests. A side effect of the second condition is that starvation is avoided in the presence of a continuous stream of incompatible requests. Indeed, request execution would not be fair if a request A could overtake a request B even though incompatible, because then request A would prevent request B from executing, and this could last as long as request B is overtaken by incompatible requests. Besides, compatibility is a monotonic notion: once a request is marked as ready for execution (it has been checked for all compatibility conditions), this status cannot change until the request is executed. This default policy of execution maximises the parallelism but does not take into account the execution needs at the application-level. In Chapter 3, we extends the model in order to offer a wider range of possibilities.

2.4.3 MultiASP

MultiASP is the active object programming language that extends ASP (see Subsection 2.3.4) for the support of multiactive objects (see Subsection 2.4.1). MultiASP formalises the multiactive objects that are implemented in ProActive, and allows us to reason on multiactive object executions. A seminal version of MultiASP is given in [HHI13], and the encoding of MultiASP in the proof assistant Isabelle/HOL is publicly available [HK15]. We base this thesis on a slightly updated version of MultiASP, that we introduce in this section from a syntactical point of view, and for which we give the operational semantics in Chapter 4. We will also present in Chapter 4 our contribution to MultiASP that regards scheduling aspects. The preliminary formalisation of multiactive objects in [HHI13] is based on object instances whereas the version presented in this thesis is based on classes. This modification allows us to compare MultiASP against ABS in Chapter 4. MultiASP is an imperative programming language, whose syntax is inspired from object-oriented core languages resembling to Featherweight Java [IPW01]. It is worth noticing that the syntax of MultiASP is extremely close to what can be found in a ProActive program. This is done on purpose, as minimising the gap between the formalism and the practical implementation is, first, easier to transcribe, and second, ensures that what is proven trough formalisation still holds in a practical execution.

Figure 2.5 shows the static syntax of MultiASP. A program consists of a set of classes and one main method. Classes, methods, and statements are standard. In MultiASP syntax, x ranges over variable names, C ranges over class names, and m ranges over method names. We characterise a list of elements with the overlined notation. The list $\overline{x}$ denotes local variables when it appears in method bodies and denotes object fields when it appears in class declarations. In MultiASP, as in ProActive, there are two ways to create an object: `new` creates a new object in the current activity (a passive object), and `newActive` creates a new active object.

P	::=	$\overline{C}\ \{\overline{x}\ ;\ s\}$	program
S	::=	$\mathtt{m}(\overline{x})$	method signature
C	::=	$\mathtt{class}\ \mathtt{C}(\overline{x})\ \{\overline{x}\ \overline{M}\}$	class
M	::=	$S\{\overline{x}\ s\}$	method definition
s	::=	$\mathtt{skip}\ \mid\ x = z\ \mid\ \mathtt{return}\ e\ \mid\ s\ ;\ s$	statement
z	::=	$e\ \mid\ e.\mathtt{m}(\overline{e})\ \mid\ \mathtt{new}\ \mathtt{C}(\overline{e})\ \mid\ \mathtt{newActive}\ \mathtt{C}(\overline{e})$	expression with side effects
e	::=	$v\ \mid\ x\ \mid\ \mathtt{this}\ \mid\ \textit{arithmetic-bool-exp}$	expression
v	::=	$\mathtt{null}\ \mid\ \textit{primitive-val}$	value

Figure 2.5 – The class-based static syntax of MultiASP.

Also, no syntactic distinction exists between local and remote (asynchronous) invocations, $e.\mathtt{m}(\overline{e})$ is the generic method invocation. Similarly, as synchronisation on futures is transparent and handled through wait-by-necessity, there is no particular syntax for interacting with a future. A special variable, `this`, enables access to the current object. The sequence operator is associative, with a neutral skip element: a sequence of instructions is possibly rewritten $s; s'$, with s not a sequence. Although they are omitted in Figure 2.5, we assume that the *if* and *while* constructs exist, even if we did not define their semantics explicitly. In Chapter 4, we will present the complete semantics of MultiASP along with the contribution.

To recap, there are two main building blocks in this thesis: ProActive and MultiASP. The two of them lie under a common programming model that is multiactive objects. Figure 2.6 summarises the technology pile introduced in this section, including the historical background introduced in Subsection 2.3.4. This diagram highlights the temporal evolution and gives an idea of the increment for each block. The contributions of this thesis apply on the new stack displayed on the picture, namely, the multiactive object model, MultiASP, and ProActive.

2.5 Determinism in Active Object Executions

Since active objects are concurrent entities, their global execution, characterised as a sequence of communications, is non-deterministic. However, the design choices and the implementation details of active object frameworks affect the level of

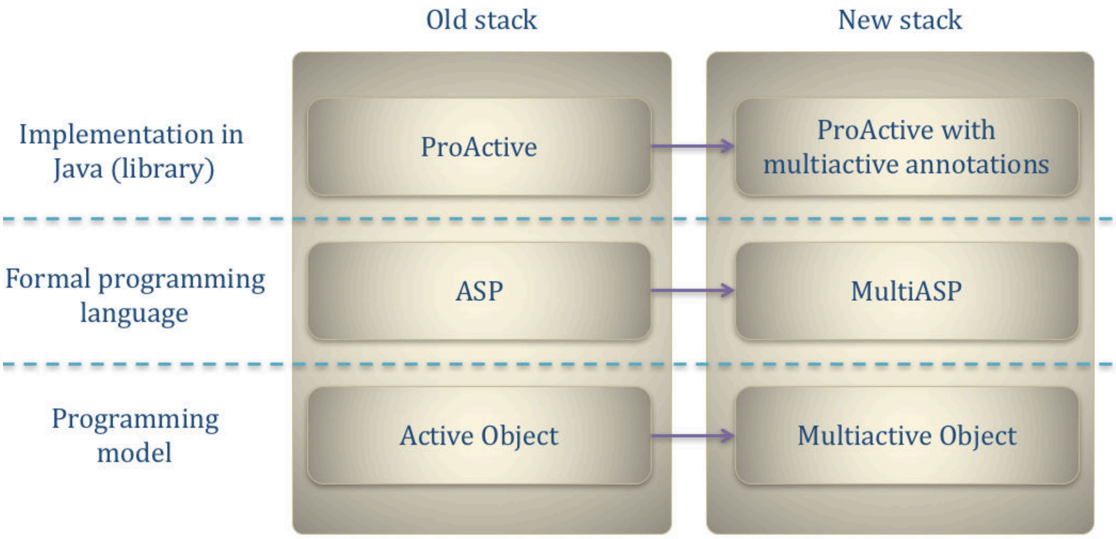

Old stack
New stack
Implementation in Java (library)
ProActive
ProActive with multiactive annotations
Formal programming language
ASP
MultiASP
Programming model
Active Object
Multiactive Object

Figure 2.6 – Interweaving of ASP, ProActive, multiactive objects, and MultiASP

determinism of active object executions. These choices and details make some frameworks more or less deterministic than other, either globally, i.e. at the level of the execution of the application, or locally, i.e. at the level of the execution of the active object. In general, being more deterministic implies having more properties on the execution. The programmer can use these properties to better predict the behaviour of the program. It is thus easier to program under these conditions: less interleavings to consider, and more sequential events. On the other hand, having less determinism enables more flexibility on the implementation side. It leaves room for choice and optimisation of the execution, for example choosing the most efficient scheduling with less constraints. Sometimes, having too strong constraints on the execution, and on the ordering of events, even makes the execution impossible (deadlock); this is clearly a counter objective of program determinism.

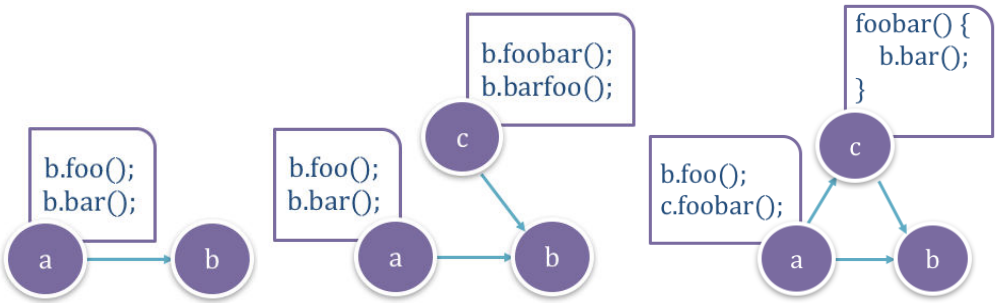
In the context of active objects and actors, a first variable source of determinism is that the communication between them can be implemented with various communication channels, that offer various guarantees. Although very specialised, this implementation detail is of great importance for the programmer, because it gives the set of possible executions that can be expected looking at asynchronous invocations. Three examples of asynchronous invocations are given in Figure 2.7.

They reflect the cases where the type of communication channel changes the execution guarantees. Throughout the survey of active object languages conducted in Section 2.3, we came across four different approaches of asynchronous communications between active objects, summarised below with the guarantees they offer.

No assumption on the order of reception of requests. In the original actor model, the only guarantee that actors give is that messages are put in the mailbox of actors in the same order as they are received. In [Agh86], this behaviour is denominated as a FIFO *arrival order*. This type of communication channel does not guarantee that two sequential invocations of the same active object are received in the same order. In this case, instantiated in Figure 2.7a, active object `b` has in its request queue either `foo` before `bar`, or `bar` before `foo`. This unpredicted behaviour is motivated by the fact that objects `a` and `b` might be separated by a network, which does not ensure ordering without a specific additional layer. For example, ABS and Creol make no supposition on the order of reception of requests. This semantics has the advantage of not excluding any underlying execution platform for such active object languages.

FIFO point-to-point. This communication channel ensures that all asynchronous invocations that are sequentially sent from one sender are ordered in the same sequence on the recipient side. If FIFO point-to-point ordering is used in the case of Figure 2.7a, then the execution of this example becomes deterministic: the only possibility is that active object `b` receives request `foo` before request `bar`. For example, Rebeca actor language offers this guarantee. However, when requests arrive from different senders to a same recipient, no assumption can be made. In Figure 2.7c, no one can tell if active object `b` will receive request `foo` (sent from `a`) before request `bar` (sent from `c`) only with a FIFO point-to-point ordering guarantee. The advantage is that the programmer can send consecutive requests that have a dependency between them.

Causally ordered. Communications that occur on causally ordered channels are correlated with the *happened-before* relation: the sequence of requests are



(a) No ordering vs. FIFO point-to-point.

(b) FIFO point-to-point vs. no interleaving.

(c) FIFO point-to-point vs. causal ordering.

Figure 2.7 – Examples of invocations affected by communication channels.

preserved within multiple hops. In Figure 2.7c, considering that active object a sends request foobar after request foo to active object b, all requests to b that are sent during the execution of foobar are received after request foo on b. Causal ordering of requests gives a stronger determinism in active object executions FIFO point-to-point channels. For example, ASP - and MultiASP - ensures causal ordering of requests. It has the same advantage as FIFO point-to-point, but the guarantees also operate when there are intermediates in the communication.

Interleaving-free sequences. This special kind of communication enables compound asynchronous invocations that are ensured to be put in the request queue of the recipient atomically: no other requests can be inserted in between the compound requests. In Figure 2.7b, if interleaving-freedom is applied to the invocation sequences of active objects a and c, then b can only receive these requests in two different orders: either foo and bar in the first place, or foobar and barfoo before. For example, SCOOP actor language features the specification of interleaving-free sections. Within an interleaving-free sequence, the communication is intrinsically FIFO point-to-point, but it gives more determinism with respect to concurrent invocations from other senders. The advantage for the programmer is that he does not have to consider the possibility of having other executed requests within a sequence; thus it is easier to program correctly.

It is worth noticing that the communication channel that is taken is distinguished from the scheduling policy that is applied, although both notions are used complementarily. Indeed, the communication channel impacts on the order of reception of requests whereas the scheduling policy impacts on the order of their execution. A scheduling policy is often fully deterministic, although some active object frameworks do not guarantee any execution order of requests, like Creol and ABS. Additionally, since these languages feature cooperative scheduling of requests, their global execution is highly non-deterministic. As we will see in Chapter 4, when an active object language is implemented for a concrete execution platform, it must define a scheduling policy that offers a minimum of guarantees. Advanced scheduling policies are also studied in more details in Chapter 3 in the context of ProActive, and also compared to related works.

Apart from communication channels and scheduling policies, another potential source of non-determinism is when the active object language allows a request to release the execution thread in the middle of its processing. This is the case of all active object languages that feature cooperative scheduling. Cooperative scheduling introduces an additional source of non-determinism because the request order obtained through the communication channel is mixed with the order in which awaiting requests are paused, typically consecutive to an `await` command, like in Creol, JCoBox, ABS, and Encore. Contrarily to communication channels, that impact on the global determinism of active object executions, the presence of language constructs like `await` weakens the local determinism of active objects, especially because, most of the time, the reactivation of an awaiting request depends on the progression of another request in a concurrent activity. In ASP and in AmbientTalk, since requests run to completion, the interleaving possibilities are totally discarded. Thus, ASP and AmbientTalk programs execute more deterministically than programs involving cooperative scheduling. Besides, when the active object model is based on object groups, like in ABS and JCoBox, requests that target different objects compete for the same execution thread. Thus, the scheduling decision must be considered in addition to the decision between starting and resuming requests, increasing even more the execution possibilities.

In conclusion, the multitude of aspects that impact the determinism of active object executions make the active object languages and frameworks difficult to compare to each other. The experimental evaluation of their performance is intricate, because they do not offer the same execution guarantees. The reason for this difficulty is that the sources of non-determinism are placed at different levels of the different languages. For example, MultiASP places the non-determinism at the level of the activity, by allowing parallel requests. Because of parallelism, one could think that MultiASP is less deterministic than active objects based on cooperative scheduling. However, MultiASP requests always run to completion, whereas an execution thread of cooperative scheduling interleaves different requests, so it is not possible to conclude directly whether cooperative scheduling provides more or less determinism than multi-threaded scheduling. Nevertheless, MultiASP ensures a causal ordering of requests, which is the communication channel that gives the most determinism. In general, the communication channel that is employed gives the restrictions of the protocols that built upon an active object language. In Chapter 5, we will see that having causally ordered communications is a basic condition for having a correct fault tolerance protocol for MultiASP active objects. Indeed, it makes programs fully deterministic, as it was formally proven for ASP in [CHS04], under some conditions. Overall, the determinism in active object executions is a important aspect to consider when active object languages are developed and when active object frameworks are compared to each other. Also, these aspects are crucial when active object languages are translated for specific execution platforms; this is a problematic that we explore in Chapter 4.

2.6 Positioning of this Thesis

There is now an undeniable effervescence that surrounds asynchrony in programming languages. This fact has led to the emergence of many programming models and languages, and also to soft evolutions in well-established programming languages. The active object programming model and associated programming models are becoming leaders in the asynchronous computing era, thanks to their provided safety and convenience of programming. However, we are still facing a gap between two categories of active object-based programming languages. The programming languages that are developed in academics feature the latest programming abstraction whereas the programming languages that are used in the industry, and that are put into production, still cope with modest increments that are inspired from developments made in academics.

All the active object-based programming languages presented in Section 2.3 have the common point of being designed for a particular purpose, and excel in it. This is because they have approached different areas. For example, ABS is a powerful modelling language that focuses on property analysis and verification, and offers hooks for code generation. Other languages, like Encore and ProActive, target another objective, that is providing an efficient runtime for active objects. But there too, two different directions are taken: Encore targets multicore platforms whereas ProActive reaches its potential in distributed settings. Likewise, distributed execution is the basis of AmbientTalk programs, but the runtime of AmbientTalk is focused on connectivity in mobile ad hoc networks whereas the runtime of ProActive is optimised for HPC, which leads to dissimilar programming models and implementations.

Among the presented languages, some of them only have a limited support for distributed execution (JCoBox), do not take distribution into account (Creol), or not yet (Encore). Other languages make distributed execution their main focus, but they are not yet fully implemented (some backends for ABS) or are not ready for production code (AmbientTalk). A new trend is to study new programming paradigms in a layered manner. Basically, the language for specifying and verifying the program is decoupled from the language of execution, which enables using the language that is the most efficient in the considered layer. Such a design implies having, in the middle, an automatic layer that translates the specification language into the execution language. Yet, the two end points must map, semantically. ABS has several backends but only two of them consider distributed execution. The Haskell backend for ABS [BB16] provides a cloud aware translation of ABS models. However, the execution language is not multi-platforms and, as the implementation is based on Haskell continuations, it is not supported on industrial execution platforms like the JVM. The application domain of the Haskell backend for ABS is thus relatively focused. The second distributed backend for ABS, targeting the JVM, and based on Java 8 features [Ser+16], is starting to be investigated and currently experiencing implementation challenges. In this thesis, we also provide a fully working and proven backend for ABS that runs on the JVM, but with a novel approach: by encoding ABS in another active object language. We believe that the layered approach for designing new programming languages is the most beneficial when it relies on layers that are proven and experienced.

The programming languages developed in academics give a set of advanced language constructs and abstractions that are likely to become the basis for all programmers. However, at the time of Big Data, distributed execution is a 'must have' for any language platform. In order to be accepted in the industry, the programming languages must ensure a good performance in the first place, and also be compatible with the industry platforms. But often, programming languages are developed from scratch in order to investigate new paradigms, and the transposition to mainstream languages or platforms is then quite tough, because the constraints of the execution platforms and execution environments have not been taken into account. Moreover, industrial use cases and data are rarely made available for research, necessarily leading to biased implementations. During the HAT project[3], an industrial use case from Fredhopper, a e-Commerce company[4], has been successfully modelled in ABS [WDS12]. As mentioned in the paper, this partnership has highly influenced further decisions made for the design of ABS. This is why, not only the design but also the implementation challenges have to be tackled on the research side. Indeed, integrating new language features and runtimes might entail severe design issues. Technological transfer from research to industry is not a straightforward engineering work: it has crucial issues and is not ensured to be successful, nor to stick to the original objective.

To some extent, the couple formed by ASP and ProActive, is representative of this mixed approach. ASP provides a provable formalism on which to confront new language constructs and abstractions. On the other hand, ProActive enables checking that the new developments make sense in practical settings, and show their applicability in general purpose computing. Moreover, both of them evolve in a coordinated manner, which reduces the gap from theory to practice at each iteration. As the fundamental model of ASP was becoming inadequate for multicore computing, a need for updating both the formal model and the library was

[3] EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu)

[4] www.fredhopper.com

inescapably rising. The presented, up-to-date model MultiASP started to give up mono-threaded active objects in favour of controlled multi-threading, as well as its seminal implementation in ProActive. Because the multiactive object model was, to the best of our knowledge, the first active object model supporting multi-threaded execution, it raised new challenges. Among them, cleverly adapting the local execution is crucial to make the best use of parallelism. Multi-threaded execution in ProActive also challenges the previously established non functional features that make its strength, in particular, robustness of execution in distributed settings, that are known to be extremely patchy. Clearly, a need for thorough support of controlled multi-threading in multiactive objects appeared, both in the formal model and in its implementation. These are the questions we tackle in this thesis. Yet, multiactive objects did not only bring new problems, they also brought an opportunity for a higher expressiveness. This is another aspect we explore in this thesis, by challenging multiactive objects against the other models and this way, probe their capacity.

Overall, we position this thesis as a complete contribution to multi-threaded, asynchronous and safe programming models, although we mainly focus our work on MultiASP for formalisation and on ProActive for implementation. This divided approach is similar to the ones of ABS and Rebeca (Subsection 2.3.7), with their use of backends: proving properties on a high-level, neat language, and relying on massively used platforms for implementation burdens. Overall, throughout the contributions of this thesis, we take a particular care in making interacting all approaches. We believe that this work can impact both academic and industrial fields. We have achieved the formalisation and implementation of a thorough model and language that cater for a high expressiveness, and we are confident in its success because it is experienced, proven, robust, and also resourceful for future work.

Chapter 3

Request Scheduling for Multiactive Objects

Contents

In this chapter, we present the first contribution of this thesis, which can be summarised as an application-level scheduling for multiactive objects (see Subsection 2.4.1), and its implementation in the ProActive library (see Subsection 2.4.2). After motivating the need for application-level scheduling in the first section, we present an approach for controlling the creation and allocation of threads of multiactive objects in the second section. Then, in a third section, we present a priority specification mechanism that allows the processing of requests in prioritised order. We will see that the way priorities are applied do not jeopardise the safety of multiactive object execution. The imbrication of the components in the multiactive object scheduler of ProActive is summarised in a fourth section. The compositional software architecture of the scheduler proves to have a well defined separation of concerns and enables adaptation of scheduling components without affecting the core of the active object library. Finally, in the last section, we test the priority specification mechanism on several use cases and we experiment different internal representation of priorities. This chapter is associated to the publication [3].

3.1 Motivation

The multiactive object programming model automatically provides a request-level parallelism thanks to the specifications written by the programmer. Bringing parallelism in active objects also brings the question "how parallel requests should be scheduled for execution?". In particular, an order of execution has to be chosen. Also, a maximum degree of parallelism must be defined, otherwise all the benefits of parallelisation could be lost with too many parallel threads. Indeed, in multiactive objects, even if requests are compatible (see definition in Section 2.4.1), we cannot infinitely create new threads on-the-fly, whenever some parallelism can be done, and expect a good execution performance. One has to size the program for the resources that will be needed for execution. Another thing that must be taken into account is the applicative requirements. Often, the different requests sent to an active object have different importance regarding the application business. This fact must be reflected in the execution order of requests. A default multiactive

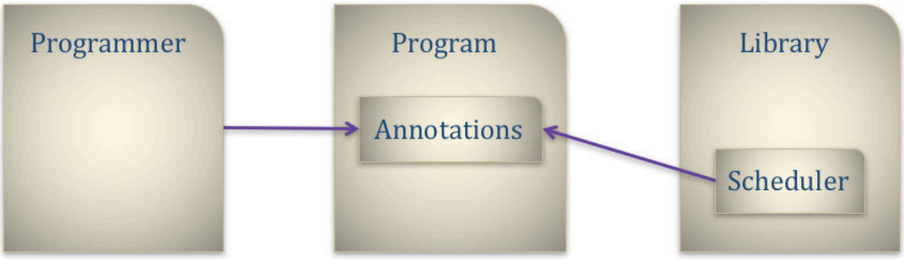


Figure 3.1 – Indirect interaction between the programmer and the multiactive object scheduler through the annotation interface

object execution policy, implemented in ProActive, ensures fairness and maximum parallelism (see Section 2.4.2), but cannot fit the aforementioned aims because request execution is not controllable by the programmer through the program. In this first contribution, we empower the programmer with scheduling customisation of request execution for multiactive objects. We extend the ProActive library and propose new and updated multiactive object annotations for the programmer that is interested in the performance of a multiactive object-based application. Using this new set of annotations does not require more expertise on the programmer's side: the specification is purely declarative, and represent an interface between the programmer and the request scheduler, as shown in Figure 3.1.

Although multiactive scheduling annotations enable high performance tuning, the programmer does not have a direct interaction with the scheduler. Thanks to this interface, the programmer can focus on what the tuning should do, and not on how to do it. This is why we claim that it is a scheduling made at application-level. However, empowering the programmer also means increasing the probability of introducing bugs. In order to prevent unwanted behaviours, for example if the programmer writes inconsistent annotations, we implement the interpreter of multiactive object annotations with some guardrails.

Through this work, we aim at a complete programming model for high performance computing that is suited to all programmers. Programmers that know the benefits (but that ignore the challenges) of parallelism, concurrency, and scheduling can easily and safely use this programming model. They are supposed to use a small part of our tool set and to rely mainly on default behaviours. The range ends with the most experimented programmers that, contrarily to beginners, know the challenges and are willing to rely on a high-level framework in order to precisely tune their applications while avoiding mistakes. For now, existing active object and actor frameworks used in major development projects are either very restrictive, and in this case, they are often mixed with other concurrent paradigms (which can end up badly), or too permissive and in this case, they can lead to incorrect usage. This phenomenon is well studied in [TDJ13]. Providing request scheduling controls to the multiactive object framework is also a way to attract the programmers with a better expertise in concurrency, by making the framework complete and reasonably usable. Besides, as our annotation mechanism is incremental, we offer a smooth transition between ease of usage and performance.

3.2 Threading Policy

Multiactive objects are based on several threads of execution. In order to allow the programmer to have more control on the creation of threads and on their allocation for requests, we propose a new set of annotations that can be divided into two parts. We implement them in the ProActive library. On one hand, we define threading constructs that apply at multiactive object grain. On the other hand, we define threading constructs that apply at the grain of multiactive groups. Both kinds of threading constructs must be declared on top of a class of a multiactive object. We detail the two kinds of constructs in the two next subsections. When code snippets are shown, they show the multiactive object annotations as they can be used in ProActive.

3.2.1 Thread Limit per Multiactive Object

The first kind of threading construct developed for the multiactive object programming model consists in defining, through a class annotation, the configuration of threads that applies for all the multiactive objects that are instances of this class. In practice, we introduce the `@DefineThreadConfig` class annotation to be able to specify three characteristics related to thread management:

– The maximum number of threads handled by a multiactive object. We call this maximum number of threads the *thread limit* of the multiactive object.

On one hand, this parameter prevents creating too many threads and having a thread explosion for a multiactive object. On the other hand, it allows, internally, to initialise an adapted thread pool at multiactive object startup, which avoids the latency of creating threads on-the-fly.

- The kind of thread limit. In our model, we say that a thread limit can be of two kinds: either *soft* of *hard*. A *soft thread limit* counts, in the maximum number of threads, only the threads that are currently active. In other words, it counts only the threads that are not waiting for a future to be resolved, i.e. which are not in the wait-by-necessity state. On the contrary, a *hard thread limit* counts, in the maximum number of threads, all the created threads, even the ones that are in wait-by-necessity. Consequently, with a soft thread limit, there can exist more threads than the specified maximum number of threads, which cannot happen with a hard thread limit. The rationale for using a soft thread limit is that then, a multiactive object cannot deadlock because the thread limit is too small.

- Finally, a reentrance parameter enables reusing a thread in wait-by-necessity for the processing of a request that can unblock the wait-by-necessity. To this end, we systematically trace chains of requests such that we know if a request is an ancestor of another one. Thus, we stack the processing of a new request on a waiting thread if we are sure that the request that is blocked cannot be unblocked without the processing of this new request. This is a way to drastically optimise the number of threads that are used by the application. This is an advanced parameter.

```
@DefineThreadConfig(threadPoolSize=5, hardLimit=true, hostReentrant=false)
public class MyClass {
  ...
}
```

Listing 3.1 – An example of thread configuration annotation.

Listing 3.1 gives an example of usage of the `@DefineThreadConfig` annotation. A multiactive object created with this configuration strictly has five threads at maximum to process its requests all along its life span. Since the configuration specifies a hard thread limit, no additional thread will ever be created to compensate a thread that is blocked in wait-by-necessity. Moreover, in this example reentrant requests are not allowed to be stacked on the same thread (`hostReentrant=false`). Such a configuration has more chances of having a deadlock at runtime, due to the impossibility of compensating the threads that are blocked in wait-by-necessity. In this situation, unless the use case is well studied, it might be better to release one of the two specified constraints with either `hardLimit=false` to be able to use other threads when a thread is in wait-by-necessity or `hostReentrant=true` to enable the re-utilisation of a thread in wait-by-necessity by a request that can unblock this status.

As all multiactive object annotations, the `@DefineThreadConfig` annotation is optional. Moreover, inside this annotation, parameters are also optional. If they are not defined, default values are used: the maximum number of threads is limited to highest possible integer[1]. By default, a multiactive object executes in a soft thread limit, and does not enable reentrant calls. These threading features were already primarily implemented in the core of the ProActive library. However, they were hard coded in the library with minor dynamic adaptation, such as automatically setting the maximum number threads to the number of cores of the running machine, and necessitated to change the source code of the library in order to take new values into account. This required the recompilation of the library each time a value was set. My role was to design the programming interface of the library that would allow the programmer to configure those aspects from the application, and to make these values apply at runtime, by making the request scheduler take them into account. Naturally, we chose the annotation mechanism as programming interface in order to offer a good flexibility and to be as consistent as possible with the existing multiactive object programming model.

```
public boolean switchHardLimit(boolean hardLimit) {
  boolean formerLimit = this.limitTotalThreads;
  this.limitTotalThreads = hardLimit;
  return formerLimit;
```

[1] in Java, this number is pointed to by `Integer.MAX_VALUE` and is encoded with 32 bits.

```
}
```

Listing 3.2 – Scheduler method to change thread limit at runtime.

In addition to the configuration of threads through annotations, we proposed an API to dynamically switch from soft thread limit to hard thread limit, and conversely. The rationale for this alternative is that, sometimes, scheduling decisions need to be made at runtime, depending on current resource utilisation, or depending on events that are external to the application. In practice, the programmer can call a method of the scheduler of a multiactive object that changes the kind of its thread limit. Listing 3.2 shows the public method added to the scheduler that enables dynamic switching of thread limit kind. This method returns the thread limit kind that was applied before the change, which enables even more dynamic decisions.

3.2.2 Thread Limit per Group

Limiting the number of threads per multiactive object is rather coarse grain and does not enable specifying which requests are processed on these available threads. Indeed, the default policy to allocate the threads does not enforce a special meaning at the application level. A request that is ready to execute is allocated the next available thread, regardless of how many requests of the same type already execute. But a programmer would like to balance the execution of requests according to their business. Consider for example a weakly consistent database, where write requests come regularly in a large batch. Then, one would like to balance reading requests and write requests, in order to limit the impact of write batches on the reading latency. In addition to the global thread limitation that applies per multiactive object, we propose a mechanism to set a limit on the number of threads that are used at the same time by the requests of the same group. Typically, this option allows the programmer to do two things. First, it allows the programmer to reserve some threads for the processing of the requests of a given group. Second, it allows the programmer to specify the maximum number of threads that can be used by the requests of a given group at the same time. Those two options can be combined to create scheduling behaviours that favour or curb some kind of requests, and enables treating request in a qualitative way. Since such options applies to groups of requests, we extend the `@Group` annotation with two optional parameters that correspond to the two situations explained above.

```
@DefineGroups({
  @Group(name="group1", selfCompatible="false", minThreads=2),
  @Group(name="group2", selfCompatible="true", minThreads=1, maxThreads=2)
})
```

Listing 3.3 – An example of thread limit per group.

Listing 3.3 shows an example with two groups defined: `group1` and `group2`. `group1` specifies only the number of threads that are reserved for the processing of its requests. `group2` specifies one one hand that one thread is reserved for its requests and, on the other hand, that its requests cannot take more than two threads at the same time. When the two of those options are specified, there exists a fixed lower and upper bounds on the number of threads that are taken by the mentioned group.

Naturally, for a given group, the number of reserved threads must be lower or equal to the maximum number of threads specified for this group. As the thread limit options are purely declarative, they can be easily used by a wide range of programmers. But also, as we perform no static verification on annotations, there can subsist inconsistencies in the programmer's specifications. For example, a wrong partitioning of threads between groups could lead to starvation of requests, or even to deadlocks. Our primary aspiration is to trust the programmer's annotation, but our objective is also to ensure a certain safety of execution. This is why we define a number of policies that override the programmer's specifications in the case we detect inconsistencies in the annotations at runtime. Here are the policies we define for thread limits:

– If the number of reserved threads is higher than the maximum number of threads for a given group, then the number of reserved threads of this group is lowered to the maximum number of threads specified (both parameters have the same value).

– If the sum of reserved threads for all groups is higher than the size of the global thread pool, then we increase the size of the global thread pool to the result of this sum.

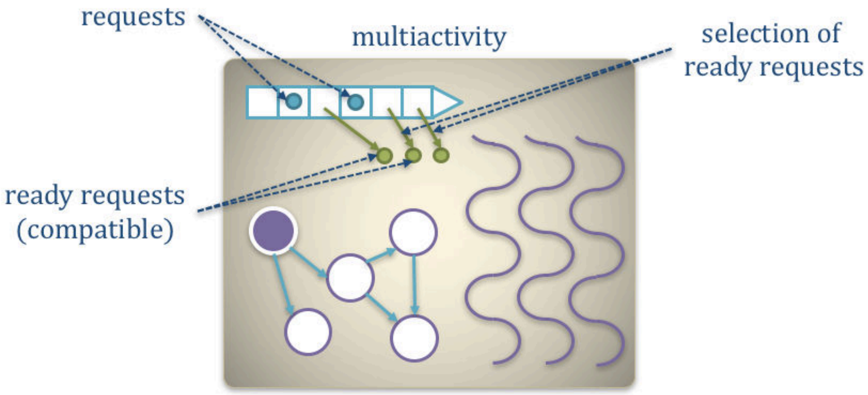


Figure 3.2 – The ready queue of a multiactive object.

- In those cases, a warning is displayed in the console in order to let the programmer know about those automatic modifications.

Such policies constitute a precious guardrail for inexperienced programmers. They prevent the programmer from making obvious mistakes and help him debugging them.

3.3 Request Priority

3.3.1 Principle

In a multiactive object configuration where the number of threads is limited, requests compete for thread resources. More precisely, the requests that compete for execution are the ones that are, at a given moment, compatible with the requests that are executing and compatible with the requests that were received before. We qualify such requests as *ready-to-execute* requests, filtered in the reception queue thanks to the compatibility predicate. Due to the limited number of threads, there might exist requests ready for execution, but that cannot be executed because all threads of the multiactive object are busy. Ready requests form a set that is a subset of the requests that are present the reception queue. We call this subset the *ready queue* of a multiactive object. The ready queue is, like the reception queue, ordered according to the order of reception. It is represented on Figure 3.2.

A step further in the qualitative processing of requests is to be able to reorder requests that wait in the ready queue, according to their importance. Indeed, it is possible that a programmer would like some requests to be executed before some others if possible, that is, express a *priority* relationship between the requests. We define the priority relationship in our context as the fact to reorder the requests in the ready queue such that requests that have a high priority overtake requests that have a low priority. Note that overtaking in this case does not contradict the compatibility relationship, since the requests that are in the ready queue are compatible altogether. In this section, we present a mechanism that allows the programmer to specify a prioritisation of requests for request execution. This priority mechanism applies on the ready request queue only, because applying priority of requests on the reception queue would be unpredictable since it is not sure that all requests of the reception queue can be executed. Obviously, the main objective of priorities is to reduce the response time of the most important requests. The criterion of importance can be established for various reasons and is always determined by the programmer.

In the following, we introduce the programming model that allows the programmer to specify request priority in the ProActive library in Subsection 3.3.2. In the ProActive runtime implementation, we maintain a priority graph to represent request priorities. We introduce the properties of this priority graph in Subsection 3.3.3 and we discuss its implementation in Subsection 3.3.4.

3.3.2 Programming Model

To implement the priority specification mechanism on top of multiactive objects, we must first answer the question "how does the programmer specify request priorities?". We will then answer the question "how are request priorities internally represented?" in Subsection 3.3.4. In order to allow the programmer to define request priorities, we extend the set of multiactive object annotations. In order to remain consistent with the existing programming model, we apply priorities on groups of requests (see Subsection 2.4.2). Consequently, all requests of the same group have the same priority. This is not a significant restriction: a group can still be split into several groups in order to assign them different priorities. We choose a graph-based data structure to represent the dependencies between priorities. A classical approach for prioritised execution would have been to represent a priority with an integer. However, this approach entangles the programmer with the internal representation of priorities, and forces him to take into account all the values previously given before deciding on a new priority value. This argument gets worse considering the fact that, in the ProActive implementation of multiactive objects, annotations are inherited according to class inheritance. Moreover, the graph-based representation can express a partial order, which is interesting in our case because we do not want to force the programmer to define a priority for all the methods of the class.

Multiactive object annotations that deal with group priority are led by the `@DefinePriorities` annotation, that gathers several partial priority declarations. `@DefinePriorities` can contain several `@PriorityHierarchy` annotations. A `@PriorityHierarchy` annotation creates an ordering on several groups of requests. The order in which the groups are declared defines the priority dependencies, like `group1 > group2 > group3`. Several groups can belong to the same priority level. A `@PrioritySet` annotation is used for this purpose. Overall, the priority definition uses three nested priority annotations.

```
@DefineGroups({
  @Group(name="atomic", selfCompatible=false),
  @Group(name="concurrentRead", selfCompatible=true, parameter="Key",
                                                        condition="!equals"),
  @Group(name="concurrentWrite", selfCompatible=true, parameter="Key",
                                                        condition="!equals"),
  @Group(name="monitoring", selfCompatible=true)
})
@DefineRules({
  @Compatible({"atomic", "concurrentRead", "concurrentWrite"},
                                                        condition="!isLocal"),
  @Compatible({"concurrentRead", "concurrentWrite", "monitoring"}),
})
@DefinePriorities({
  @PriorityHierarchy({
    @PrioritySet(groupNames = {"concurrentWrite"}),
    @PrioritySet(groupNames = {"concurrentRead"}),
    @PrioritySet(groupNames = {"monitoring"})
  }),
  @PriorityHierarchy{
    @PrioritySet(groupNames = {"atomic"}),
    @PrioritySet(groupNames = {"monitoring"})
  })
})
public class Peer {

  @MemberOf("atomic")
  public JoinResponse join(Peer p) { ... }

  @MemberOf("concurrentWrite")
  public void add(Key k, Value v) { ... }

  @MemberOf("concurrentRead")
  public Value lookup(Key k) { ... }

  @MemberOf("monitoring")
  public void monitor() { ... }

}
```

Listing 3.4 – Peer class with priority annotations.

Listing 3.4 shows an example of priority annotations, applied on the `Peer` class that was firstly introduced in Subsection 2.4.2[2]. The priorities defined in this example are pictured on Figure 3.3. In this example, we have defined that `add` requests have a higher priority than `lookup` requests, through the priority specification of their respective groups. Also, the priority annotations specify that both of these groups have a higher priority than the `monitoring` group. As such, `monitor` requests will have a lower priority compared to `add` and `lookup` requests. Aside, another priority graph is defined through a second `@PriorityHierarchy` annotation, that creates a priority dependency between the `atomic` and `monitoring` groups. But since no priority relationship is defined between the `atomic` group and the `concurrentWrite` and `concurrentRead` groups, their competing requests will behave in a FIFO manner. This priority model is flexible: it is easy to add new

[2] Compared to the version of Subsection 2.4.2, separated multiactive object groups have been created for holding `add` and `lookup` methods.

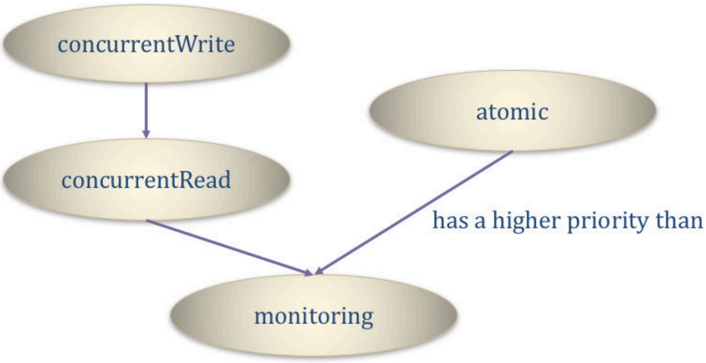


Figure 3.3 – Dependency graph for Listing 3.4

priorities, independently from the priorities that were previously defined. Most of the time, there are several ways to define the same dependency graph with different priority annotations.

In spite of their convenience and their added value in multiactive object-based applications, priorities can raise scheduling issue in case of thread-limited execution (Section 3.2). Considering a limited number of threads and a prioritised execution of requests, low priority requests can suffer of *starvation*. Starvation is a situation where access to a resource is infinitely delayed such that the time to access it cannot be bounded. In our context, requests compete for threads and requests have priorities. A starvation situation occurs if there is a continuous stream of high priority requests, preventing low priority requests from being executed. The second scheduling issue that can arise with priorities is known as *priority inversion*. When all the available threads are already allocated, if a request with highest priority arrives, then this request must wait whereas there might be lower priority requests that are executing. This situation, where low priority requests block the execution of high priority requests, cannot be avoided without request preemption. In order to counterbalance starvation and priority inversion issues, the thread bounding mechanism per group introduced in Subsection 3.2.2 must be used. The mechanism for reserving threads per group can be used to avoid starvation whereas the mechanism for limiting threads per group can be used to avoid priority inversion. In practice, using a sufficient thread limit per multiactive object (Subsection 3.2.1) is often a simple key to execution fairness, but it is not adapted to very specific use cases where scheduling is a crucial point of the application.

3.3.3 Properties

This subsection presents a formalisation of the priority specification mechanism explained above. To recap, the priority of a ready-to-execute request is checked against the priority of the other ready-to-execute requests, if any, in order to determine the position of the request in the ready queue. We design the scheduling process such that the scheduling time and the memory footprint are minimised. The main question that must be answered is whether a ready-to-execute request overtakes another one. In this section, we present the properties of the scheduling process, that is based on the characteristics of the priority graph. Beforehand, we introduce some notations. A request is denoted R, a group G and the ready queue Q. The priority relationship is denoted with the relation $\longrightarrow$, whose operands are nodes of the graph. It relates two groups of requests, and is obtained through the priority definition of the programmer. In the priority graph, reachability is denoted $\longrightarrow^+$. For example, $G_1 \longrightarrow^+ G_2$ means that there exists a directed path from node G_1 to node G_2. In extension, group G_1 has a higher priority than group G_2. $\longrightarrow^+$ is also defined as the *transitive closure* of $\longrightarrow$. Below are the characteristics of the priority graph maintained in multiactive objects.

Graph construction. By construction, the priority graph is cycle-free. Since it is possible, with a wrong usage of priority annotation, to create a circular dependency, we guard the graph construction against this possibility, by preventing any dependency that introduces a cycle. Thus, we enforce a directed acyclic graph. More precisely, when a priority annotation is processed, we do not add the dependency in the priority structure if it introduces a cycle. We output an error message for feedback, but the execution proceeds. Also, we ensure that each multiactive object group has a unique corresponding node in the graph. As such, multiple occurrences of the same group name in the priority annotations always point to the same priority node in the graph.

Possibility to overtake. A request from group G_1 can overtake a request from group G_2 if and only if $G_1 \longrightarrow^+ G_2$, which is, if there exists a directed path from G_1 to G_2. A request from group G_1 has no priority relationship with a request from group G_2 if $(\neg\, G_1 \longrightarrow^+ G_2 \;\wedge\; \neg\, G_2 \longrightarrow^+ G_1)$, which is, 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 $G_1 \mathbin{//} G_2$. Consequently, a request from group G_1 cannot overtake a request from group G_2 if either $G_1 \mathbin{//} G_2$ or $G_2 \longrightarrow^+ G_1$. Note that, for any group G, $G \mathbin{//} G$, which means that requests with the same priority have no priority relationship.

Insertion in ready queue. When a request is marked as ready-to-execute, we check the possibility to overtake the other requests in the ready queue, in order to determine the position of the request in it. More precisely, if we consider a request R belonging to group G, and a ready queue made of $R_1, R_2, ..., R_n$ respectively belonging to groups $G_1, G_2, ..., G_n$, then R is inserted just before the R_i with the smallest i, such that $G \longrightarrow^+ G_i$, or at the end of the queue if no such R_i exists. In other words, a request from group G must be inserted just before the first request that can be overtaken, starting from the request that is first in the ready queue (i.e. that has the highest priority and that is the oldest having this priority).

The previously defined characteristics allow us to define the insertion process of a request in the ready queue.

Definition 1. *[Insertion process] Suppose* $group(R) = G$, $Q = [R_1, ..., R_n]$, *and* $\forall i \in 1..n$, $group(R_i) = G_i$, *then:*

if $\forall i, G_i \longrightarrow^+ G \lor G \mathbin{//} G_i$, *then* $\quad insert(R,Q) = [R_1, ...R_n, R]$

else let $j = min(i \mid G \longrightarrow^+ G_i)$ *in* $\quad insert(R,Q) = [R_1, ..., R_{j-1}, R, R_j, ..., R_n]$

Definition 1 guarantees that the ready queue is always ordered according to the characteristics of the priority graph, that defines the overtaking conditions. This leads us to the ordering property of the ready queue.

Property 1. *[Ready queue ordering] The ready queue is always ordered such that, if* $i \leq j$, *then* $G_i \longrightarrow^+ G_j$ *or* $G_i \mathbin{//} G_j$.

We prove this property below, by checking that the property is maintained both when a request is inserted following Definition 1, and when the first request is removed from the ready queue.

Proof. By recurrence on the length of Q. Property 1 is trivially verified for an empty request queue ($length(Q) = 0$). For the inductive case, assume that we have a request from group G to insert in the ready queue that is made of $R_1, R_2, ..., R_n$ and that verifies Property 1. The requests $R_1, R_2, ..., R_n$ respectively belong to groups $G_1, G_2, ..., G_n$. We consider R from group G_R to be inserted in the queue. Let $R'_1, R'_2, ..., R'_{n+1}$ be the new queue after insertion. Let us look at the two cases of the insertion process of Definition 1.

1. Suppose that we did not find $j \le n$ such that $j = min(i|G_R \longrightarrow^+ G_i)$. Then, by recurrence hypothesis, the beginning of the queue $R'_1, R'_2, ..., R'_n$ is ordered and verifies Property 1. Additionally, $\forall i \le n$, R'_i is before R in the new queue, and because we did not find j, we have $\neg(G_R \longrightarrow^+ G_i)$. Thus, $G_i \longrightarrow^+ G_R$ or $G_i//\ G_R$.

2. Suppose now that we found $j \le n$ such that $j = min(i|G_R \longrightarrow^+ G_i)$. Then we have $R_1, ..., R_{j-1}, R, R_j, ..., R_n$. Consider $i \le k$, we have to prove $G'_i \longrightarrow^+ G_R$ or $G'_i//G_R$. If i and $k \ne j$, then we conclude by recurrence hypothesis. Otherwise, there are three cases to consider. We prove them by contradiction.

($i \le j = k$) Since $i \le j$, we cannot have $G_R \longrightarrow^+ G'_i$ because this is contradictory with the definition of j, which stipulates that j is the minimum i that satisfies $G_R \longrightarrow^+ G'_i$. Consequently, $G'_i \longrightarrow^+ G_R$ or $G'_i//G_R$.

($i = j < k$) Suppose that we have $G'_k \longrightarrow^+ G_R$. By definition of j, we have also $G_R \longrightarrow^+ G'_{j+1}$, so we have $G'_k \longrightarrow^+ G_R \longrightarrow^+ G'_{j+1}$. By transitivity we have $G'_k \longrightarrow^+ G'_{j+1}$ and $j+1 \le k$, which is contradictory to the recurrence hypothesis, that is that $R_1, R_2, ..., R_n$ verifies Property 1. So, there cannot exist a $k > j$ such that $G'_k \longrightarrow^+ G_R$.

($i = j = k$) In this case, we have $G_R//G_R$, and this verifies Property 1.

All the cases converge towards the fact that the queue is always ordered if the insertion process of Definition 1 is applied. □

In summary, we have designed and implemented in ProActive a priority model that the programmer can express through multiactive object annotations on top of a class. The multiactive object runtime, in our case ProActive, extracts from the priority graph the overtaking possibilities. Thank to this knowledge, it can systematically reorder the ready queue upon the event of a new request arrival. Thus, we can give the guarantee that the first requests of the ready queue (the ones that are executed first) are the ones that have the highest priority.

3.3.4 Implementation and Optimisation

According to the scheduling process defined in Subsection 3.3.3, in the worst case the priority graph must be entirely searched for each request that is inserted in the ready queue. Although using a graph for representing priority is expressive, it might lead to a performance issue if the implementation of the graph exploration is not efficient. This fact is mainly carried by the internal representation of the graph. Some related works use integers as internal representation of priorities. We started with a graph-based approach, that is less efficient than the integer approach. This is why, in our implementation, we refine the internal representation of the graph in order to have a performance similar to the integer-based representation. For that, we rely on the transitive closure of a directed graph. The transitive closure of the priority graph of Figure 3.3 is the same graph with an additional edge from node `concurrentRead` to node `monitoring`. In our case, this basically means that, if group A has higher priority than group B, and group B has a higher priority than group C, then we know that group A has a higher priority than group C. We can compute this knowledge for all pairs of groups. We can then store this knowledge such that, afterwards, answering the question whether a request can overtake another one takes a constant time. Indeed, the transitive closure of a graph can be seen as a binary matrix that stores the overtaking possibilities between all pairs of groups. In such a matrix M, of size $N \times N$ where N is the number of groups, we store a positive value in $M[G_1][G_2]$ if there exists an edge from group G_1 to group G_2 in the transitive closure of the priority graph. This computation happens just once at application launching time, when the multiactive object annotations of the class are processed. Then, this knowledge is used the whole execution time: each time the priority of a pair of groups has to be considered, we just pick the priority value associated to this pair in the matrix. The drawback of the matrix structure, compared to the graph, is the memory usage. Indeed, there are exactly N^2 entries in the matrix against only N nodes for the graph plus $N \times (N + 1)/2$ edges at maximum. In practice, the number of methods of an active objects that can be remotely invoked (and thus, prone to priority ordering) is usually rather small. However, if a performance problem still exists, whether in computing time or in memory, the knowledge that is extracted from the priority graph can still be stored using highly efficient probabilistic data structure, like the bloom filter [Blo70], in order to store the overtaking possibilities.

Overall, the priority model is primarily based on a graph structure with solid properties. The graph structure is adapted to represent priorities because it is more expressive than the representations based on a total order: it does not force to apply priorities on all requests. The programming model enables a piecewise definition of the priority graph, which makes it scalable. The implementation of the priority runtime is also scalable, because overtaking possibilities can be cached in a flat structure in order to preserve execution throughput, at the expanse of some latency at application start. Section 3.5 measures the performance of the priority runtime of ProActive. Finally, the priorities defined by the programmer are not dynamic, which is in part why we need to use thread limits per group to prevent specific scheduling problems, like starvation. Dynamic priorities could be considered in future work in order to adapt them to runtime events.

3.4 Software Architecture

In order to adapt the execution of multiactive object requests according to the new scheduling constructs (related to thread management and priorities), we have modified the ProActive runtime, and more specifically the multiactive object request scheduler. We focus this section on the implementation of the multiactive object request scheduler of the ProActive library, more specifically on annotation processing and on request execution. We detail the integration of the scheduling mechanisms introduced in this chapter with the software architecture of ProActive. We also mention the data structures that are used to implement the different components of the request scheduler of ProActive.

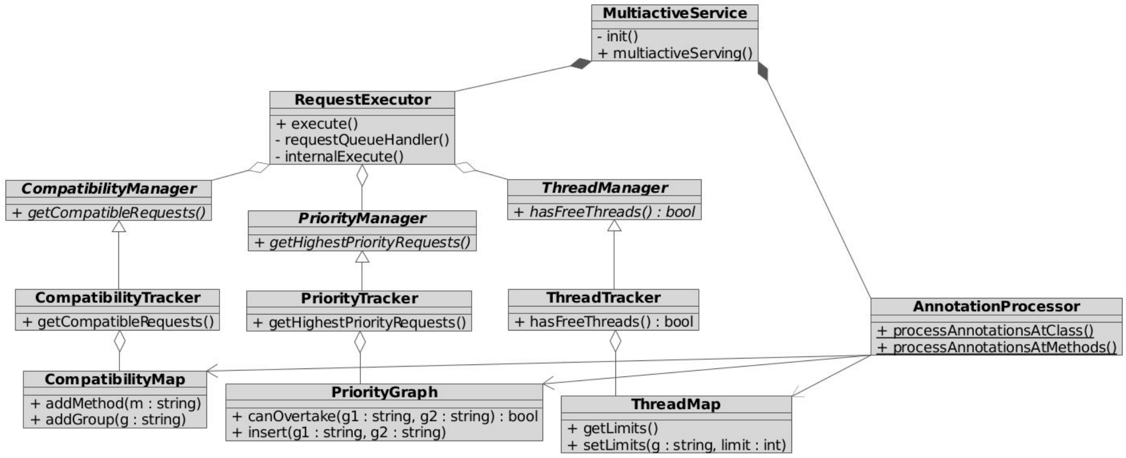

MultiactiveService
- init()
+ multiactiveServing()
RequestExecutor
+ execute()
- requestQueueHandler()
- internalExecute()
CompatibilityManager
+ getCompatibleRequests()
PriorityManager
+ getHighestPriorityRequests()
ThreadManager
+ hasFreeThreads() : bool
CompatibilityTracker
+ getCompatibleRequests()
PriorityTracker
+ getHighestPriorityRequests()
ThreadTracker
+ hasFreeThreads() : bool
AnnotationProcessor
+ processAnnotationsAtClass()
+ processAnnotationsAtMethods()
CompatibilityMap
+ addMethod(m : string)
+ addGroup(g : string)
PriorityGraph
+ canOvertake(g1 : string, g2 : string) : bool
+ insert(g1 : string, g2 : string)
ThreadMap
+ getLimits()
+ setLimits(g : string, limit : int)

Figure 3.4 – Class diagram of multiactive objects with scheduling controls

Figure 3.4 displays the compositional class diagram of the multiactive object request scheduler that is implemented in ProActive. This diagram highlights the most important interfaces that are available and that connect the different classes. A multiactive object is associated to a multiactive service, in order to activate multiactive execution. The `MultiactiveService` class is composed of two entities: an annotation processor and a request executor. Firstly, the `AnnotationProcessor` class aims at processing the annotations that are written by the programmer in a Java class. The annotation processor first initialises all the data structures needed for an multiactive object execution. More precisely, it creates and fills the compatibility mapping between groups, builds the priority graph, and stores the various thread limits that will be checked all along the lifetime of a multiactive object. Typically, this phase is where the optimisations for a further fast execution is made, like for example the caching of request priorities to enable fast access (see Subsection 3.3.4). Consequently, this phase incurs a certain latency when a multiactive object is started. The overhead of multiactive object execution is evaluated in [HHI13; HHI12] for the compatibility part and in Section 3.5 for the priority part.

A multiactive service is also composed of a request executor, that encapsulates the request scheduling behaviour. The `RequestExecutor` class is a black box that outputs the next request to execute according to compatibilities, priorities and available threads. To this end, the request executor must keep track of what is going on in the multiactive object, e.g. which requests are executing and with which compatibilities, which requests are in the ready queue and with which priority, and which usage of threads is made. The request executor constantly relates such dynamic information with the static information that was defined by the programmer, and that is interpreted by the annotation processor.

The multiactive object request scheduler of ProActive defines a clear separation of concerns between the classes that store static information, and the dynamic situation where all the metrics are tracked and checked at a given moment. The classes that store static information appear at the leaves of the diagram of Figure 3.4, and are named representatively of the data structure they use. The classes that hold dynamic metrics, i.e. the multiactive object situation at given time, are named with the `Tracker` suffix. The classes that are named with the `Manager` suffix compare the static structures against the tracked metrics, and offer the interfaces to interact with this knowledge. In general, the request executor accesses the interfaces of the compatibility manager, of the priority manager and of the thread manager. In this thesis, we have introduced all the classes that relate to the priority stack and to the threading stack. The priority mechanism relies on a customised priority queue that adheres to the *producer/consumer* design pattern. A dedicated thread registers compatible requests to the priority queue while another thread polls the queue to retrieve the requests that have the highest priority. The synchronisation of the two threads is implemented using the native concurrency mechanisms of Java. The order in which the request executor proceeds is as follows. It first filters requests in the reception queue thanks to the `getCompatibleRequests` predicate of the `CompatibilityManager` class, giving back the requests that are ready to be executed. Among ready requests, a second filter is applied using the `getHighestPriorityRequests` predicate of the `PriorityManager` class. This method returns the list of requests that have the highest priority. Then, for these requests, each of them is evaluated in FIFO order such that the ones that satisfy the `hasEnoughThreads` predicate of the `ThreadManager` class are immediately executed with a thread of the multiactive object. The requests that fail one of the steps of compatibility, priority, and thread availability, will be checked again in the next scheduling round in order to get executed.

The software architecture applied in ProActive for the implementation of the multiactive object runtime can be adapted to any other implementations of active objects, provided that the language features meta-data (annotations) and features reflection for meta-data processing.

3.5 Evaluation

In this section, we evaluate the performance of the implementation of multiactive object priorities in ProActive through micro-experiments, that is, focusing on the internal scheduling of one multiactive object. Regarding the implementation details of Section 3.4, a specific point that could be a bottleneck in the scheduling of requests, is when the position of a request in the ready queue is computed. In this sense, the priority mechanism must have a low overhead on the scheduling time in order to be beneficial. Firstly, we check that our implementation of the priority mechanism is effective: requests that have a high priority have a high throughput compared to the requests with a lower priority. Then, we evaluate the insertion time of requests in a setting where the ready queue grows, and we conclude on the scalability of the priority model.

3.5.1 Experimental Environment and Setup

In all experiments, we execute an application, written with the ProActive library, on a single machine. Indeed, although ProActive is a distributed library, we focus here on the evaluation of local aspects. The machine used for the experiments has four processors Intel Core Q6600 that have four cores each, and has 8GB of memory. The experiments are made with a JDK in version 7.

We have developed the applications on the EventCloud platform [Pel14] that relies on ProActive for implementing a distributed peer-to-peer storage and publish/subscribe system. The main Java class of the applications we develop for the experiments is the `PeerImpl` class, that represents a peer in the peer-to-peer distributed system. We annotate this class with multiactive object annotations. The methods of this class that are experimented have a small body definition, with just a few logging instructions. In a realistic multiactive object setting, requests that are sent to a multiactive object should do enough computation to balance the communication overhead. However, in our case small methods allow us to compare the lowest execution time with the full overhead of priorities. In all experiments, all groups are declared compatible in order for the ready queue to be as large as the reception queue. In addition, before starting executing requests, we block the multiactive object execution on purpose until all requests of the experiment are registered in order to evaluate the insertion time of requests according to the length of the queue. For all the previous reasons, the following experiments present a worst-case scenario in which the priority mechanism is tested.

3.5.2 High Priority Request Speed Up

The first scenario aims at showing that the priority mechanism is effective. We consider the `PeerImpl` class annotated with two groups of requests, that we name group A and group B. These groups hold a single method that is respectively named A and B. After instantiating a multiactive object of this class, we send to it 1000 requests, alternating A and B requests. The class is configured with a strict thread pool of four threads, and we specify no thread limit for any group. This scenario is executed in two different priority settings: first with no priority (setting #1), and second with group A having a higher priority than group B (setting #2). We measure the execution time of a request according to its position in the initial request set. We average the measured execution time for each 50 consecutive requests of the same group (either A or B), forming an evaluation based on batches of requests. For example, the first point of a curve corresponds to the averaged execution time for the 50 first requests of a given group (A or B). This way we can see how much B requests have to wait compared to A requests, depending on their position in the queue. More precisely, we measure the execution time of requests from the time when the request queue is filled with all requests and the time when the service of a request ends. Therefore, the execution time is composed of: the time to insert the request in the ready queue (insertion time), the time spent in the ready queue (waiting time), and the time to serve the request (service time).

Figure 3.5 displays the results of the two experimented settings for this scenario.

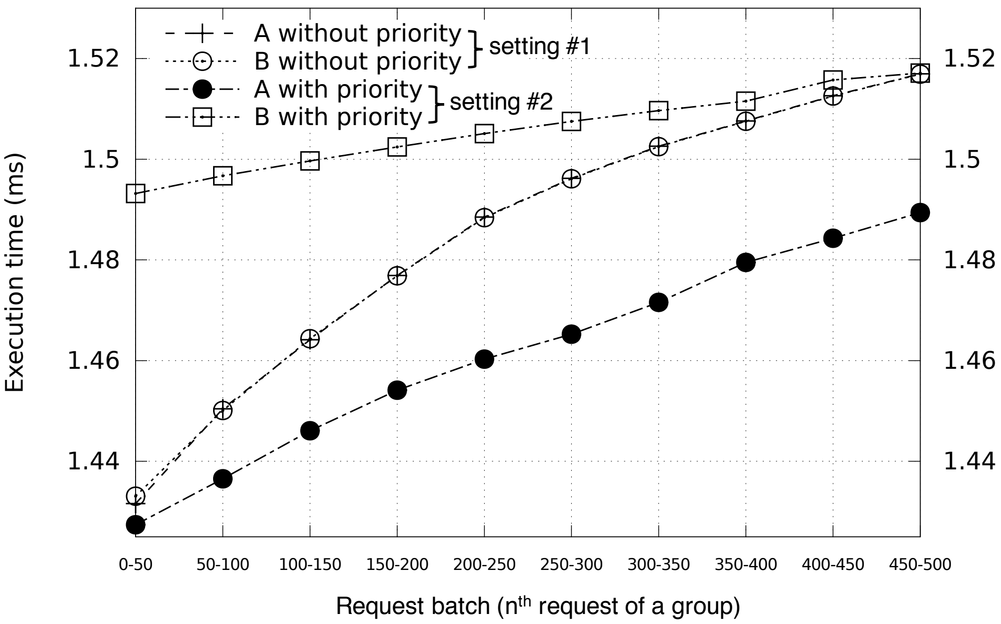


Figure 3.5 – Execution time of requests (per batch) with and without priority.

When not using priorities (setting #1), the execution time of requests increases equally for the requests from group A and the requests from group B. This is because the requests are executed in the same order as they are received, which is alternatively A and B requests. Consequently, the time to insert a request in the ready queue is the same for both kinds of requests, so is the time to wait in the ready queue. A and B requests give exactly the same increasing global execution time when no priority is applied.

When using priorities (setting #2), the execution time of A requests has a dynamic that is different from the execution time of B requests. Since the communication time and the service time of all requests are necessarily equal, it is the insertion time and the waiting time that are dominating the dynamic of the curves. First, we clearly see that the curve of high priority A requests (black dotted line) is always kept below the curve of setting #1 without priority (crossed line). This shows that the priority mechanism succeeds in increasing the throughput of high priority requests. Secondly, the curve of low priority B requests (squared line) is always above the one without priority (circled line). We notice that the execution time of B requests is much longer than the one of A requests for the first batch of requests (0-50). This is due to the fact that the first B requests have to wait for the execution of all A requests before being executed. Then, the execution time of B requests increases slowly, because once A requests are treated, the insertion time of B requests is reduced. Note also that the more requests there are in the queue, the longer is a scheduling round (to choose the next request to execute when a thread is freed), because of the manipulation of larger data structures.

In conclusion, the priority mechanism reduces the waiting time of high priority requests, which is more visible when the queue is long. Reducing the waiting time contributes to the speed up of the global execution of high priority requests.

3.5.3 Overhead of Prioritised Execution

We now experiment the priority mechanism of multiactive objects in two different scenarios that evaluate the *overhead* of having priorities. In this context, the overhead is the time needed to insert a request in the ready queue according to its priority, from the moment the request is ready to be executed (i.e. is compatible). In particular, when a request is inserted in the ready queue, the queue is locked until the request is put at the right position, in order to prevent data races between pushing and pulling threads. So, keeping the insertion time as small as possible reduces the possibility of having a bottleneck at this point. As a comparison, we can consider the time needed to insert a request when no priority applies: the request is appended in constant time to the tail of the ready queue. However, with priorities, we might look through the whole ready queue in order to find the right position of a request. Consequently, this operation depends on the number of requests that are in the ready queue. In addition, the structures used to store and retrieve request priorities play a big part in the complexity of the insertion process.

In this subsection, we compare the graph-based priority mechanism associated to multiactive objects with a representation of priorities based on integers, that we have developed specially as a point of comparison. Integer-based priorities are less expressive because they force priorities to fit in a totally ordered set, but the integer-based representation allows us to have a baseline for comparison with the lowest memory footprint and the lowest access time. In the two following

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

Listing 3.5 – Graph-based.

```
@DefineIntegerPriorities({
  @Priority(groupNames={"G1"}, val=5),
  @Priority(groupNames={"G2"}, val=4),
  @Priority(groupNames={"G3"}, val=3),
  @Priority(groupNames={"G4"}, val=2),
  @Priority(groupNames={"G5"}, val=1)
})
```

Listing 3.6 – Integer-based.

Figure 3.6 – Priority annotations for Alternate scenario.

experiments, we set a multiactive object thread pool of one thread, in order to focus the study on priorities only, and we do not set any other thread limit.

Alternate High and Low Priority Requests

This scenario aims at evaluating the overhead of the priority mechanisms in a simple configuration. Like in the previous scenario, we annotate the `PeerImpl` class, but with new groups and new priorities. We define five groups of requests from $G1$ to $G5$ with multiactive object annotations, and each of them holds one method of the same name. We declare a linear priority dependency between the groups: $G1$ has a higher priority than $G2$, $G2$ has a higher priority than $G3$, etc. In the first setting, we declare the priorities with the integer-based mechanism and in the second one we declare them with the graph-based priority specification. Both priority definitions are displayed on Figure 3.6. These priority definitions are semantically equivalent, although they use different annotations and rely on different structures behind the scene.

The application consists in sending 1000 requests to the multiactive object. Like in the previous application, we send alternatively one request of the highest priority and one request of the lowest priority, that is alternatively from group $G1$ and from group $G5$. No request from the other groups is sent, other groups are simply meant to have a bigger priority structure. We execute this application in the two settings corresponding to the two kinds of priority annotations. For each setting, we measure the average insertion time for each batch of 50 consecutive

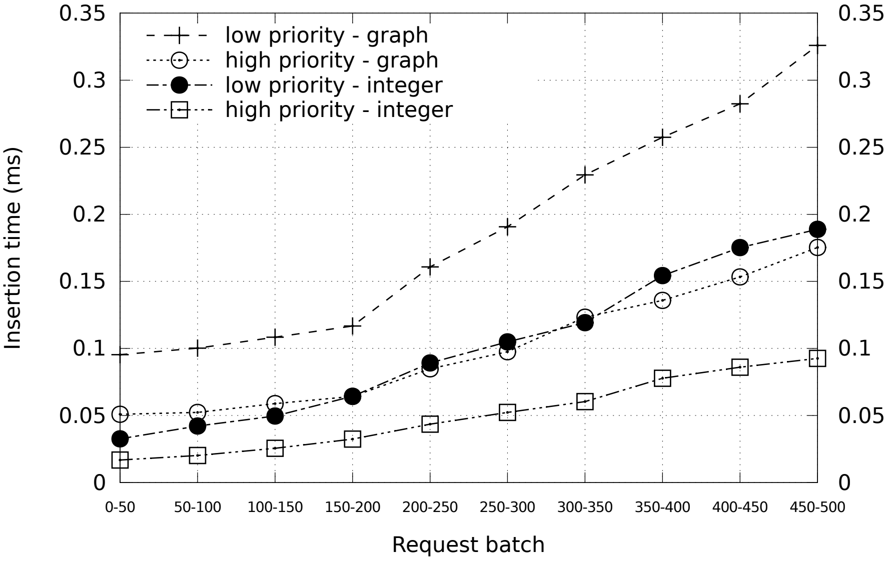


Figure 3.7 – Insertion time of low and high priority requests (per batch).

requests of the same group, like in the previous application.

Figure 3.7 shows the results of the graph-based and integer-based priority mechanisms, for the two groups of requests. For both evaluated mechanisms, the insertion time of high priority requests is twice faster than the insertion time of low priority requests, because high-priority requests have to be checked against half less requests than low-priority requests. However, the time to insert a request using the priority graph takes almost twice more time than when using the priorities based on integers. This is observable for both high and low priority requests. Thus, using a priority graph is twice slower in this configuration, because the graph must be searched each time a request is inserted in the ready queue. For this application, we also measured the service time of a request. As we measure quasi-empty request bodies, the result represents the minimum time required to execute a request. We measured an average request execution time of 1.4 ms. Knowing this, the average insertion time of a request in the batch [200-250] represents only only 8% of the service time. In other words, the insertion time is always below 10% of the minimal service time when the ready queue contains less than 200 requests, which is a reasonable restriction. The cost of using a priority graph can be thus balanced with this rational knowledge.

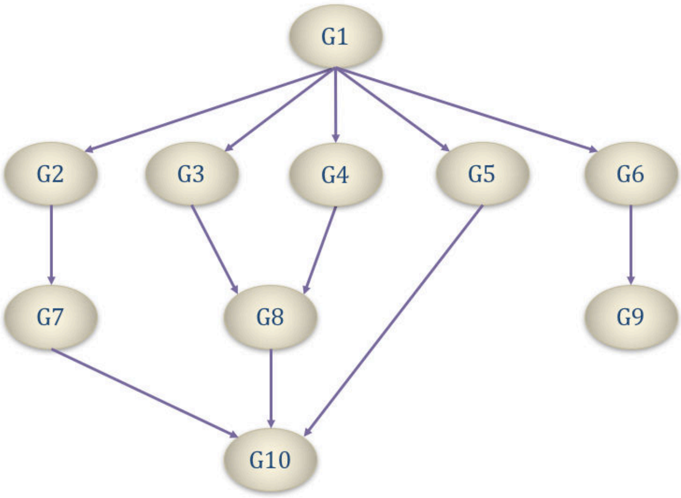


Figure 3.8 – Dependency graph of scenario with sequential requests.

Sequential Requests from Not Linearisable Graph

This scenario illustrates a more realistic case than the previous one, in which the priority dependencies are not linear, and in which all groups of requests participate in the application. We define ten groups of requests, from $G1$ to $G10$, each of them holding a single method, respectively named from $g1$ to $g10$. The priority dependencies between them are expanded in Figure 3.8. This graph can be directly translated into multiactive object priority annotations. The corresponding priority graph definition is displayed on Figure 3.9. Declaring the same priority relationship with an integer-based priority mechanism is not possible. In order to compare the two priority mechanism anyway, we encode the priority graph as one of its possible linear extensions for defining it with the integer-based priority mechanism: $G1$ has value 4, $G2$ to $G6$ have value 3, $G7$ to $G9$ have value 2, and $G10$ has value 1, the highest value being the one that has the highest priority.

In the application, we sequentially send a request of each group in a predefined order (with no particular meaning), that is: $g7$–$g1$–$g2$–$g9$–$g4$–$g10$–$g8$–$g3$–$g6$–$g5$. Using a predefined sequence of requests allows us to have deterministic results while using the whole graph. We send 500 requests per group, so in total, 5000 requests are sent to the multiactive object. Unlike the previous scenario, we do not look specifically into the overhead of the priority graph just for a given priority, but

```
@DefineGraphBasedPriorities({
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G2"}),
    @Set(groupNames = {"G7"}),
    @Set(groupNames = {"G10"})
  }),
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames ={"G3","G4"}),
    @Set(groupNames = {"G8"}),
    @Set(groupNames = {"G10"})
  }),
  ...
```

Listing 3.7 – Part 1.

```
  ...
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G5"}),
    @Set(groupNames = {"G10"})
  }),
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G6"}),
    @Set(groupNames = {"G9"})
  })
})
```

Listing 3.8 – Part 2.

Figure 3.9 – Graph-based priority annotations for Sequential scenario.

for all levels of priorities. We use again the batch granularity for our results and compute the average insertion time each 50 consecutive requests sent from the same group. After, we average these results among all groups for each batch of requests.

Figure 3.10 shows the global average insertion time of a request depending on the request batch, for three priority mechanisms. The overhead of the graph-based priority mechanism on the insertion time is more than twice higher than the overhead of the integer-based priority mechanism, which is bigger than in the previous scenario. This is explained by the fact that, in this scenario, the graph is more complex in terms of nodes and edges. The difference between the graph-based and the integer-based priority mechanisms grows bigger as the number of requests in the queue increases. In this sense, at some point the performance of the application could suffer from the graph-based representation of priorities. To address this problem, we use the optimisation of the internal representation of graph-based priorities presented in Subsection 3.3.4. In this case, the transitive closure of the priority graph is built and then, the knowledge that is extracted from it is stored in a binary matrix of size 10 × 10. The results of the execution of the application with this optimisation can be seen on Figure 3.10. In this case, using

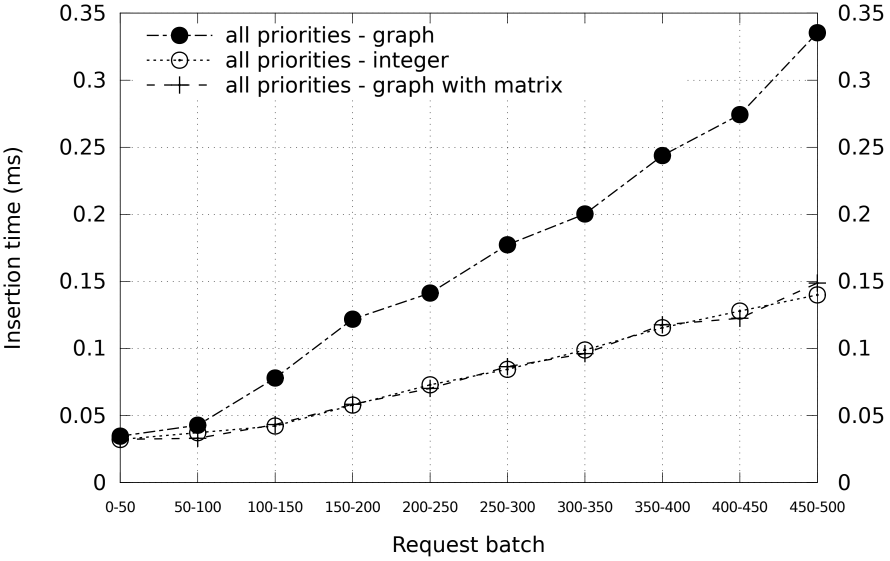


Figure 3.10 – Insertion time of sequential requests - Not linearisable priority graph.

the matrix is as efficient, in terms of insertion time, as using the integer-based priority mechanism. Additionally, this optimised internal representation preserves the ease of usage at the programming-level, and preserves the high expressiveness of the priority graph.

3.6 Conclusion

In this chapter, we have presented advanced constructs that allow a programmer to customise the request scheduling of multiactive objects. We have implemented the presented constructs in the ProActive library, that offers now a thorough implementation of the multiactive object programming model. From the programmer point of view, these mechanisms take the form of new multiactive object annotations that can be optionally used and that predictably influence multiactive object execution. In a first place, we have defined new constructs related to the management of multiple threads inside a multiactive objects. The introduced constructs allow the programmer to control the parallelism and to allocate the threads in a better way. Unsafe specifications are overriden by the multiactive object runtime in order to prevent mistakes and to make the application execution come through.

Robustness of multiactive object execution is deepened on an aspect that is different from wrong configurations in Chapter 5. In a second place, we have enlarged multiactive object annotations with request priorities. For that, the priority definition and internal representation are based on a priority graph. We have given the properties of the priority graph, as well as how it is efficiently implemented in ProActive, enforcing a strict separation of concerns. Finally, we have experimented multiactive object priorities in various settings that show that they are useful and efficient. Throughout this work, we have completed the multiactive object framework to give all the means to the experienced programmer to optimise step by step its multiactive object-based applications. In summary, the obtained programming model can be characterised as:

User-friendly. The threading specification and the priority relationship between requests are easy to define for the programmer and also easy to maintain and extend, even with a lot of multiactive object groups.

Expressive. The threading specification and the priority representation allow the programmer to express a large variety of scheduling patterns, and allow him to do as much as what could be possible with a low-level mechanism.

Consistent. Like in the existing multiactive object programming model, the extended features relating to request scheduling are based on meta-programming, and ensure a backward compatibility.

Effective. The new specifications have a deterministic effect on the scheduling of requests, and this effect can be rationally expected from the programmer.

Efficient. The new features introduce a minimal additional cost to the execution of multiactive objects, more precisely when applying the scheduling policy.

The mentioned characteristics demonstrate the completeness of the multiactive object programming model, and of its implementation in ProActive. Even with all customisation possibilities of multiactive objects, involving compatibilities, priorities and thread management, the scheduling of requests is still deterministic thanks to a consistent framework. In addition, despite the fact that the implementation of the model was only described in the case of ProActive, the whole multiactive object programming model is not tight to a particular language. It can be implemented in any object-oriented programming language that supports meta programming, and an existing active object-based language can be straightforwardly extended for that. In conclusion, the contributions presented in this chapter enlarge the multiactive object programming model, so that it can benefit from a very precise scheduling while remaining high-level and safe.

* * *

Other works have been conducted on concurrency models or on scheduling policies, that facilitate access to parallelism or that raise scheduling concerns at the application-level. JAC (Java with Annotated Concurrency) [HL06] is an extension of Java that adds concurrency constructs. JAC offers a set of annotations that allow the programmer to specify whether an annotated entity must be guarded by a lock, or if it can safely be executed concurrently by several threads. JAC offers a simplified view of concurrency for the programmers that are not experts in this domain. Unlike multiactive object annotations, JAC annotations are processed before compilation. Low-level thread synchronisation is added where needed in the program thanks to the analysis of JAC annotations. However, this prevents dynamic decisions made at runtime, as it is possible with multiactive objects. In addition, JAC offers a `@schedule` annotation, placed on top of a method, that allows the programmer to define a boolean method for deciding whether the method can safely be executed when it is invoked. In this boolean method, the list of methods awaiting to be executed is fully accessible, which makes it possible to define fine-grain scheduling policies. This way to impact on the scheduling of requests requires some expertise with queue manipulation but as it is programmatic, it is more expressive than a declarative approach like ours or like the one of Creol(see below).

Although ABS (see Subsection 2.3.3) has by default no special policy to execute ready requests, ABS offers user-defined schedulers [Bjø+13] that can override the non deterministic default policy. The programmer can define several customised schedulers, and then associate them to object groups. In practice, user-defined schedulers are methods that are directly written in ABS. Such methods can be used through the `scheduler` annotation, by simply mentioning the scheduler name. Like in JAC, a user-defined scheduler selects a request to execute among a list of waiting requests. A lot of information can be retrieved from a request, such as arrival date, waiting time, request name, and request flags, so very precise scheduling policies can be expressed. Again, this way of customising request scheduling in active objects can be pretty complex for an inexperienced programmer. Compared to multiactive object scheduling, ABS schedulers are less guarded: in our approach, the interpretation of annotations and their automatic translation into scheduling constraints enable a filtering of mistakes. But on the other hand, ABS schedulers form a more powerful system in term of expressiveness.

Creol (see Subsection 2.3.1) also features an application-level scheduling mechanism. In [Nob+12], the authors extend Creol with request priorities. Each request has either a user-defined priority or a default priority, materialised by an integer. In Creol, the lowest integer specifies the highest priority. In order to enable a prioritised execution, an object interface must define a range of possible priority values. Afterwards, when a method of an object is called, the caller may assign a priority value in the defined range. Additionally, as the 'client side' might always ask for the highest priority, the 'server side' can also set a priority value for particular method calls. Then, to decide on the final priority of a given request, the object applies a deterministic function that takes into account both the priority of the caller and the priority given when the method is defined.

In conclusion, Creol has a programming model for priorities that is sophisticated but that is more difficult to reason about, as it is defined by the composition of different priorities from different sources. On the other hand, both JAC and ABS schedulers provide a fine-grained selection of requests, because the programmer directly manipulates the queue. However, one might argue that these solutions are low-level and force the programmer to write a significant part of the scheduling code, which is error-prone. On the contrary, multiactive object scheduling constructs are based on specification rather than programming. Consequently, using the scheduling constructs of multiactive objects is easy and higher-level, but they feature less expressiveness. The presented constructs hide the internal components of an active object behind the annotations, which is not the case when the programmer has a direct access to the request queue.

Chapter 4

From Modelling to Deployment of Active Objects

Contents

This chapter globally makes an in-depth study of active objects languages, by simulating different paradigms of active object languages with the paradigms of multiactive objects. Concretely, we present an approach for translating active object languages that are based on cooperative scheduling into a precise multi-threaded execution of active objects. We instantiate our approach by translating

ABS into ProActive, and this way realising a distributed execution backend for the ABS modelling language. To this end, we use the request scheduling features of multiactive objects that have been presented in Chapter 3. After giving a motivation for this work in a first section, the high-level approach and implementation details are given in a second section. An experimental evaluation of the ProActive backend for ABS is also presented there. A third section formalises and proves the correctness of the translation using the ABS semantics and using the calculus associated to ProActive, namely MultiASP. We present in the same section the modifications/extensions that we have brought into MultiASP in order to integrate the scheduling features of Chapter 3. Finally, the last section provides an informative feedback on this work. This chapter is associated to the publication [4].

4.1 Motivation and Challenges

There are multiple reasons for providing ABS with a distributed execution through ProActive. We have seen in Chapter 2 that the reason why there are many different implementations of the active object programming model is because each of them fits precise needs, from reasoning about programs to efficient execution in practical settings. The active object languages that target application deployment in real-world systems comply to constraints that are related to already existing execution platforms and programming languages. Such active object languages are mostly used by programmers interested in the performance of the application. ProActive typically fits in this category. AmbientTalk and Encore also base their execution on mainstream platforms. Other active object languages mostly target verification and proof of programs, but have not been originally designed for distributed execution, like typically ABS, Creol, and Rebeca. In general, ABS is massively used and developed by academics, and is also less constrained by existing execution platforms.

Overall, in this chapter we present a generic approach for executing all forms of cooperative active objects in distributed settings, relying on the experience of ProActive in this field. Additionally, the proof of correctness ensures that the verifications performed on the source code are still valid when the program is executed. For demonstration of our approach, we implement it specifically with

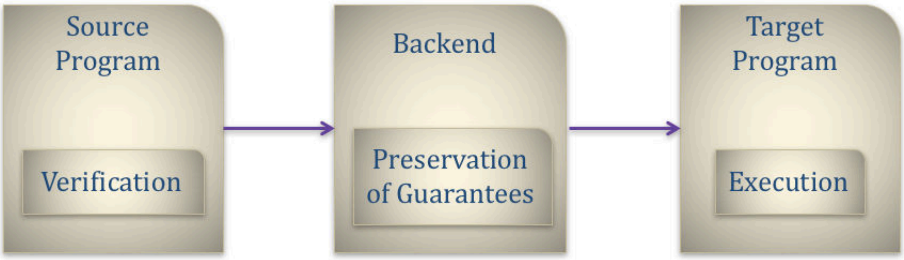


Figure 4.1 – From program development to program execution with backends.

ProActive and ABS, and present the ProActive backend for ABS. ProActive is representative of practical concerns because it provides an asynchronous and distributed programming model that is transparent to the programmer, making it more accessible to non-experts. Secondly, ProActive is a Java library, and can be executed on any standard JVM, making it runnable quite everywhere. Also, the underlying object structure of ProActive, encapsulating objects in an activity, makes the programming of application less intuitive but more scalable with the number of activities and objects, thus directly usable practically to tackle industrial use cases. ABS requires more expertise from the programmer because of the existence of asynchronous invocations, futures, and synchronisation points but this is compensated by the convenience of its object group model, that only considers one type of objects. Finally, some tools recently developed for ABS target cloud settings, that are intrinsically distributed. Thus, providing ABS programs with a distributed execution has a real significance. Using an existing, reliable platform to do this is probably the most appropriate approach. While fully implemented and effective, our practical translation of ABS into ProActive is certainly not restricted to this single scope. Indeed, ProActive and ABS characteristics encompass most of the paradigms that can be found in active object languages, and we discuss the adaptability of our work to the other active object languages.

Approaching language adaptation through a backend provides flexibility: a larger range of specific needs can be covered, separately at the modelling level and at the execution level. The workflow from an application's design to the application's execution is displayed on Figure 4.1. Backend-based application development allows active object paradigms, classified in Section 2.2, to be mixed at will, in order to eliminate unwanted constraints at a given level. This work advocates the rise active object languages based on multiple paradigms. However, backend-based application development also bring new challenges. The first practical concern boils down to answering the question “how can we embody a language construct that does not exist in the targeted language?”. The second concern is summarised in the question “how can we prove that this incarnation is correct?”. We answer those two questions throughout this chapter. We will see that answering them actually compels us to analyse the languages, giving an invaluable knowledge about active object languages and their implementations.

Overall, our contribution through this work benefits to two domains: formal methods and high performance distributed computing. First, we allow experts in formal methods, used to design programming languages and verification tools, to deploy, with minimum effort, distributed and efficient applications. Second, the work presented in this chapter should convince experts in distributed and high performance computing that formally verified programs can also be run efficiently. Having this guarantee should promote formal methods more largely in the industry.

4.2 Systematic Deployment of Cooperative Active Objects

In this section, we present the ProActive backend for ABS in an example-driven way. Basically, this section shows how the formal translation that is defined in Section 4.3 is instantiated in practice. The implementation of the backend is based on the Java backend for ABS[1], that produces Java programs that comply to the ABS semantics. Such Java programs solely execute locally, on a single machine. The ProActive backend relies on the part of the Java backend that translates the functional layer of the language. The translation of the object layer and of the concurrency layer is modified in order to produce ProActive code instead of local Java code. Finally, other adaptations are also made on the ABS compiler to enable distributed execution.

[1] Available at `http://abs-models.org/`.

4.2.1 A ProActive Backend for ABS

Object Addressing

In order to translate ABS into ProActive, a first challenge is to handle the difference of active object model. According to the classification of active object languages given in Subsection 2.2.1, ABS follows an object group model whereas ProActive follows a non uniform object model. As such, we must define an approach to embody object groups into active and passive objects, and in general to define what happens when a new object is created. In ABS, all objects can be referenced from any COG. In a distributed setting, local references cannot be used to reference objects that are not in the same memory space. In order to access objects across the network, a referencing system must be used. In ProActive, the objects that can be referenced globally are active objects, because active objects can be referenced beyond the local memory space through proxies and remote method invocations (see Subsection 2.3.4). On the contrary, passive objects can only be referenced locally.

For the implementation of the ProActive backend for ABS, translating each ABS object into a ProActive active object in order to make them remotely accessible is not a viable solution. Indeed, in the ProActive translation, all objects would have to be registered in the global referencing system. Also, all objects would have to unnecessarily carry a thread and a request queue associated to them. Moreover, this solution would require a complex synchronisation of threads, in order to comply with the semantics of the initial ABS program. The approach we choose for representing ABS objects in ProActive is scalable both in the number of objects and in the number of threads[2]. We put several translated ABS objects under the control of one ProActive active object. In other words, all translated ABS objects are passive objects in ProActive. This approach represents well the object group model of ABS. Indeed, the ProActive active objects that gather the translated ABS objects play the role of the ABS COG. However, for a distributed execution, the translated ABS objects must be referenceable from remote memory spaces. To this end, in the ProActive translation, we rely on a two-level object referencing system. The ProActive backend introduces a Java class named `COG` that represents

[2] Recall that in ProActive, active object threads are plain Java threads

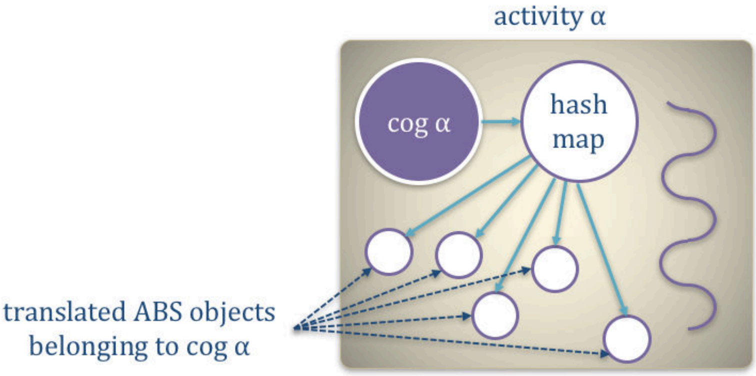


Figure 4.2 – Representation of the translation of a cog in ProActive.

ABS cogs. Figure 4.2 shows an ABS cog and its associated ABS objects, translated into ProActive. A ProActive active object of class `COG` is instantiated for each cog that is in the initial ABS program. Consequently, `COG` active objects have a global, network-wide reference that makes them accessible remotely. Thus, we consider the cog as unit of distribution. Finally, in the translated program, a `COG` active object maintains a local reference to all the translated ABS objects that it contains. For that, a hash map from object identifiers to object references is used. Thus, in the ProActive translation of an ABS program, there are only active objects of class `COG`, all the other objects are passive objects, and they are the translation of the ABS objects. In other words, the `COG` active object is the global entry point to reach the objects that it contains. This referencing system has a much reduced overhead than translating all ABS objects into ProActive active objects.

In order to translate ABS object instantiation into ProActive active/passive object instantiation, we need to translate the `new` and `new local` statements of ABS into ProActive code. As an example, consider first an ABS code snippet where a new ABS object is created in a new cog, as follows:

ABS
```
Server server = new Server();
```

The ProActive backend for ABS translates the ABS code above into the following ProActive code (the names of the ABS variables are preserved):

ProActive
```
Server server = new Server();
COG cog = PAActiveObject.newActive(COG.class, new Object[]{Server.class}, node);
server.setCog(cog);
cog.registerObject(server);
```

In the ProActive translation, Line 1 creates a regular server object. Line 2 uses the newActive ProActive primitive in order to create a new COG active object. Additionally to the parameters of the COG constructor (in an object array), ProActive allows the programmer to specify a node onto which the active object is deployed (this point is clarified in the 'Distribution' paragraph later). Line 3 lets the server object have a reference to its COG. In order to do this, in the ProActive translation we make all translated ABS objects inherit from the ABSObject class, that has the setCog method. Finally, in Line 4, we use the registerObject method of the COG class in order to make available the server object in the object registry maintained by this COG. Due to the nature of the cog object (it is an active object), and due to the by-copy semantics of ProActive, in Line 4 the server object is deeply copied in the local memory space of the created COG.

Additionally, we have to translate ABS objects that are created in the current COGwith the new local keyword. When the ProActive backend encounters this statement, the generated ProActive code is the same as above, but without the new COG instantiation. Instead, the local COG is retrieved with a synchronous method call, and the object is registered in this COG.

Object Invocation

With the approach described above, when an object is created in a new COG, in the ProActive translation there exist two objects corresponding to the new ABS object: the one that has been created locally, and the one that has been copied in the activity of the new COG. In order to enable the same object invocation model as in ABS, we translate each asynchronous ABS invocation (distinguishable with the ! syntax[3]) into two ProActive method calls that navigate through our two-level reference system: first accessing the COG of the invoked object, and then retrieving the invoked object thanks to a local identifier. In the ProActive translation, the pair (COG, identifier) is a global unique reference for each translated ABS object and, allows the generated ProActive program to retrieve any translated ABS object

[3] In case the language featured transparent invocations, there would be two possibilities to distinguish asynchronous method calls: either a static analysis of the code would have to be made beforehand to detect the presence of an active object or, the generated code would offer several branches in order adapt at runtime depending on the dynamic type of the object.

at runtime.

As an ABS example, we consider the server object reference that we have introduced previously, and we asynchronously call a start method on it:

```ABS
server!start();
```

The ProActive backend translates this asynchronous method call as follows:

```ProActive
server.getCog().execute(server.getId(), "start", new Object[]{});
```

The generic execute method of the COG class retrieves the local reference of an object, thanks to the identifier returned by the getId() method, and invokes a method on this object by reflection. As the execute method is invoked on the active object returned by the getCog method[4], this is an asynchronous method call; this puts a request in the COG's queue. The fact that the execute method call is a remote invocation implies that the parameters of this call are copied in the targeted activity. For the two first parameters of the execute method, that are the identifier of the targeted object and the name of the method to invoke, making a copy is not problem because these parameters are immutable and lightweight. The third parameter of the execute method packages the parameters of the method to invoke, if any, in an object array. In case the start method takes two parameters in ABS, like in server!start(p1, p2), the corresponding translated objects in ProActive are put in the object array parameter of the execute method call, as follows:

```ProActive
server.getCog().execute(server.getId(), "start", new Object[]{p1, p2});
```

In this case, the whole object array is copied. Thus, the translated objects p1 and p2 are manipulated by copy in the remote activity. However, in the remote activity, these copies can only be used to invoke a method on them[5]. Consequently, any method invocation, translated using the execute method, will pass by the COG that holds the original object. In the end, only the original object will ever be modified during the program execution. Note that we call 'original object' the

[4] The object returned by getCog is precisely a proxy to the remote active object; see Subsection 2.3.4.

[5] In case the language enables field access, a method for getting and setting the field would have to be generated in the translation, and any field access would have to be translated into a getter or setter call, which could be done synchronously or asynchronously depending on the targeted semantics.

object that is locally referenced by its COG. In particular, the 'original object' is not the first object that is created (with `new`), but the copy that is passed through the `registerObject` method of the `COG` class. Thanks to this approach, in the translated ProActive code, all the modifications that are made concurrently on the copies of an object are centralised and sequentialised by the COG that manages this object. This is in fact exactly the behaviour that we want to get in the translated ProActive programs, since ABS programs pass the method parameters by reference.

The fact that only one copy of an object reflects all the modifications that were made on it allows us to optimise the copies of an object, such that they are reduced to the minimum information that they need to contain. In particular, outside its original activity, there are only two characteristics of an object that are needed: the local identifier of the object, and the (global) reference to its COG. As a copy of an object is only used to reach the original one through its COG, no information about the state of the object is needed in a copy. Consequently, the ProActive backend for ABS offers a customised serialisation mechanism that only embeds the aforementioned elements when an object is passed by copy to another activity. This allows the generated ProActive program to save memory and bandwidth when it is executed.

In ABS, synchronous method invocations are handled with an asynchronous call whose future is directly gotten (with `.get`) just after. As such, we do not need to add any special handler for translating ABS synchronous method calls into ProActive, we simply use the composition of the translation of asynchronous method calls and of the translation of ABS `.get` (see below).

Request Scheduling

Active object languages often support special threading and request scheduling models. For that, they offer language constructs to the programmer in order to interact with threads and futures. Most of these constructs can be translated into ProActive thanks to the customisation capabilities of multiactive objects. We demonstrate here the ability of multiactive objects to embody cooperative scheduling of requests through the translation of the ABS `await` language construct into multiactive objects. Specifically, we consider the translation of `await` on future variables (representative of cooperative scheduling), of `get` (representative of languages that do not offer transparent futures), and also the translation of `await` on boolean conditions, and of `suspend` for completeness. The list of the features of multiactive objects that are used in order to translate ABS constructs are the following (see Subsection 2.4.2 and Chapter 3).

- Groups and compatibilities.
- Global thread limit and kind of thread limit (soft or hard).
- Thread limit per group.

The ABS `await` language construct on futures. An `await` statement releases the execution thread if the specified future is not resolved. For example, let us modify the previous ABS example in order to await on the future returned by the asynchronous method call:

ABS

```
Fut<Bool> startedFut = server!start();
await startedFut?;
```

In order to have the same behaviour in the ProActive translation, we force a wait-by-necessity. For that we use a special ProActive primitive, `getFutureValue`, as follows:

ProActive

```
PAFuture.getFutureValue(startedFut);
```

Since, in ProActive, a wait-by-necessity blocks the current thread, we need to adapt the multiactive object configuration in order to get a behaviour that is similar to what happens in ABS. Previously, we have translated all ABS method calls such that, in ProActive, they are wrapped in the `execute` method of the `COG` class. Consequently, configuring the `COG` class with multiactive object annotations is sufficient to control the way all translated ABS requests are scheduled. In practice, when the ProActive backend creates the `COG` class, it also annotates it in order to specify a thread pool limited to one thread, and a soft thread limit, which basically means that there can be many parallel threads for a `COG` object, but only one thread is not in wait-by-necessity. The ProActive backend also annotates the `execute` method, and puts it in a multiactive object group that is self compatible, so that many requests of this group can be executed on parallel threads. The multiactive object annotations applied on the `COG` class are displayed in Listing 4.1

ProActive

```
@DefineGroups({
  @Group(name="scheduling", selfCompatible=true)
})
@DefineThreadConfig(threadPoolSize=1, hardLimit=false)
public class COG {
  ...
  @MemberOf("scheduling")
  public ABSValue execute(UUID objectID, String methodName, Object[] args) {
  ...
  }
  ...
}
```

Listing 4.1 – Multiactive object annotations on the `COG` class for request scheduling.

Thanks to the fact that several `execute` requests can be executed in parallel and that at most one such request is not in wait-by-necessity, when the `getFutureValue` is executed on an unresolved future, the current thread is automatically put in wait-by-necessity. Here, the ProActive runtime can schedule another `execute` request that is not in wait-by-necessity, which is similar to the behaviour of the original ABS program.

The ABS `get` language construct. In ABS, the `get` language construct blocks the execution thread of the COG until the future is resolved, as for example on the previously created future variable:

ABS

```
Fut<Bool> startedFut = server!start();
await startedFut?;
...
Bool started = startedFut.get;
```

The translation of `get` into ProActive would have been straightforward with the blocking `getFutureValue` primitive. However, we have disabled its blocking aspect by using multiactive object annotations in order to translate `await`. In order to correctly translate `get`, we temporarily harden the kind of thread limit (i.e. all threads, even the ones in wait-by-necessity, are counted in the thread limit), so that no other thread of the `COG` object in ProActive can start or resume as long as the future is not resolved. This is because the strict thread limit of one thread is already reached considering the current thread. Each time a `get` is encountered in ABS, the following ProActive code is injected in the translated program:

ProActive

```
this.getCog().switchHardLimit(true);
PAFuture.getFutureValue(startedFut);
this.getCog().switchHardLimit(false);
```

In this code, since the retrieved COG is the local one, the `switchHardLimit` method calls are synchronous. This ensures that the kind of thread limit is effectively hardened when the future is awaited. Note that, although we adapt the thread policy dynamically, this approach is still safe because it is less permissive than the configuration that is given by the multiactive object annotations.

The ABS `await` language construct on conditions. Another role of `await` is to use it followed by a boolean condition:

ABS

```
await a == 3;
```

In the case of the ProActive backend, our approach is to wrap the evaluation of the boolean condition in a method call that returns a future. After this invocation, we can wait for this future with the `getFutureValue` primitive, like for the translation of `await` on future variables. In order to implement this approach, during the translation of an ABS class, each time an `await` on a condition is encountered, we generate a new method in its translated ProActive class. This method takes all the parameters that are needed in order to check periodically if the condition is fulfilled, and returns once it succeeds. Then, the `await` of the ABS program is translated in the ProActive program into a generic `awaitCondition` asynchronous method call, that is performed on the `COG` active object that holds the current object. The result of this call is then waited, as follows:

ProActive

```
PAFuture.getFutureValue(this.getCog().
            awaitCondition(this.getId(), "cond7517d1ff7c52", new Object[]{a}));
```

The `awaitCondition` method of the `COG` class takes as parameters the name of the generated condition method to execute, the identifier of the object that defines the generated condition method, and the parameters needed for the condition evaluation, here `a`. This asynchronous method call is wrapped in a forced wait-by-necessity in order to let another request being scheduled while the condition is evaluating. This is because the `awaitCondition` method is configured with adequate multiactive object annotations. Thanks to the call to `awaitCondition`, the `COG` object can execute the generated condition method by reflection. This approach is similar to the the translation of ABS asychronous method calls into ProActive, except that another method of the `COG` class is used. For the execution of `awaitCondition` requests, we add a multiactive group and new thread limits in the `COG` class, as can be seen on Listing 4.2:

ProActive

```
@DefineGroups({
  @Group(name="scheduling", selfCompatible=true),
  @Group(name="waiting", selfCompatible=true, minThreads=10, maxThreads=10)
})
@DefineRules({
    @Compatible({"scheduling", "waiting"}),
})
@DefineThreadConfig(threadPoolSize=11, hardLimit=false)
public class COG {
  ...
  @MemberOf("scheduling")
  public ABSValue execute(
                      UUID objectID, String methodName, Object[] args) {...}
  @MemberOf("waiting")
  public ABSValue awaitCondition(
                      UUID objectID, String conditionName, Object[] args) {...}
}
```

Listing 4.2 – Complete multiactive object annotations on the `COG` class.

In order to separate the evaluation of condition methods from the execution of translated method calls, we dedicate new threads in a COG for the evaluation of condition methods. For that, the ProActive backend sets a thread limit of 11 threads per COG object thanks to the appropriate multiactive object annotation. The `waiting` group is self compatible (to enable parallel condition evaluations) as well as compatible with the `scheduling` group, because `execute` requests can be processed at the same time as condition evaluation. Also, the `waiting` group defines precise bounds on the number of threads that are taken by the requests of this group. In particular, the number of threads that are reserved by the `waiting` group is one less than the total number of threads, in order to leave a single thread to the `scheduling` group. This ensures that the execution of applicative requests are always handled by one active thread, like in ABS.

Distribution

In addition to the translation of active object paradigms, the ABS compiler is slightly modified in order to enable distributed execution of ABS programs through their translation into ProActive proactive programs, thanks to the ProActive backend. We also make some other adaptations in the generated Java classes in order to ensure an efficient distributed execution.

Serialisation. The main challenge when moving objects from one memory space to another is to reshape them so that they can be transmitted on the network medium. In particular, a serialised version of an object must be created by the sender so that afterwards the object graph can be rebuilt from the serialised version by the recipient. As ProActive is based on Java RMI, all objects that are part of a remote method call (i.e. parameters and return values) must implement the Java `Serializable` interface, otherwise a distributed execution throws an exception. Thus, we take a particular care in making the generated Java classes implement this interface, when needed.

Copy Optimisation. As said earlier for the translation of asynchronous calls, in order to minimise copy overhead, in the ProActive translation we declare the translated class fields with the `transient` Java keyword, which prevents them from being embedded in a serialised version of an object (they are replaced by `null`). The other fields that receive a value when the object is created go through a customised serialisation mechanism: they are part of a copy only the first time they are serialised (i.e. when the object is copied in its hosting COG). Afterwards, we only embed in the copy of an object the object's identifier and the reference to its COG.

Deployment. Any distributed program needs a deployment specification mechanism in order to place pieces of the program on different machines. ProActive embeds a deployment descriptor that is based on XML configuration files, where the programmer declares physical machines to be mapped to virtual nodes (this is explained in more details in Section 5.1). Such virtual nodes can then be denominated in the program in order to deploy an active object on them. In our case, as the ProActive program is generated, we have to raise the node specification mechanism at the level of the ABS program. We slightly modified the ABS syntax (and parser) to allow the programmer to specify a node on which a new COG must be deployed, thanks to the node name. The ProActive backend then links this node name to the deployment descriptor of ProActive. Now, an ABS `new` statement is optionally followed by a string that identifies a node, as follows:

ABS

```
Server server = new "mynode" Server();
```

During translation, the ProActive node object corresponding to this node identifier is retrieved, and given as parameter of the `newActive` primitive to deploy the active object on this node. For this specification to work, a simple descriptor file (XML) similar to the following must be created and attached to the ABS program:

GCM/ProActive

```
<GCMDeployment>
  <hosts id="mynode" hostCapacity="1"/>
  <sshGroup hostList="172.16.254.1"/>
</GCMDeployment>
```

In this example, the node identified by the string `"mynode"` maps to the machine that has the IP address mentioned in the `hostList` attribute. Domain Name System (DNS) names can be specified here as well. Many other deployment options can also be defined. If several machines are specified in the `hostList`, then the ProActive backend associated each of them to a new `COG` active object in a round robin manner. If there is no node that is specified in ABS, then the new `COG` object is created on the same machine in a different JVM, enforcing this way a strict isolation of COGs.

Wrap Up and Applicability

In conclusion, by carefully setting multiactive object annotations, and by adapting to distribution requirements, we are able to execute ABS programs through distributed ProActive programs. This translation is automatically handled by the ProActive backend for ABS. We formalise and prove the correctness of this translation in Section 4.3. The approach presented here and instantiated in the case of ABS and ProActive is not restricted to this single scope: the translation concepts are generic enough to be applied to most active object languages, and systematically turn them into deployable active object languages. The effort to port our result to other active object languages depends on which of the two chosen languages they are the closest to. Indeed, ProActive and ABS characteristics encompass most of active object languages. To get an efficient translation of different active object models into distributed multiactive objects, one needs to answer the questions that are related to the active object language classification given in Section 2.2. Most of all, one must carefully consider the location of objects: "Should objects be grouped to preserve the performance of the application? If yes, how and under which control?". When it comes to distribute active objects over several memory spaces, the only viable solution is to address the objects hierarchically. This strategy is easily applicable to any active object language based on object groups. As such, adapting this work to JCoBox is straightforward. The most challenging aspect is that in JCoBox the objects share a globally accessible and immutable memory. In this case, the global memory could be translated into an active object that serves its content: since it is immutable, communicating the content by copy is correct. In the case of uniform active object languages, like Creol, creating one active object per translated object handles straightforwardly the translation but limits scalability. The best approach is to group several objects behind a same active object for performance reasons, like building abstract object groups that resemble ABS and JCoBox. In the case of Encore, objects are already scattered into active and passive objects, which makes the translation of the object model straightforward. Once the distributed organisation of objects has been defined, then preserving the semantics of the source language relies on a precise interleaving of local threads, which is accessible thanks to the various threading controls offered by multiactive objects. For instance, we have presented in the context of the ProActive backend for ABS a precise threading configuration that simulates cooperative scheduling. But other scheduling policies are easily workable with multiactive objects. For example, the transposition of the ProActive backend to AmbientTalk could seem tricky on the scheduling aspect, due to the existence of callbacks. However, a callback on a future can still be considered as a request that is immediately executed in parallel, but that starts by a wait-by-necessity on the adequate future. In conclusion, the gap between active object languages can be summarised in the global organisation of objects and threads, and we have given in this section all the directions that enable a faithful translation.

4.2.2 Evaluation

In this section, we evaluate the ProActive backend for ABS by applying it on several applications written in ABS. For all ABS applications we experiment, we compare the execution of the ProActive program generated by the ProActive backend to the execution of the program generated by the Java backend. The Java backend for ABS encodes the ABS semantics in Java and is an accurate reference for the execution of ABS applications, although it only provides a local execution. Comparing the generated ProActive programs to the generated Java programs for the same ABS appplication allows us to analyse our results knowing that both versions are run on the JVM.

The ABS tool suite[6] offers a set of examples of ABS programs, introduced in a tutorial paper [Häh13]. We first try the ProActive backend on these small ABS applications. Four applications are tested: a Deadlock program that hangs by circular dependencies between activities, a Bank account application that synchronises withdrawal operations, a Leader election algorithm over a ring, and a Chat application that redirects messages to clients. These applications are specifically oriented towards coordination between independent entities, this is why it is important to check that the behaviour of these applications is the same when they are translated by the ProActive backend and executed in a distributed manner with ProActive. The number of lines of ABS code as well as the number

[6] See `http://abs-models.org/`.

Application	Lines of ABS code	Number of COGs (ABS) = Number of JVMs (ProActive)
DEADLOCK	69	2
BANK ACCOUNT	167	3
LEADER ELECTION	62	4
CHAT	324	5

Table 4.1 – Characteristics of the ABS tool suite program examples.

of COGs for each application are displayed on Table 4.1.

In the ProActive translation of these programs, each `COG` active object is placed in a separate JVM. Only one JVM is used for executing the program generated by the Java backend. For all ABS applications, we observe that the behaviour of the generated ProActive program is the same as the behaviour of the generated Java program. In particular, the DEADLOCK program is also deadlocking in the translation, and the right results are produced in the translation of the other programs. Thus, the request scheduling enforced in ProActive respects the one of the reference implementation. Beyond the adequacy of the obtained results, we cannot use the program examples of the ABS tool suite to evaluate the performance of the generated ProActive programs. Indeed, these examples run in a few milliseconds and, as such, they are inadequate for the performance analysis in a distributed execution. We focus the rest of the experiments on a developed use case that exhibits more computational requirements.

We develop in ABS an application of pattern maching of a DNA sequence in a DNA database. In order to parallelise the search, we implement it using the MapReduce programming model [DG08]. We implement the MapReduce programming model in ABS with a `MapReduce` class that is responsible for splitting the input data and dispatching `map` and `reduce` calls to several workers (`Worker` class). Each worker is instantiated in a new COG so that workers can execute in parallel. Workers do not communicate with each other, they only communicate with the `MapReduce` object, that is in a separate COG.

We consider a searched pattern of 250 bytes, and a database of 5MB of DNA sequences. The `MapReduce` object considers 100 DNA samples of 50kB each, and creates 100 pairs of `(pattern, sample)`. The pairs are distributed evenly to the workers and processed in parallel through the `map` method. Each mapper tries to match the pattern on the given sample, and returns the maximum matching sequence found in this sample. Once all mappers finish, all the results that are locally produced by a mapper are aggregated and passed to a single reducer that outputs the global maximum matching sequence. The general algorithm that is used to match the pattern is shown on algorithm 1.

```
while pattern not ended do
    while sample not ended do
        while pattern matching sample do
            update maximum matching
        end while
    end while
end while
```

Algorithm 1 Maximum matching sequence algorithm (mapper phase).

We compute the global execution time of the use case when varying the number of workers, i.e. when varying the degree of parallelism. When executing the program given by the Java backend, we execute it on a single machine, since it does not support distribution. When using the ProActive backend, we deploy two `COG` active objects (i.e. two workers) per machine. We used a cluster of the Grid5000 distributed platform [Cap+05]. All machines have 2 CPUs of 2.6GHz with 2 cores, and 8GB of memory.

Figure 4.3 shows the execution time of both ProActive and Java translations of the ABS application from 2 to 50 workers. Therefore, 1 to 25 machines are used in the case of the ProActive execution.

The execution time of the application generated by the ProActive backend is sharply decreasing for the first added machines and then decreases at a slower rate. On the other hand, the execution of the Java program reaches an optimal degree of parallelism of 4 workers (which actually is the number of cores of the machine) and then cannot benefit from higher parallelism; execution time even increases afterwards due to context switching overhead, as the thread-based parallelism happens on the same machine. On the opposite, increasing the degree of parallelism for the application generated by the ProActive backend gives a linear speedup from

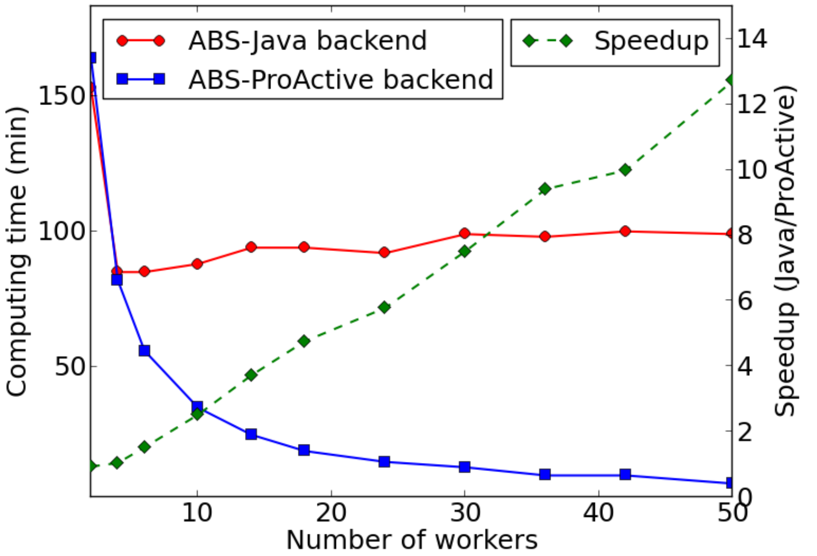


Figure 4.3 – Execution time of DNA matching application. Each measurement is an average of five executions.

the local Java programs, because it balances the load between machines. In the best case, the application completes 12 times faster with the ProActive backend. We can highlight that simply using 10 machines makes the application complete 5 times faster, and this is because the machines that are used are only half loaded: two active objects are deployed on each of them although they have four available cores. Still, thanks to the ProActive backend, the application completes precisely in 19 minutes instead of 1 hour and 35 minutes in the case of the Java backend. Nonetheless, we can notice that for the first experiment point, the program generated by the Java backend performs better than the ProActive one: precisely, it is 7% faster. This is due to the overhead induced by the distributed execution, involving a cost for serialisation and communication. This shows that the ProActive backend can be used for ABS applications that balance coordination aspects with computational aspects.

We know now that the ProActive backend is beneficial thanks to the distributed execution of ABS programs, compared to a local execution. The next question is whether it is effective compared to a program that would be directly written in ProActive. In other words, we want to evaluate the quality of the generated code,

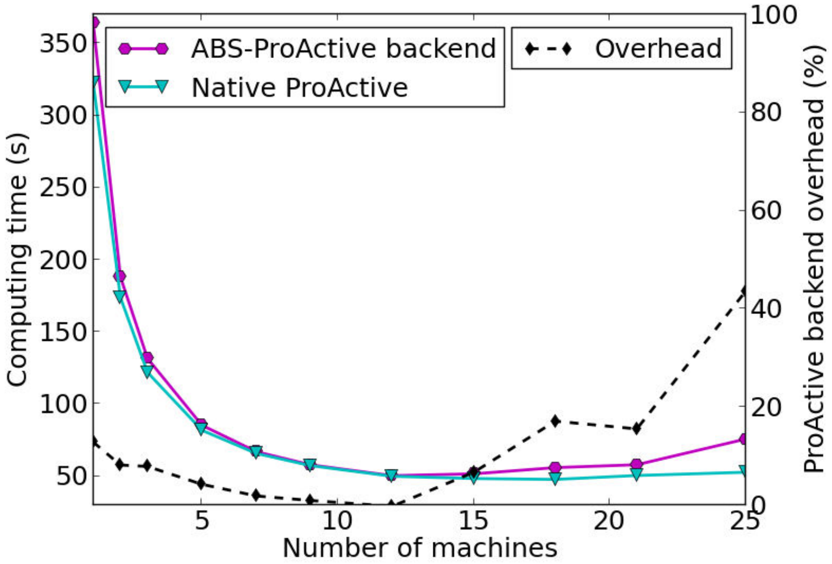


Figure 4.4 – Execution time of DNA matching application without functional layer.

and the overhead of the simulation of ABS semantics with multiactive objects. Figure 4.4 compares the execution time of the DNA matching application in two versions: the first one is the generated program of the ProActive backend for ABS, and the second one is a hand-written version of the application directly in ProActive, i.e. executing according to ProActive semantics. For this experiment, in the program generated by the ProActive backend, we have manually replaced the translation of functional ABS types (integers, booleans, lists, and maps) with the corresponding standard Java types. Indeed, the implementation of the functional layer of ABS in Java is not as efficient as primitive Java types. Considering that our point is to evaluate the additional communication cost of the ProActive translation, we cannot let the performance of the implementation of ABS types in Java confuse our evaluation[7]. In this setting, we can see that the overhead introduced by the ProActive backend is rather low compared to a native version of the application, since it is generally kept under 10%. At the biggest stage, the generated application starts having a higher overhead because it involves too much commu-

[7] As can be compared in the case of the generated ProActive execution of Figures 4.3 and 4.4, the difference is of an order of magnitude depending on the implementation of types; fixing this problem has been first explored in [Ser+14].

nication compared to the computational load of each entity, and since this glut of communication has even more importance in the translated program (because of additional accesses to intermediate objects), then the application's execution suffers even more from it.

In conclusion, the ProActive backend allows us to turn an ABS program into a high-performance distributed application. In addition, the ProActive code, generated by the ProActive backend and simulating the ABS semantics, maintains a low overhead compared to a hand-written ProActive application.

4.3 Formalisation and Translation Correctness

Beyond the didactic introduction of the ProActive backend for ABS in Section 4.2, this section aims at proving the correctness of the translation given by the backend, i.e. that a generated ProActive program simulates well an original ABS program. In order to establish the correctness of the translation, we use the formalisation of the two languages: the ABS semantics, and the formalisation of the ProActive library, namely MultiASP. We recall the two calculus in Subsection 4.3.1. Then, we introduce in Subsection 4.3.2 the formalisation of the request scheduling mechanisms that were presented in Chapter 3, as an extension of MultiASP. From the complete semantics of the two languages, we establish a translational semantics and an equivalence relation in Subsection 4.3.3. Finally, after having listed the restrictions and defined some properties in Subsection 4.3.4, we prove the correctness of the translation through the proof of two precise theorems.

4.3.1 Recall of ABS and MultiASP Semantics

ABS Syntax and Semantics

For the formalisation of ABS, we refer to the ABS syntax and semantics given in [GLL15]. In order to make this thesis standalone, we recall the syntax of ABS in Figure 4.5, to which we added the `while` and `suspend` statements. This part of the ABS syntax focuses on the concurrent object layer of ABS. In particular, the syntax for its functional layer is omitted. We recall the runtime syntax of ABS in

$$
\begin{array}{rcll}
P & ::= & \overline{I}\ \overline{C}\ \{\overline{T\ x\ ;}\ s\} & \text{program} \\
T & ::= & \mathit{primitive\text{-}type} \mid \texttt{Fut <}T\texttt{>} \mid \texttt{I} & \text{type} \\
I & ::= & \texttt{interface I}\ \{(\overline{S\ ;}\} & \text{interface} \\
S & ::= & T\ \texttt{m}(\overline{T\ x}) & \text{method signature} \\
C & ::= & \texttt{class C}(\overline{T\ x})\ [\texttt{implements}\ \overline{\texttt{I}}]\ \{\overline{T\ x\ ;}\ \overline{M}\} & \text{class} \\
M & ::= & S\ \{\overline{T\ x\ ;}\ s\} & \text{method definition} \\
g & ::= & b \mid x? \mid g \wedge g' & \text{guard} \\
s & ::= & \texttt{skip} \mid x = z \mid \texttt{if}\ e\ \{\, s\,\}\ \texttt{else}\ \{\, s\,\} \mid \texttt{while}\ b\ \{\, s\,\} & \text{statement} \\
 & & \mid \texttt{return}\ e \mid s; s \mid \texttt{await}\ g \mid \texttt{suspend} & \\
z & ::= & e \mid e.\texttt{m}(\overline{e}) \mid e!\texttt{m}(\overline{e}) \mid \texttt{new}\ [\mathit{local}]\ \texttt{C}(\overline{e}) \mid x.\texttt{get} & \text{expression with side effects} \\
e & ::= & v \mid x \mid \texttt{this} \mid \mathit{arithmetic\text{-}bool\text{-}exp} & \text{expression} \\
v & ::= & \texttt{null} \mid \mathit{primitive\text{-}val} & \text{value}
\end{array}
$$

Figure 4.5 – Syntax of the concurrent object layer of ABS.

$$
cn \quad ::= \quad \epsilon \mid \mathit{fut}(f, \mathit{val}) \mid \mathit{ob}(o, a, p, q) \mid \mathit{invoc}(o, f, \texttt{m}, \overline{v}) \mid \mathit{cog}(c, \mathit{act}) \mid cn\ cn
$$

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
\mathit{act} & ::= & o \mid \varepsilon & \mathit{val} & ::= & v \mid \bot \\
p & ::= & \{l \mid s\} \mid \texttt{idle} & a & ::= & [\cdots, x \mapsto v, \cdots] \\
q & ::= & \epsilon \mid \{l \mid s\} \mid q\ q & v & ::= & o \mid f \mid \ldots \\
s & ::= & \texttt{cont}(f) \mid \ldots & & &
\end{array}
$$

Figure 4.6 – Runtime syntax of ABS.

Figure 4.6. An ABS configuration consists of objects ($ob(...)$), COGs ($cog(...)$), invocations ($invoc(...)$), and futures ($fut(...)$). The operational semantics of ABS is given on Figure 4.7. The notation $[\![e]\!]^A$ represents the evaluation of expression e in ABS. This semantics is slightly changed from the one of [GLL15], which already represents an adapted semantics of the ABS that was preliminary introduced in [Joh+11] (minor updates). In the version presented here, the modifications we make on the ABS semantics simply aim at making a clearer proof, syntactically; the essence of the translation is preserved. For example, we make no difference between the initialisation of object fields through constructors or through a method call at instantiation. This allows us to avoid considering initialisation methods in the equivalence relation, which makes it easier to read. More precisely, we give below the list of the modifications we bring to the ABS semantics:

- The distinction between class fields and class parameters has been removed, they are syntactic sugar for class fields initialised automatically to `null`.

$$\begin{array}{c}\text{(S\textsc{kip})}\\ ob(o,a,\{l \mid \texttt{skip}; s\},q)\\ \overset{\text{A}}{\rightarrow} ob(o,a,\{l \mid s\},q)\end{array}$$

$$\begin{array}{c}\text{(A\textsc{ssign}-L\textsc{ocal})}\\ \dfrac{x \in \mathrm{dom}(l) \quad v = \llbracket e \rrbracket^{A}_{(a+l)}}{\begin{array}{c} ob(o,a,\{l \mid x = e; s\},q)\\ \overset{\text{A}}{\rightarrow} ob(o,a,\{l[x \mapsto v] \mid s\},q)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(A\textsc{ssign}-F\textsc{ield})}\\ \dfrac{x \in \mathrm{dom}(a) \setminus \mathrm{dom}(l) \quad v = \llbracket e \rrbracket^{A}_{(a+l)}}{\begin{array}{c} ob(o,a,\{l \mid x = e; s\},q)\\ \overset{\text{A}}{\rightarrow} ob(o,a[x \mapsto v],\{l \mid s\},q)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(C\textsc{ond}-T\textsc{rue})}\\ \dfrac{\texttt{true} = \llbracket e \rrbracket^{A}_{(a+l)}}{\begin{array}{c} ob(o,a,\{l \mid \texttt{if}\ e\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\},q)\\ \overset{\text{A}}{\rightarrow} ob(o,a,\{l \mid s_1; s\},q)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(C\textsc{ond}-F\textsc{alse})}\\ \dfrac{\texttt{false} = \llbracket e \rrbracket^{A}_{(a+l)}}{\begin{array}{c} ob(o,a,\{l \mid \texttt{if}\ e\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\},q)\\ \overset{\text{A}}{\rightarrow} ob(o,a,\{l \mid s_2; s\},q)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(A\textsc{wait}-T\textsc{rue})}\\ \dfrac{f = \llbracket x \rrbracket^{A}_{(a+l)} \quad v \neq \bot}{\begin{array}{c} ob(o,a,\{l \mid \texttt{await}\, x\,\texttt{?}; s\},q)\ \mathit{fut}(f,v)\\ \overset{\text{A}}{\rightarrow} ob(o,a,\{l \mid s\},q)\ \mathit{fut}(f,v)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(A\textsc{wait}-F\textsc{alse})}\\ \dfrac{f = \llbracket x \rrbracket^{A}_{(a+l)}}{\begin{array}{c} ob(o,a,\{l \mid \texttt{await}\, x\,\texttt{?}; s\},q)\ \mathit{fut}(f,\bot)\\ \overset{\text{A}}{\rightarrow} ob(o,a,\texttt{idle},q \cup \{l \mid \texttt{await}\, x\,\texttt{?}; s\})\ \mathit{fut}(f,\bot)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(R\textsc{elease}-C\textsc{og})}\\ ob(o,a,\texttt{idle},q)\ \mathit{cog}(c,o)\\ \overset{\text{A}}{\rightarrow} ob(o,a,\texttt{idle},q)\ \mathit{cog}(c,\epsilon)\end{array}$$

$$\begin{array}{c}\text{(A\textsc{ctivate})}\\ \dfrac{c = a(\mathrm{cog})}{\begin{array}{c} ob(o,a,\texttt{idle},q \cup \{l \mid s\})\ \mathit{cog}(\alpha,\epsilon)\\ \overset{\text{A}}{\rightarrow} ob(o,a,\{l \mid s\},q)\ \mathit{cog}(c,o)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(R\textsc{ead}-F\textsc{ut})}\\ \dfrac{f = \llbracket e \rrbracket^{A}_{(a+l)} \quad v \neq \bot}{\begin{array}{c} ob(o,a,\{l \mid x = e.\texttt{get}; s\},q)\ \mathit{fut}(f,v)\\ \overset{\text{A}}{\rightarrow} ob(o,a,\{l \mid x = v; s\},q)\ \mathit{fut}(f,v)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(N\textsc{ew}-O\textsc{bject})}\\ \dfrac{\begin{array}{c} o' = \mathrm{fresh}(\texttt{C}) \quad \mathit{fields}(\texttt{C}) = \overline{x}\\ \overline{v} = \llbracket \overline{e} \rrbracket^{A}_{(a+l)} \quad a' = [\overline{x \mapsto v}, \mathrm{cog} \mapsto c]\end{array}}{\begin{array}{c} ob(o,a,\{l \mid x = \texttt{new local C}(\overline{e}); s\},q)\ \mathit{cog}(c,o)\\ \overset{\text{A}}{\rightarrow} ob(o,a,\{l \mid x = o'; s\},q)\ \mathit{cog}(c,o)\\ ob(o',a',\texttt{idle},\varnothing)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(N\textsc{ew}-C\textsc{og}-O\textsc{bject})}\\ \dfrac{\begin{array}{c} c' = \mathrm{fresh}(\,) \quad o' = \mathrm{fresh}(\texttt{C}) \quad \mathit{fields}(\texttt{C}) = \overline{x}\\ \overline{v} = \llbracket \overline{e} \rrbracket^{A}_{(a+l)} \quad a' = [\overline{x \mapsto v}, \mathrm{cog} \mapsto c']\end{array}}{\begin{array}{c} ob(o,a,\{l \mid x = \texttt{new C}(\overline{e}); s\},q)\\ \overset{\text{A}}{\rightarrow} ob(o,a,\{l \mid x = o'; s\},q)\\ ob(o',a',\texttt{idle},\varnothing) \quad \mathit{cog}(c',\epsilon)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(R\textsc{endez-vous}-C\textsc{omm})}\\ \dfrac{\begin{array}{c} f = \mathrm{fresh}(\ ) \quad o' = \llbracket e \rrbracket^{A}_{(a+l)} \quad \overline{v} = \llbracket \overline{e} \rrbracket^{A}_{(a+l)}\\ p'' = \mathrm{bind}(o', f, m, \overline{v}, \texttt{class}(o'))\end{array}}{\begin{array}{c} ob(o,a,\{l \mid x = e\texttt{!m}(\overline{e}); s\},q)\ ob(o',a',p',q')\\ \overset{\text{A}}{\rightarrow} ob(o,a,\{l \mid x = f; s\},q)\ ob(o',a',p',q' \cup p'')\ \mathit{fut}(f,\bot)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(C\textsc{ontext})}\\ \dfrac{cn \overset{\text{A}}{\rightarrow} cn'}{cn\ cn'' \overset{\text{A}}{\rightarrow} cn'\ cn''}\end{array}$$

$$\begin{array}{c}\text{(C\textsc{og}-S\textsc{ync}-C\textsc{all})}\\ \dfrac{\begin{array}{c} o' = \llbracket e \rrbracket^{A}_{(a+l)} \quad \overline{v} = \llbracket \overline{e} \rrbracket^{A}_{(a+l)} \quad f = \mathrm{fresh}(\,)\\ c = a'(\mathrm{cog}) \quad f' = l(\mathrm{destiny})\\ \{l' \mid s'\} = \mathrm{bind}(o', f, \texttt{m}, \overline{v}, \mathit{class}(o'))\end{array}}{\begin{array}{c} ob(o,a,\{l \mid x = e\texttt{.m}(\overline{e}); s\},q)\\ ob(o',a',\texttt{idle},q')\ \mathit{cog}(c,o)\\ \overset{\text{A}}{\rightarrow} ob(o,a,\texttt{idle},q \cup \{l \mid \texttt{await}\ f\texttt{?}; x = f.\texttt{get}; s\})\ \mathit{fut}(f,\bot)\\ ob(o',a',\{l' \mid s'; \texttt{cont}\ f'\},q')\ \mathit{cog}(c,o')\end{array}}\end{array}$$

$$\begin{array}{c}\text{(C\textsc{og}-S\textsc{ync}-R\textsc{eturn}-S\textsc{ched})}\\ \dfrac{c = a'(\mathrm{cog}) \quad f = l'(\mathrm{destiny})}{\begin{array}{c} ob(o,a,\{l \mid \texttt{cont}\ f\},q)\ \mathit{cog}(c,o)\\ ob(o',a',\texttt{idle},q' \cup \{l' \mid s\})\\ \overset{\text{A}}{\rightarrow} ob(o,a,\texttt{idle},q)\ \mathit{cog}(c,o')\\ ob(o',a',\{l' \mid s\},q')\end{array}}\end{array}$$

$$\begin{array}{c}\text{(S\textsc{elf}-S\textsc{ync}-C\textsc{all})}\\ \dfrac{\begin{array}{c} f' = l(\mathrm{destiny}) \quad o = \llbracket e \rrbracket^{A}_{(a+l)} \quad \overline{v} = \llbracket \overline{e} \rrbracket^{A}_{(a+l)}\\ f = \mathrm{fresh}(\,) \quad \{l' \mid s'\} = \mathrm{bind}(o, f, \texttt{m}, \overline{v}, \mathit{class}(o))\end{array}}{\begin{array}{c} ob(o,a,\{l \mid x = e\texttt{.m}(\overline{e}); s\},q)\\ \overset{\text{A}}{\rightarrow} ob(o,a,\{l' \mid s'; \texttt{cont}(f')\},q \cup \{l \mid \texttt{await}\ f\texttt{?}; x = f.\texttt{get}; s\})\\ \mathit{fut}(f,\bot)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(R\textsc{eturn})}\\ \dfrac{v = \llbracket e \rrbracket^{A}_{(a+l)} \quad f = l(\mathrm{destiny})}{\begin{array}{c} ob(o,a,\{l \mid \texttt{return}\ e; s\},q)\ \mathit{fut}(f,\bot)\\ \overset{\text{A}}{\rightarrow} ob(o,a,\texttt{idle},q)\ \mathit{fut}(f,v)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(R\textsc{em}-S\textsc{ync}-C\textsc{all})}\\ \dfrac{o' = \llbracket e \rrbracket^{A}_{(a+l)} \quad f = \mathrm{fresh}(\,) \quad a(\mathrm{cog}) \neq a'(\mathrm{cog})}{\begin{array}{c} ob(o,a,\{l \mid x = e\texttt{.m}(\overline{e}); s\},q)\ ob(o',a',p,q')\\ \overset{\text{A}}{\rightarrow} ob(o,a,\{l \mid f = e\texttt{!m}(\overline{e}); x = f.\texttt{get}; s\},q)\\ ob(o',a',p,q')\end{array}}\end{array}$$

$$\begin{array}{c}\text{(S\textsc{elf}-S\textsc{ync}-R\textsc{eturn}-S\textsc{ched})}\\ \dfrac{f = l'(\mathrm{destiny})}{\begin{array}{c} ob(o,a,\{l \mid \texttt{cont}(f)\},q \cup \{l' \mid s\})\\ \overset{\text{A}}{\rightarrow} ob(o,a,\{l' \mid s\},q)\end{array}}\end{array}$$

Figure 4.7 – Semantics of ABS.

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
v & ::= & o \mid \alpha \mid \ldots & \mathit{Storable} & ::= & [\overrightarrow{x \mapsto v}] \mid v \mid f \\
\mathit{elem} & ::= & \textsc{fut}(f, v, \sigma) \mid \textsc{fut}(f, \bot) \mid \textsc{act}(\alpha, o, \sigma, p, Rq) & \sigma & ::= & \overrightarrow{o \mapsto \mathit{Storable}} \\
\mathit{cn} & ::= & \overline{\mathit{elem}} & q & ::= & (f, m, \overline{v}) \\
E & ::= & \{\ell \mid s\} & Rq & ::= & \varnothing \mid q :: Rq \\
F & ::= & E \mid E :: F & \ell & ::= & \texttt{this} \mapsto v, \overrightarrow{x \mapsto v} \\
p & ::= & \overrightarrow{q \mapsto F} & s & ::= & x = \bullet \mid \ldots
\end{array}
$$

Figure 4.8 – The runtime syntax of MultiASP.

- Linked to the previous modification, an object is `idle` when it is instantiated, instead of having the *init* request in the request queue (in the New-Object case) or as current task (in the New-Cog-Object case). Consequently, a cog that is newly created has no current activated object. We will see that with the restrictions on the translation given in Subsection 4.3.4, this modification has no impact because we can still force a method being executed first.

- Statements following a `return` instruction are discarded. This fits better with most of mainstream programming languages. In addition, having a second `return` in further statements would cause a deadlock in the semantics since there would be no recipient for the associated future, so the rule could not be reduced in this case[8].

MultiASP Syntax and Semantics

The calculus of ProActive, MultiASP, takes its foundations in the mono-threaded active object language ASP (see Subsection 2.3.4). MultiASP has been first introduced in [HHI13]. Compared to this preliminary version, the MultiASP that is presented in this thesis has a class-based syntax (briefly introduced in Subsection 2.4.3) instead of a syntax that is based on objects, and its semantics has been adapted accordingly. This modification aims at making the syntax of MultiASP closer to the syntax of ABS, enabling a more precise correspondence between the two languages. In this thesis, we also extend the preliminary version of MultiASP with threading capabilities in Subsection 4.3.2. For now, we simply

[8] Note that the original Java backend for ABS already implements this semantics for the `return` rule.

present the core of the language. MultiASP represents a classical object-oriented language except the presence of the `newActive` keyword that is used to instantiate a ProActive active object. Like in ABS, the runtime syntax of MultiASP consists of a transition relation between configurations. It is shown in Figure 4.8 and detailed hereafter. At runtime, the set of elements of a MultiASP configuration are of two kinds: activities and future binders, described below. The transition relation uses three infinite sets: *object locations* in the local store, ranged over by the terms $\{o,\ o',\ \cdots\}$; *active objects names*, ranged over by the terms $\{\alpha,\ \beta,\ \cdots\}$; and *future names*, ranged over by the terms $\{f,\ f',\ \cdots\}$. Additionally, the following terms are defined:

Values are distinguished between *runtime values* $(v,\ \cdots)$ and *storable values*. Runtime values can be either static values, object locations, or active object names. An object is represented as a mapping from field names to their values, denoted with $[\overrightarrow{x \mapsto v}]$. Storable values are either objects, futures or runtime values.

Activities are of the form $\text{ACT}(\alpha, o, \sigma, p, Rq)$. An activity contains terms that define:
– α, the name of the activity.
– o, the location of the active object in σ.
– σ, a local store mapping object locations to storable values.
– Rq, a FIFO queue of requests, awaiting to be served.
– p, a set of requests currently served: it is a mapping from requests to their thread F. A thread is a stack of methods being executed, and each method execution E consists of local variables that are present in ℓ and of a statement s to execute. The first method of the stack is the one that is executing, the others have been put in the stack due to previous local synchronous method calls. ℓ is a mapping from local variables (including `this`) to runtime values.

Future binders are of two forms. The form $\text{FUT}(f, \bot)$ means that the value of the future has not been computed yet: it is an unresolved future. The form $\text{FUT}(f, v, \sigma)$ is used when the value of the future is known: it is a resolved future. If the future's value is a passive object, then v will be its location in the piece of store σ. As only active objects are remotely accessible, the

$$\begin{array}{rll}
[\![\mathit{primitive\text{-}val}]\!]_{(\sigma+\ell)} \triangleq \mathit{primitive\text{-}val} & [\![f]\!]_{(\sigma+\ell)} \triangleq \bot & \\
[\![\alpha]\!]_{(\sigma+\ell)} \triangleq \alpha & [\![\texttt{null}]\!]_{(\sigma+\ell)} \triangleq \texttt{null} & \\
[\![x]\!]_{(\sigma+\ell)} \triangleq [\![\ell(x)]\!]_{(\sigma+\ell)} & \text{if } x \in \mathrm{dom}(\ell) & \\
[\![x]\!]_{(\sigma+\ell)} \triangleq [\![\ell(\texttt{this})(x)]\!]_{(\sigma+\ell)} & \text{if } x \notin \mathrm{dom}(\ell) & \\
[\![o]\!]_{(\sigma+\ell)} \triangleq o & \text{if } \sigma(o) = f \text{ or } \sigma(o) = [\overrightarrow{x \mapsto v}] & \\
[\![o]\!]_{(\sigma+\ell)} \triangleq [\![\sigma(o)]\!]_{(\sigma+\ell)} & \text{else} &
\end{array}$$

Figure 4.9 – Evaluation function

part of the store referenced by this location must also be transmitted when the future's value is sent back to the caller. This step involves a serialisation mechanism that is explained in the auxiliary functions below.

In the reduction rules of MultiASP semantics, we state that object o is fresh if it does not exist in the store in which it will be added. Similarly, a future or an activity name is fresh if it does not exist in the current configuration. The freshness of an object or a future is also used in a similar way in the semantics of ABS. We denote mappings by $\overrightarrow{\_ \mapsto \_}$, and use union $\cup$ (resp. disjoint union $\uplus$) over mappings. Mapping updates are of the form $\sigma[x \mapsto v]$, where σ is a mapping and the new value associated to x is v. *dom* returns the domain of a mapping. Additionally, the following auxiliary functions are used in the semantic rules.

– $[\![e]\!]_{(\sigma+\ell)}$ returns the value of e by computing the arithmetic and boolean expressions and retrieving the values stored in σ or ℓ, according to the evaluation function of Figure 4.9. It returns a value and, if the value is a reference to a location in the store, it follows references recursively; it only returns a location if the location points to an object [9].

– $[\![\overline{e}]\!]_{(\sigma+\ell)}$ returns the tuple of values of $\overline{e}$.

– *fields*(C) returns the fields as defined in the class declaration C.

– *bind* initialises method execution: $\mathrm{bind}(o, \mathtt{m}, \overline{v'}) = \{\overrightarrow{y \mapsto v'}, \overrightarrow{z \mapsto \mathtt{null}}, \mathtt{this} \mapsto o \mid s\}$, where the arguments of method m, typed in the class of o, are $\overline{y}$, and where the method body is $\{\overline{z}; s\}$.

[9] Note that there is no guarantee that this terminates a priori.

- *ready* is a predicate deciding whether a request q in the queue Rq is ready to be served: $\text{ready}(q, p, Rq)$ is *true* if q is compatible with all requests in p (currently served by the activity) and with all requests in Rq that have been received before q. More formally: $\text{ready}(q, p, Rq) = \texttt{true}$ iff $\forall q' \in \text{dom}(p).\mathit{compatible}(q, q') \wedge \forall q' \in Rq.\mathit{compatible}(q, q')$.

- Serialisation reflects the communication style happening in Java RMI. All references to passive objects are serialised when communicated between activities, so that they are always handled locally. We formalise a serialisation algorithm that marks and copies the objects to serialise recursively. *serialise* is defined as the mapping verifying the following constraints:

$$\begin{array}{l}\text{serialise}(o, \sigma) = (o \mapsto \sigma(o)) \cup \text{serialise}(\sigma(o), \sigma)\\ \text{serialise}([\overrightarrow{\overline{x \mapsto v}}], \sigma) = \bigcup_{v' \in \overline{v}} \text{serialise}(v', \sigma)\\ \text{serialise}(f, \sigma) = \text{serialise}(\alpha, \sigma) = \text{serialise}(\texttt{null}, \sigma) = \varnothing\\ \text{serialise}(\mathit{primitive\text{-}val}, \sigma) = \varnothing\end{array}$$

- $\text{rename}_\sigma(\overline{v}, \sigma')$ renames the object locations appearing in σ' and $\overline{v}$, making them disjoint from the object locations of σ; it returns the renamed set of values $\overline{v'}$ and another store σ'', as: $(\overline{v'}, \sigma'')$.

Figure 4.10 shows the semantics of MultiASP. The rules only show the activities and futures involved in the reduction, the rest of the configuration is kept unchanged. Most of the rules are triggered depending on the shape of the first statement of an activity's thread. The reduction rules of MultiASP are described below:

- Serve picks the first request that is ready in the queue (i.e. compatible with executing requests and with older requests in the queue) and allocates a new thread to serve it. It fetches the method body and creates the execution context.

- Assign-Local assigns a value to a local variable. If the statement to be executed is an assignment of an expression that can be reduced to a value, then the mapping of local variables is updated accordingly.

$$\textsc{Serve}\quad \frac{\text{ready}(q,p,Rq) \qquad q=(f,m,\overline{v}) \qquad \text{bind}(o_\alpha,m,\overline{v})=\{\ell \mid s\}}{\textsc{act}(\alpha,o_\alpha,\sigma,p,Rq :: q :: Rq') \rightarrow \textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell|s\}\} \uplus p, Rq :: Rq')}$$

$$\textsc{Assign-Local}\quad \frac{x \in \text{dom}(\ell) \qquad v = \llbracket e \rrbracket_{(\sigma+\ell)}}{\textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid x=e;s\} :: F\} \uplus p, Rq) \rightarrow \textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell[x \mapsto v] \mid s\} :: F\} \uplus p, Rq)}$$

$$\textsc{Assign-Field}\quad \frac{\ell(this)=o \qquad x \in \text{dom}(\sigma(o)) \qquad x \notin dom(\ell) \qquad \sigma' = \sigma[o \mapsto (\sigma(o)[x \mapsto \llbracket e \rrbracket_{(\sigma+\ell)}])]}{\textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid x=e;s\} :: F\} \uplus p, Rq) \rightarrow \textsc{act}(\alpha,o_\alpha,\sigma',\{q \mapsto \{\ell \mid s\} :: F\} \uplus p, Rq)}$$

$$\textsc{New-Object}\quad \frac{\mathit{fields}(\mathtt{C})=\overline{x} \qquad o \ \text{fresh} \qquad \sigma' = \sigma \cup \{o \mapsto [\overrightarrow{x=v}]\} \qquad \llbracket \overline{e} \rrbracket_{(\sigma+\ell)} = \overline{v}}{\begin{array}{c}\textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid x=\mathtt{new}\ \mathtt{C}(\overline{e});s\} :: F\} \uplus p, Rq) \\ \rightarrow \textsc{act}(\alpha,o_\alpha,\sigma',\{q \mapsto \{\ell \mid x=o;s\} :: F\} \uplus p, Rq)\end{array}}$$

$$\textsc{New-Active}\quad \frac{\mathit{fields}(\mathtt{C})=\overline{x} \qquad o,\gamma \ \text{fresh} \qquad \sigma' = \{o \mapsto [\overrightarrow{x=v}]\} \cup \text{serialise}(\overline{v},\sigma) \qquad \llbracket \overline{e} \rrbracket_{(\sigma+\ell)} = \overline{v}}{\begin{array}{c}\textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid x=\mathtt{newActive}\ \mathtt{C}(\overline{e});s\} :: F\} \uplus p, Rq) \\ \rightarrow \textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid x=\gamma;s\} :: F\} \uplus p, Rq) \quad \textsc{act}(\gamma,o,\sigma',\varnothing,\varnothing)\end{array}}$$

$$\textsc{Invk-Active}\quad \frac{\begin{array}{c}\llbracket e \rrbracket_{(\sigma+\ell)} = \beta \qquad \llbracket \overline{e} \rrbracket_{(\sigma+\ell)} = \overline{v} \\ f,o \ \text{fresh} \qquad \sigma_1 = \sigma \cup \{o \mapsto f\} \qquad (\overline{v_r},\sigma_r) = \text{rename}_{\sigma'}(\overline{v},\text{serialise}(\overline{v},\sigma)) \qquad \sigma'' = \sigma' \cup \sigma_r\end{array}}{\begin{array}{c}\textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid x=e.\mathtt{m}(\overline{e});s\} :: F\} \uplus p, Rq) \ \textsc{act}(\beta,o_\beta,\sigma',p',Rq') \\ \rightarrow \textsc{act}(\alpha,o_\alpha,\sigma_1,\{q \mapsto \{\ell \mid x=o;s\} :: F\} \uplus p, Rq) \quad \textsc{act}(\beta,o_\beta,\sigma'',p',Rq' :: (f,m,\overline{v_r})) \quad \textsc{fut}(f,\bot)\end{array}}$$

$$\textsc{Invk-Passive}\quad \frac{\llbracket e \rrbracket_{(\sigma+\ell)} = o \qquad \llbracket \overline{e} \rrbracket_{(\sigma+\ell)} = \overline{v} \qquad \text{bind}(o,m,\overline{v}) = \{\ell' \mid s'\}}{\begin{array}{c}\textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid x=e.\mathtt{m}(\overline{e});s\} :: F\} \uplus p, Rq) \\ \rightarrow \textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell' \mid s'\} :: \{\ell \mid x=\bullet;s\} :: F\} \uplus p, Rq)\end{array}}$$

$$\textsc{Return-Local}\quad \frac{v = \llbracket e \rrbracket_{(\sigma+\ell)}}{\begin{array}{c}\textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid \mathtt{return}\ e;s_r\} :: \{\ell' \mid x=\bullet;s\} :: F\} \uplus p, Rq) \\ \rightarrow \textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell' \mid x=v;s\} :: F\} \uplus p, Rq)\end{array}}$$

$$\textsc{Return}\quad \frac{v = \llbracket e \rrbracket_{(\sigma+\ell)}}{\begin{array}{c}\textsc{act}(\alpha,o_\alpha,\sigma,\{(f,m,\overline{v}) \mapsto \{\ell \mid \mathtt{return}\ e;s_r\}\} \uplus p, Rq) \ \textsc{fut}(f,\bot) \\ \rightarrow \textsc{act}(\alpha,o_\alpha,\sigma,p,Rq) \quad \textsc{fut}(f,v,\text{serialise}(v,\sigma))\end{array}}$$

$$\textsc{Update}\quad \frac{\sigma(o)=f \qquad (v_r,\sigma_r) = \text{rename}_\sigma(v,\sigma') \qquad \sigma'' = \sigma[o \mapsto v_r] \cup \sigma_r}{\textsc{act}(\alpha,o_\alpha,\sigma,p,Rq) \ \textsc{fut}(f,v,\sigma') \rightarrow \textsc{act}(\alpha,o_\alpha,\sigma'',p,Rq) \quad \textsc{fut}(f,v,\sigma')}$$

Figure 4.10 – Semantics of MultiASP.

- Assign-Field assigns a value to a field of the current object (the one pointed to by `this`). It is similar to the previous rule except that it modifies the local store.

- New-Object creates a new local object in the store at a fresh location, after evaluation of the object parameters. The new object is assigned to a field or to a local variable by one of the two rules above.

- New-Active creates a new activity that contains a new active object. It picks a fresh activity name, and assigns the serialised object parameters: the initial local store of the activity is the piece of store referenced by the parameters. The freshness of the new active object ensures that it is not already in the serialised store.

- Invk-Active performs an asynchronous remote method invocation on an active object. It creates a fresh future with undefined value. The arguments of the invocation are serialised and put in the store of the invoked activity, possibly renaming locations to avoid clashes. The special case $\alpha = \beta$ requires a trivial adaptation with the rule Invk-Active-Self:

Invk-Active-Self

MultiASP

$$\frac{\begin{array}{c}[\![e]\!]_{(\sigma+\ell)} = \alpha \qquad [\![\overline{e}]\!]_{(\sigma+\ell)} = \overline{v} \qquad f, o \ \text{ fresh} \qquad \sigma_1 = \sigma \uplus \{o \mapsto f\} \\ (\overline{v_r}, \sigma_r) = \text{rename}_{\sigma_1}(\text{serialise}(\overline{v}, \sigma)) \qquad \sigma' = \sigma_r \cup \sigma_1\end{array}}{\begin{array}{c}\textsc{act}(\alpha, o_\alpha, \sigma, \{q \mapsto \{\ell | x = e.\mathtt{m}(\overline{e}); s\} :: F\} \uplus p, Rq) \rightarrow \\ \textsc{act}(\alpha, o_\alpha, \sigma', \{q \mapsto \{\ell \mid x = o; s\} :: F\} \uplus p, Rq :: (f, m, \overline{v_r}))\ \textsc{fut}(f, \bot)\end{array}}$$

- Invk-Passive performs a local synchronous method invocation. The method is retrieved and an execution context is created; the thread stack is extended with this execution context. The interrupted execution context is second in the stack; the result of the method execution will replace the hole • when it is computed.

- Return-Local handles the return value of local method invocation. It replaces the hole • in the second entry of the stack by the returned value.

We know that the return statement corresponds here to a local invocation because it is not the only execution context in the stack.

- Return is triggered when a request finishes. It stores the value computed by the request as a future value. Serialisation is necessary to pack the objects referenced by the future value.

- Update updates a future reference with a resolved value. This is performed at any time when a future is referenced and the future value is resolved.

The local rules reflect a classical object oriented language: New-Object and Assign-Field modify the local store, and Invk-Passive and Return-Local affect the local execution context. The other rules, namely Serve, New-Active, Invk-Active Return, and Update reflect how asynchrony, typical to active object languages, is handled in MultiASP. The initial configuration of a MultiASP program consists in evaluating a program $P = \overline{C}\ \{\overline{x\ ;}\ \ s\}$. To this end, the main block is wrapped in a new activity that serves a single request containing the main block. Precisely, the initial MultiASP configuration is: $\textsc{act}(\alpha_0, o, o \mapsto \varnothing, \{q_0 \mapsto \{\overrightarrow{x \mapsto null}|s\}\}, \varnothing)$.

4.3.2 Semantics of MultiASP Request Scheduling

In this subsection, we formalise the advanced mechanisms presented in Chapter 3, that allow the programmer to control the scheduling of requests in multiactive objects. Our main approach is to introduce additional qualifiers on activities and on requests. They are related to how the thread limits have been configured through the multiactive object annotations and API, presented in Section 3.2. Such qualifiers basically reflect the status of the given element at a given time. We also define below the needed auxiliary functions to complement the MultiASP semantics with the request scheduling aspects. But first of all, we extend the syntax of MultiASP so that the thread limit mechanism can be programmatically changed from a *soft limit*, i.e. a thread blocked in a wait-by-necessity is not counted in the thread limit, to a *hard limit*, i.e. all threads are counted in the thread limit. The statements accepted by the MultiASP syntax are thus extended with two new possibilities:

$$s \ ::= \ ... \ | \ \textbf{setLimitSoft} \ | \ \textbf{setLimitHard}$$

Secondly, we extend MultiASP semantics such that the multiactive object group of a request q can be retrieved through an auxiliary function $group(q)$. Additionally, a filtering operator $p\big|_g$ returns the requests from group g among the set of threads p. There can be a thread limit defined for each group of requests. The thread limit of a group g can be retrieved with $\mathcal{L}_g$. The semantics of MultiASP with thread limits is an adaptation of the MultiASP semantics presented in Subsection 4.3.1. In order to indicate the status of each thread, we qualify each of the currently served request as either active or passive. The set of threads p contains then two kinds of served requests: actively served request, denoted with $q_A \mapsto F$, and passively served requests (requests blocked in wait-by-necessity), denoted with $q_P \mapsto F$. The auxiliary function $\text{Active}(p)$ returns the number of actively served requests in p.

Regarding activities, each activity has either a soft limit status written $\text{ACT}(\ldots)_S$ (which is the default for all created activities if not specified otherwise), or a hard limit status written $\text{ACT}(\ldots)_H$. sh is used as a variable ranging over S and H.

Finally, the definition of the status of requests and activities and the definition of auxiliary functions allow us to modify the reduction rules of MultiASP, in order to formalise the threading policies of multiactive objects, as follows:

- We add a rule ACTIVATE-THREAD for activating a thread. Indeed, to activate a thread we need to look at the group of the considered request and check if this group has not reached its thread limit. Note that in the rule, the sh variable indicates that the kind of thread limit is kept unchanged during the reduction.

MultiASP

ACTIVATE-THREAD

$$\frac{Group(q) = g \qquad \text{Active}(p\big|_g) < \mathcal{L}_g}{\text{ACT}(\alpha, o_\alpha, \sigma, \{q_P \mapsto F\} \uplus p, Rq)_{sh} \rightarrow \text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto F\} \uplus p, Rq)_{sh}}$$

- Each rule allowing a thread to progress requires that the request processed by this thread is active. To this end, q is replaced by q_A in all MultiASP reduction rules except for the Serve and Update rules.

- The Serve rule is only triggered if the thread limit of the group of the considered request is not reached, i.e. if: $\text{Active}(p|_{group(q)}) < \mathcal{L}_g$.
- We add two additional rules, Set-Hard-Limit and Set-Soft-Limit, for changing the kind of thread limit of an activity, either to a hard or a soft thread limit.

$$\begin{array}{l}\textsc{Set-Hard-Limit}\\ \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid \textbf{setLimitHard}; s\} :: F\} \uplus p, Rq)_{sh}\\ \qquad \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid s\} :: F\} \uplus p, Rq)_H\end{array} \qquad \text{MultiASP}$$

$$\begin{array}{l}\textsc{Set-Soft-Limit}\\ \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid \textbf{setLimitHard}; s\} :: F\} \uplus p, Rq)_{sh}\\ \qquad \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid s\} :: F\} \uplus p, Rq)_S\end{array} \qquad \text{MultiASP}$$

- If an activity has a soft thread limit status, a wait-by-necessity passivates the current thread. Note that in MultiASP, a wait-by-necessity occurs only in case of method invocation on a future, since field access is only allowed on the current object `this`. We add a rule Invk-Future that encompasses the case where a method invocation is performed on an unresolved future, in which case the thread of the currently processed request becomes passive.

$$\textsc{Invk-Future}\quad \frac{[\![e]\!]_{(\sigma+\ell)} = o \qquad \sigma(o) = f}{\begin{array}{l}\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid x = e.\mathtt{m}(\overline{e}); s\} :: F\} \uplus p, Rq)_S\\ \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q_P \mapsto \{\ell \mid x = e.\mathtt{m}(\overline{e}); s\} :: F\} \uplus p, Rq)_S\end{array}} \qquad \text{MultiASP}$$

To conclude, a few precise modifications on the MultiASP semantics allow us to formalise the threading policies of multiactive objects. Four rules are added, and almost all existing rules are updated, but only with subtle additional information. Indeed, the threading extension of MultiASP is built upon the existing MultiASP semantics and does not modify the core of the language.

4.3.3 Translational Semantics and Restrictions

This section formalises the translation of ABS programs into MultiASP programs, thanks to the semantics of the two languages, presented above (Subsections 4.3.1 and 4.3.2). We show here the translational semantics that formalises the automatic translation provided by the ProActive backend for ABS (Section 4.2). We present afterwards the specific restrictions of the translation.

Translational Semantics

Most of the translation from ABS to MultiASP impacts the statements. The rest of the source structure (classes and methods) is unchanged except the two following:

- We define a new class `COG` in the MultiASP translation. It has methods to store and retrieve local objects, and to execute a method on a local object. The structure of the `COG` is displayed in Listing 4.3. `UUID` is the type of object identifiers. In practice, the body of the *execute* method is slightly more complex since reflection must be used to go from the method name to the method invocation.

MultiASP

```
Class COG {
  UUID freshID()
  UUID register(Object x, UUID id)
  Object retrieve(UUID id)
  Object execute(UUID id, MethodName m, params) {
    w=this.retrieve(id);
    x=w.m(params);
    return x
  }
}
```

Listing 4.3 – The `COG` class. Most method bodies are left out.

- All translated ABS classes are extended with two parameters: a *cog* parameter, storing the `COG` object to which the object belongs, and an *id* parameter,

$$
\begin{array}{llll}
\llbracket x = e!m(\overline{e}) \rrbracket \triangleq & t = e.cog(); id = e.myId(); & \llbracket await\ x? \rrbracket \triangleq & w = x.get() \\
 & x = t.execute(id, m, \overline{e}) & \llbracket await\ g \wedge g' \rrbracket \triangleq & \llbracket await\ g \rrbracket; \llbracket await\ g' \rrbracket
\end{array}
$$

$$
\begin{array}{llll}
\llbracket x = y.get \rrbracket \triangleq & \mathbf{setLimitHard}; & \llbracket x = e.m(\overline{e}) \rrbracket \triangleq & a = e.cog(); b = \texttt{this}.cog(); \\
 & w = y.get(); & & \texttt{if}(a == b) \quad \{x = e.m(\overline{e})\} \\
 & \mathbf{setLimitSoft}; & & \texttt{else} \quad \{t = e.cog(); id = e.myId(); \\
 & x = y & & \qquad x = t.execute(id, m, \overline{e}); \\
 & & & \qquad \mathbf{setLimitHard}; \\
\llbracket x = e \rrbracket \triangleq & x = e & & \qquad w = x.get(); \mathbf{setLimitSoft}\}
\end{array}
$$

$$
\begin{array}{llll}
\llbracket x = \texttt{new local}\ C(\overline{e}) \rrbracket \triangleq & t = \texttt{this}.cog(); & \llbracket x = \texttt{new C}(\overline{e}) \rrbracket \triangleq & newcog = \texttt{newActive COG}(); \\
 & id = t.freshId(); & & id = newcog.freshId(); \\
 & no = \texttt{new C}(\overline{e}, t, id); & & no = \texttt{new C}(\overline{e}, newcog, id); \\
 & z = t.register(no, id); & & z = newcog.register(no, id); \\
 & x = no & & x = no
\end{array}
$$

$$
\begin{array}{ll}
\llbracket await\ g \rrbracket_{\overline{x}} \triangleq \texttt{if}(\neg g) \quad \{ & t = \texttt{this}.cog(); \ id = \texttt{this}.myId(); \\
 & z = t.execute\_condition(id, condition\_g, \overline{x}); w = z.get \ \}
\end{array}
$$

$$
\begin{array}{ll}
\llbracket suspend \rrbracket \triangleq & t = \texttt{this}.cog(); \ id = \texttt{this}.myId(); \\
 & z = t.execute\_condition(id, condition\_True, \overline{x}); w = z.get
\end{array}
$$

Figure 4.11 – Translational semantics from ABS to MultiASP (the left part in $\llbracket\rrbracket$ is ABS code and the right part of $\triangleq$ is MultiASP code).

storing the object's identifier in that COG. Methods *cog*() and *myId*() return those two parameters. A dummy method *get*() that returns `null` is added to each object. For simplicity, we also use this method for primitive values instead of writing specific cases.

The translation of statements and expressions is shown in Figure 4.11 and explained below. Note that trivial rules like $\llbracket s; s' \rrbracket = \llbracket s \rrbracket; \llbracket s' \rrbracket$ are omitted. We also suppose that necessary variables are declared locally in each method.

Object instantiation first gets a fresh identifier from the current COG. Then, the new object is created with the current COG and the identifier[10]. It is stored in a reserved temporary local variable *no*. Finally, the object is referenced in the current COG and stored in x.

Object instantiation in a new COG is similar to object instantiation in the current COG but method invocations on the *newcog* variable are asynchronous

[10] The step in which the COG of the new object is set in ProActive is directly encoded in the object constructor in MultiASP.

remote method calls. The new object is thus copied to the memory space of the remote new COG via the *register* invocation, before being assigned to x.

Await on a future uses the *get()* method, that all translated objects have, in order to trigger the wait-by-necessity mechanism and potentially block the thread if the future is not resolved.

Get on a future sets a hard limit on the current activity, so that no other thread starts, and then restores the soft limit after having waited for the future.

Await on conditions performs several sequential *get()* within an activity in soft limit. Conditional guards are detailed later in this section.

Asynchronous method call retrieves the COG of the object and delegates the call to the *execute* method of the `COG` class, as described in Subsection 4.2.1.

Synchronous local method call distinguishes two cases, like in ABS. Either the call is local and an execution context is pushed in the stack, or the call is remote and, like in the ABS semantics, we perform an asynchronous remote method invocation and immediately wait the associated future within an activity in hard limit.

All other instructions that do not deal with method invocation, future manipulation, or object creation, are kept unchanged. Note that the guard statement is slightly changed: in our case we suppose that it starts by a set of awaited futures and finishes by boolean expressions. Indeed, futures are single-valued assignments, and once they are available they will remain available. Thus, it is safe to check them before, even if, as mentioned in [Häh13], the ABS `await` statement accepts only monotonic guards and conjunctive composition.

The above translation from ABS to MultiASP is only valid under the condition of a precise multiactive object configuration. In the translated MultiASP code, as in ProActive, there exist different multiactive object groups and each group has its own thread limit, as follows:

$$\mathit{group}(\mathit{freshId}) = g_1 \qquad \mathit{group}(\mathit{execute}) = g_2 \qquad \mathit{group}(\mathit{register}) = g_3$$
$$\mathcal{L}_{g_1} = 1 \qquad \mathcal{L}_{g_2} = 1 \qquad \mathcal{L}_{g_3} = \infty$$
$$\forall q, q'.\ (q \neq q' \neq freshId() \wedge (\nexists id. q = register(x, id) \wedge q' = execute(id, m, \overline{e}))) \Rightarrow$$
$$compatible(q, q')$$

Group g_1 encapsulates $freshId$ requests. These requests cannot execute in parallel safely, so g_1 is not self compatible, and can only use one thread at a time. Group g_2 gathers *execute* requests. It is limited to one thread to comply with the threading model of ABS, and the requests are self compatible to enable interleaving. Group g_3 contains $register$ requests that are self compatible and that have an infinite thread limit. Concerning compatibility between groups, they are all compatible except g_3 and g_2: their compatibility is defined dynamically such that an *execute* request and a $register$ request are compatible only if they do not affect the same identifier.

In order to support ABS conditional guards (`await` on conditions), for each guard g, we generate a method *condition_g* that takes as parameters the needed local variables $\overline{x}$[11]. The method body can normally access the fields of the object `this`. A condition evaluation g is defined as follows:

$$\mathit{condition\_g}(\overline{x}) = \texttt{while}(\neg g)\ \mathit{skip}; \texttt{return null}$$

We formally define the ABS `suspend` statement the same as conditions but with a condition that is always true. We define an *execute_condition* method in the `COG` class that generically executes generated condition methods. The *execute_condition* method has its own group with an infinite thread limit because any number of conditions can evaluate in parallel. Thus, we have the following additional information:

$$\mathit{group}(\mathit{execute\_condition}) = g_4 \quad \mathcal{L}_{g_4} = \infty$$

[11] If guards were not monotonic, we would have to check again the condition when the thread that waits is resumed (at this point no other thread can interfere with the object state). In that case, the conjunctive guards are expressed as a set of guards on futures (monotonic by nature) plus a single conditional expression guard (possibly containing conjunction).

Restrictions of the Translation

We define four specific restrictions on the translation. Firstly, MultiASP ensures causal ordering of communications with a small rendez-vous that precedes all asynchronous method calls: the request *is dropped* in the remote request queue *synchronously*. This brief synchronisation does not exist in ABS where requests can arrive in any order. The differences between communication channels in active object languages are discussed in Section 2.5. We will reason on $\overset{\text{A}}{\rightarrow}$, the ABS semantics with rendez-vous communications, where the message sending and reception rule is replaced by the Rendez-vous-Comm rule:

$$\text{ABS}\qquad \frac{\textsc{Rendez-vous-Comm}\\ \text{fresh}(f) \qquad i'\_\beta = [\![e]\!]^{A}_{a+l} \qquad \overline{v} = [\![\overline{e}]\!]^{A}_{a+l} \qquad p'' = \text{bind}(i'\_\beta, f, m, \overline{v}, \texttt{class}(o'))}{\begin{array}{c} ob(i\_\alpha, a, \{l|x = e!m(\overline{e}); s\}, q)\ ob(i'\_\beta, a', p', q') \\ \overset{\text{A}}{\rightarrow} ob(i\_\alpha, a, \{l|x = f; s\}, q)\ ob(i'\_\beta, a', p', q' \cup p'')\ fut(f, \bot) \end{array}}$$

Secondly, in MultiASP, while thread activation can happen in any order, the order in which requests are served is FIFO by default instead of the non-deterministic activation of a thread featured by ABS semantics. In both the Java backend for ABS and the developed ProActive backend, activation and request service are FIFO, although ProActive supports the definition of different policies through multiactive object annotations [3]. Consequently, we only reason on executions that enforce a FIFO policy, i.e. executions that serve requests and restore them in a FIFO order. Thirdly, the proof does not deal with local synchronous method invocations. Considering them would make entangled proofs and would bring no particular insight on the translation, because this part is similar to the Java backend for ABS. Finally, the restriction of Theorem 4.3.5 on future's values not being a future reference is further detailed in Section 4.4.

4.3.4 Formalisation of Equivalence

In this subsection, we present an equivalence relation that defines the conditions under which a MultiASP configuration is considered to be equivalent to an ABS configuration. Secondly, we give the lemmas on which we rely for the proof of correctness of the translation. Finally, we present in details the two theorems that represent the correctness of the translation.

Beforehand, let us define precisely our methodology and give a brief insight into our results. The two presented theorems state under which conditions the operational semantics of the two languages simulates each other (we have one theorem for each direction of the simulation). We achieve the proofs of the theorems in terms of weak simulation in the two cases. The weak simulation is due to the additional steps that we must add to the simulation in order to ensure an eventual equivalence of configurations. However, our results can be considered stronger than the light guarantees given by a weak simulation. Indeed, all the steps that we add in both simulations are deterministic, and none of them modify other activities. Thus, the added steps in the simulations never introduce concurrency. In this case, considering these steps as not observable in a simulation does not jeopardise the theorems. Thanks to this determinism in the added steps, most of the relevant properties that are proven on the source program still remain valid in the translated program. Consequently, the way the two languages simulate each other (in both directions) gives much more guarantees than a weak simulation, even if technically it remains as such.

Equivalence Relation

We define an equivalence relation $\mathcal{R}$ between MultiASP and ABS terms. This equivalence relation aims at proving that any single step of one calculus can be simulated by a sequence of steps in the other. This is similar to the proof in [DP15]. In particular, we use the same observation notion: processes are observed based on remote method invocations. The equivalence relation $\mathcal{R}$ is split into three parts: equivalence of values, equivalence of statements, and equivalence of configurations. In the following, we always refer to the notation Cn for an ABS configuration and to the notation $\underline{Cn}$ for a MultiASP configuration.

Definition 2 (Equivalence of values)**.** $\approx_{\sigma}^{Cn}$ *is an equivalence relation between values (or between a value and a storable), in the context of a MultiASP store σ and of an ABS configuration Cn, such that:*

$$v \approx_\sigma^{Cn} v \qquad\qquad f \approx_\sigma^{Cn} f \qquad\qquad i\_\alpha \approx_\sigma^{Cn} [cog \mapsto \alpha, Id \mapsto i, \overline{x \mapsto v'}]$$

$$\frac{v \approx_\sigma \sigma(o)}{v \approx_\sigma^{Cn} o} \qquad\qquad \frac{fut(f, v') \in Cn \qquad v' \approx_\sigma^{Cn} v}{f \approx_\sigma^{Cn} v}$$

Runtime values in ABS are either (global) object references, future references or primitive values. The equivalence relation $\approx_\sigma^{Cn}$ specifies that two identical values or futures are equivalent if they are the same. Otherwise, an object is characterised by its identifier and its COG name. The two last cases are more interesting and are necessary because of the difference between the future update mechanisms of ABS and MultiASP. First, the equivalence can follow as many local indirections in the store as necessary. Second, the equivalence can also follow future references in ABS, because a future might have been updated transparently in MultiASP while in ABS the explicit future read has not been performed yet.

Definition 3 (Equivalence of statements)**.** *Furthermore, we have an equivalence on statements* $s \approx_{(\sigma+\ell')}^{Cn} s'$ *<u>iff</u>:*

– *either* $[\![s]\!] = s'$
– *or* $s = (x = v; s_1) \ \wedge \ s' = (x = e; [\![s_1]\!])$ *with* $v \approx_\sigma^{Cn} [\![e]\!]_{(\sigma+\ell')}$

In other words, two MultiASP and ABS statements are equivalent if one is the translation of the other. Or, two statements are equivalent if both statements start with an assignment of equivalent values to the same variable, followed by equivalent statements.

Definition 4 (Equivalence of configurations)**.** *Finally, ABS configuration Cn and MultiASP configuration* $\underline{Cn}$ *are equivalent, written* $Cn\ \mathcal{R}\ \underline{Cn}$*, iff the condition shown in Figure 4.12 holds.*

In more details, the equivalence condition of Figure 4.12 globally considers three cases, explained below:

– The first five lines deal with equivalence of COGs. This case compares both activity content and activity requests on the ABS and MultiASP sides:

1 $\exists a.\, \mathit{cog}(\alpha, a) \in \mathit{Cn}\ \ \underline{\underline{\text{iff}}}\ \ \exists o_\alpha, \sigma, p, \mathit{Rq}.\, \mathit{act}(\alpha, o_\alpha, \sigma, p, \mathit{Rq}) \in \underline{\mathit{Cn}}\quad \underline{\text{with}}$

2 $\quad\exists \overline{v}, p', q.\mathit{ob}(i\_\alpha, \overrightarrow{x \mapsto v}, p', q) \in \mathit{Cn}\ \ \underline{\underline{\text{iff}}}\ \ \exists o, \overline{v'}.\, \sigma(o) = [\mathit{cog} \mapsto \alpha, \mathit{myId} \mapsto i, \overrightarrow{x \mapsto v'}]\ \underline{\text{with}}$

3 $\quad\overline{v} \approx_\sigma^{\mathit{Cn}} \overline{v'} \quad \wedge$

4 $$\left(\begin{array}{l} \exists l, s.\, p' = \{l|s\}\ \underline{\underline{\text{iff}}}\ \exists f, i, m, \overline{v''}, \ell', s', \ell'', s''.\, \big((f, \mathit{execute}, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \\ \wedge\ \ell'(\mathtt{this}) = o\big)\ \underline{\text{with}}\ \forall x \in \mathit{dom}(l)\backslash \mathit{destiny}.\, l(x) \approx_\sigma^{\mathit{Cn}} \ell'(x) \wedge l(\mathit{destiny}) = f) \wedge s \approx_{\sigma+\ell'}^{\mathit{Cn}} s' \end{array}\right) \wedge$$

5 $$\forall f. \left(\begin{array}{l} \big(\exists l, s.(\{l|s\} \in q \wedge l(\mathit{destiny}) = f)\ \underline{\underline{\text{iff}}} \\ \exists i, m, \overline{v''}, \ell', s', \ell'', s''.\big(\,((f, \mathit{execute}, i, m, \overline{v''})_P \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \wedge \ell'(\mathtt{this}) = o\big) \\ \vee\ \big((f, \mathit{execute}, i, m, \overline{v''}) \in \mathit{Rq} \wedge o_\alpha.\mathit{retrieve}(i) = o \wedge \mathrm{bind}(o, m, \overline{v''}) = \{\ell'|s'\}\big)\,\big) \\ \quad \underline{\text{with}}\quad (\forall x \in \mathit{dom}(l)\backslash \mathit{destiny}.\, l(x) \approx_\sigma^{\mathit{Cn}} \ell'(x)) \wedge s \approx_{\sigma+\ell'}^{\mathit{Cn}} s'\big) \end{array}\right)$$

6 $\mathit{fut}(f, v) \in \mathit{Cn}\ \ \underline{\underline{\text{iff}}}\ \ \exists v', \sigma.\, (\textsc{fut}(f, v', \sigma) \in \underline{\mathit{Cn}} \wedge \mathrm{Method}(f) = \mathit{execute})\ \underline{\text{with}}\ \ v \approx_\sigma^{\mathit{Cn}} v'$

7 $\mathit{fut}(f, \perp) \in \mathit{Cn}\ \ \underline{\underline{\text{iff}}}\ \ \textsc{fut}(f, \perp) \in \underline{\mathit{Cn}} \wedge \mathrm{Method}(f) = \mathit{execute}$

Figure 4.12 – Condition of equivalence between ABS and MultiASP configurations. We use the following notation: $\exists y.\, P\ \ \underline{\underline{\text{iff}}}\ \ \exists x.\, Q\ \underline{\text{with}}\ R$ means $(\exists y.\, P\ \ \underline{\underline{\text{iff}}}\ \ \exists x.\, Q) \wedge \forall x, y.\, P \wedge Q \Rightarrow R$. This allows R to refer to x and y. Finally, $\mathrm{Method}(f)$ returns the method of the request corresponding to future f.

- To compare activity content (Lines 1-3), we rely on the fact that activities have the same name in ABS and in MultiASP. Each ABS object *ob* must correspond to one equivalent MultiASP object in the equivalent activity α. The object equivalent to *ob* must (i) be in the store, (ii) reference α as its COG, (iii) have i as identifier (because the corresponding ABS identifier is i_α), and (iv) have the other fields equivalent to the ones of *ob*.

- To compare the requests of an activity, we need to compare the tasks that exist in ABS *ob* terms to the tasks that exist in the corresponding MultiASP *act* terms. We consider again two cases:

  1. Either the task is active (Line 4), and the single active task of *ob* in ABS (in p') must have exactly one equivalent active task in MultiASP (in p). In MultiASP, this task must have two elements in the current stack[12]: the call to the COG (the *execute* call) and the stacked call that redirected to the targeted object o, where o is equivalent to *ob*. In addition, values of local variables must be equivalent, except *destiny* that must correspond to the future of the MultiASP request. Finally, the current

[12] In the proof, we only deal with asynchronous method calls for lightening it (the synchronous case is the composition of an asynchronous method call and of the `get` statement). In this case, we know that the stack never contains more than two elements.

thread of the two tasks must be equivalent according to the equivalence on statement.

2. Otherwise the task is inactive (Line 5), and two cases are possible. Either the task has already started and has been interrupted: it is passive. In this case, the comparison is similar to the active task comparison above. Or the task has not started yet: there must be a corresponding task in the request queue Rq of the MultiASP active object α, and we check that (i) the future is equivalent, (ii) the invoked object o is equivalent to ob, and (iii) the method body retrieved by the bind predicate is equivalent to the ABS task.

– Line 6 deals with the equivalence of resolved futures. A future's value in MultiASP refers to its local store. Two resolved futures are equivalent if their values are equivalent. In MultiASP, only futures from *execute* method calls are considered, because they represent the applicative method calls.
– Line 7 deals with the equivalence of unresolved futures. In that case, the two futures must be unresolved to ensure equivalence.

Overall, the association of the equivalence of values (Definition 2), the equivalence of statements (Definition 3), and the equivalence of ABS and MultiASP configurations (Definition 4), forms the global equivalence relation $\mathcal{R}$. We rely on equivalence $\mathcal{R}$ to prove that our systematic translation of ABS programs into MultiASP programs is correct.

Preliminary Lemmas

Firstly, we define some notations and naming conventions. In order to establish a proper link between ABS and MultiASP semantics, we identify activity names of MultiASP with COG names (ranged over by α, β). Second, object locations are valid only locally in MultiASP and globally in ABS, their equivalent global reference is the pair (activity name, identifier). We suppose that each object name in ABS is of the form i_α where α is the name of the COG where the object is created, and where i is unique locally in activity α. The semantics then allows us to choose the MultiASP identifier Id and i such that they are equal. In general, terms like Cn range over ABS configurations and terms like $\underline{Cn}$ range over MultiASP configurations.

Secondly, we define some lemmas that help us establishing the equivalence in the simulation. The proofs of the lemmas we present below can be found in Appendix A.1.

Lemma 4.3.1 (Correspondence of activated objects). *In an ABS configuration, if an object ob has an active request that is not* **`idle`**, *then there exists a* COG *in which ob is the current active object.*

$$ob(i\_\alpha, \overline{x \mapsto v}, p, q) \in Cn \wedge p \neq \texttt{idle} \Rightarrow cog(\alpha, i\_\alpha) \in Cn$$

Lemma 4.3.2 (Equivalence of values).

$$v \approx^{Cn}_{\sigma} v' \Rightarrow v \approx^{Cn}_{\sigma} [\![v']\!]$$

Lemma 4.3.3 (Equivalence of evaluation functions). *Let Cn be an ABS configuration and suppose $Cn\ \mathcal{R}\ \underline{Cn}$. Let $ob(o\_\alpha, a, \{l|s\}, q) \in Cn$. By definition of $\mathcal{R}$, there exists a single activity $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$, with $\sigma(o) = [cog \mapsto \alpha, myId \mapsto i, a']$ and $(f, execute, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \wedge \ell'(\texttt{this}) = o$. For any ABS expression e we have:*

$$[\![e]\!]^{A}_{a+l} \approx^{Cn}_{\sigma} [\![e]\!]_{\sigma+\ell'}$$

Lastly, the serialisation mechanism, together with the renaming of local references, are crucial points of difference between ABS and MultiASP. We need Lemma 4.3.4 to deal with the serialisation and the renaming aspects, that are essential in a distributed context.

Lemma 4.3.4 (Equivalence after serialisation and renaming).

$$\begin{aligned}
&v \approx^{Cn}_{\sigma} v' \wedge \sigma' = serialise(\sigma, v') \Rightarrow v \approx^{Cn}_{\sigma'} v' \\
&\overline{v} \approx^{Cn}_{\sigma} \overline{v'} \wedge (\overline{v''}, \sigma') = rename_{\sigma}(\overline{v'}, \sigma) \Rightarrow \overline{v} \approx^{Cn}_{\sigma'} \overline{v''}
\end{aligned}$$

Besides the introduced lemmas, in order to deal with the proof of Theorem 4.3.5, we rely on the fact that all ABS objects are locally registered in their COG, and we define an invariant for that:

Invariant Reg. *For every activity α such that o_α is the location of the active object of activity α in its store σ_α, if the current task consists in an invocation to $o_\alpha.retrieve(i)$, then this invocation always succeeds, because the object has been registered first. The invocation returns some object o' such that $i\_\alpha \approx_\sigma^{Cn} o'$.*

In conclusion, we use the correspondence of activated objects (Lemma 4.3.1), the equivalence of values (Lemma 4.3.2), the equivalence of evaluation functions (Lemma 4.3.3), the holding of equivalence after serialisation and renaming (Lemma 4.3.4), and the invariant on the accessibility of an object (**Invariant Reg**) in order to prove Theorem 4.3.5 and Theorem 4.3.6, that are defined below.

Theorems

In order to prove the correctness of the translation from ABS to MultiASP, we prove two theorems. The two theorems exactly specify under which conditions each semantics simulates the other.

Theorem 4.3.5 (ABS to MultiASP)**.** *The translation simulates all ABS executions with FIFO policy and rendez-vous communications, provided that no future value is a future reference.*

$$Cn_0 \overset{\mathrm{A}}{\to}^* Cn \text{ with a FIFO policy} \wedge \nexists f, f'. fut(f, f') \in Cn \;\Rightarrow\; \exists \underline{Cn}. \underline{Cn_0} \to^* \underline{Cn} \wedge Cn \mathcal{R} \underline{Cn}$$

Theorem 4.3.6 (MultiASP to ABS)**.** *Any reduction of the MultiASP translation corresponds to a valid ABS execution.*

$$\underline{Cn_0} \to^* \underline{Cn} \Rightarrow \exists Cn. Cn_0 \overset{\mathrm{A}}{\to}^* Cn \wedge Cn \mathcal{R} \underline{Cn}$$

The theorems are proven in Appendix A.2. We focus the proof on the less trivial and most informative theorem: the proof that all ABS executions with FIFO policy and rendez-vous based communications are simulated by MultiASP executions. This proof is completely realised in the sense that it exhaustively considers all reduction rules of the ABS semantics. For the second proof, stating that a MultiASP translation corresponds to a valid ABS execution, we expose the differences and similarities compared to the first proof and we conclude accordingly. Globally, we are able to simulate the program execution in the other language. Additional steps are introduced in the MultiASP translation, which we refer to as *silent actions*, because they are always local and they never introduce concurrency.

This is precisely why the equivalence relation $\mathcal{R}$ has to be adapted for the proof of Theorem 4.3.6. The next section summarises our methodology for the proofs of the two theorems and highlight our main results.

4.4 Translation Feedback and Insights

This chapter presented an approach for translating the paradigms of active object languages with cooperative scheduling into an efficient setting of multiactive objects. We have first introduced the ProActive backend for ABS, a fully implemented tool that automatically translates an ABS program into a ProActive program, and allows it to be executed in a distributed environment. We have presented the crucial points of the translation, related to the difference of active object paradigms, that were introduced in Section 2.2. These differences impact the object instantiation and addressing, the asynchronous remote method invocations, and the scheduling of active object requests. We have formalised the translation using MultiASP, the calculus of ProActive. We have extended MultiASP so that it could take into account the advanced scheduling mechanisms introduced in Chapter 3. By using MultiASP and the semantics of ABS, we could establish the translational semantics, and we could prove the correctness of the translation, through the proof of two well-defined theorems, and under specific restrictions. These restrictions are however not major because the translation still cope with them and produce correct results. Moreover, the restrictions can be overcome with simple workarounds, although we do not apply these in order to be able to compare better the languages.

The most informative result of this work rises from the proofs and from the establishment of the equivalence of execution between the two active object languages. Because we used the simulation technique to formalise the translation, we had to decide on which actions would be observable in the simulation. In our case, we show in the proof that ABS request sending is simulated exactly in MultiASP, and conversely. This is also true for method return, object creation, and field assignment. The most striking example of an observable reduction in ABS that is not observable in MultiASP is the update of a future with its computed value. Indeed, the transparency of futures and of future updates create an intrinsic difference between the two programming languages. This is why, in the first proven theorem, we exclude the possibility to have a future's value being a future in the ABS configuration. Other ABS reductions can be observed in a MultiASP translation, but not exactly. For example, the local assignments are not exactly observable because of the existence of temporary variables in the MultiASP translation. Consequently, assignments to ABS local variables can be exactly observed in MultiASP, but the reverse is not ensured. Also, asynchronous method invocations are only observable the applicative ABS requests, that can be filtered in the MultiASP translation by looking at the name of the requests.

Identifying the differences of observability between active object languages is a major result: it gives a significant insight on the design of programming languages. The reason why we could define this observational difference is because we chose a faithful translational approach that matches the elements of ABS and MultiASP configurations in a one-to-one way, as much as possible. Indeed, ABS requests correspond to MultiASP requests, ABS objects correspond to MultiASP objects, and ABS futures correspond to MultiASP futures. Then, the divergent notions could be spotted more easily thanks to our faithful translation.

To conclude, we summarise in five key points below the highlights of this work on the translation and on the proofs. In particular, we summarise the principles of the equivalence between MultiASP and ABS configurations and the specific differences between the two active object languages, and we contextualise the restrictions of the proof.

Communication and ordering of request service. The semantics of ABS relies on completely asynchronous communication channels while MultiASP ensures causal ordering of requests (see discussion about communication channels in Section 2.5). The equivalence can only be valid for the specific ABS reductions that preserve causal ordering of requests. Also, MultiASP serves requests in FIFO order, so similarly we apply a FIFO service policy of ABS requests. These restrictions already exist in the Java backend for ABS. And here, we notice that these differences are more related to request scheduling policies and to communication channels than to the nature of the two active object languages.

Shallow translation. ABS requests, COGs and futures respectively match MultiASP requests, active objects and futures. Likewise, in the translation for each ABS object there exist several copies of this object in MultiASP. All copies share the same COG and the same identifier, but only one of these copies (the one that is hosted in the right COG/activity) is equivalent to the ABS object.

Futures. Because of the difference between the future update mechanism of ABS and the one of MultiASP, the equivalence relation between ABS and MultiASP configurations can follow as many local future indirections in the store as necessary. Firstly, this means that a variable holding a pointer to a future object in MultiASP is equivalent to the same variable holding directly the future reference in ABS. Secondly, it means that the equivalence can follow future references in ABS. This is because a future might have been updated transparently in MultiASP while in ABS, the explicit future read has not been performed yet.

Equating MultiASP and ABS configurations. As mentioned before, a crucial part of the proof of correctness consists in stating whether an ABS and a MultiASP configuration are considered equivalent. The principles of this equivalence are summarised hereafter. The objects are identified by their identifier and their COG name, and the equivalence can follow futures. This has already been detailed in the two previous paragraphs. The equivalence between requests distinguishes two cases. First, for active tasks, there is a single active task per COG in ABS and it must correspond to the single active thread serving an *execute* request in MultiASP. The second element in the stack of method calls corresponds to the invoked request. Second, inactive tasks in ABS correspond either to passive requests being currently interrupted or to requests that have not been served yet in MultiASP. For each task, the equivalence of executed statements, of local variables, and of the corresponding future is checked.

Observational equivalence. The precise formulation of Theorem 4.3.5 and Theorem 4.3.6 shows that the ABS behaviour is faithfully simulated by our translation and conversely. This is proven by adequately choosing the observable and not observable actions in the weak simulation (we call the latter silent actions). For example, in ABS the configurations (i) $\mathit{fut}(f, f')$ $\mathit{fut}(f', \bot)$ and the configuration (ii) $\mathit{fut}(f, \bot)$ are observationally different, whereas in MultiASP they are not. Thus, transparency of futures and of future updates create an intrinsic an unavoidable difference between the two active object languages. However, this is not a major restriction on expressiveness because it is still possible to have a wrapper for futures values: a future value that is an object containing a future. Designing such an encoding could have been possible, but it would have broken the one-to-one mapping between ABS and MultiASP futures. We chose to preserve the shallow translation instead.

A particular note can also be made on distributed future updates. In the simulation of MultiASP in ABS (Theorem 4.3.6), one issue is that the distributed future update mechanism of MultiASP cannot be strictly faithfully represented in ABS, although workarounds can be used in order to still ensure equivalence. The problem arises when futures are transmitted between activities, for example when a future is a parameter of a request sent to another activity. In this case in MultiASP, two proxies for the same future will exist in the store of the two activities, whereas only one centralised future exists in ABS. This situation can create intermediate states in MultiASP where the future value is available but not completely propagated to all the locations where the future has been transmitted. This can create behaviours that are not possible to reach in ABS, because a future update is atomically made available to all activities referencing the future. Consequently, enforcing an atomic future update in MultiASP would directly solve the issue. However, this is not a realistic setting in a distributed context. In MultiASP, we choose to copy futures and to propagate their value when they are available in an asynchronous manner. This is a design that is adapted to distribution. If a centralised representation of a future should be maintained, the performance of distributed applications would be too poor because first, this would bring additional communications to access the future's value and second, a future would represent a potential bottleneck. The solution we advocate for handling the difference in the future update mechanism of the two languages is to rely on forward-based future updates. In this approach, the value of a future is transmitted from activity to activity in a chaining manner, as many times as the future was transmitted in the first place. If we extend the restriction of having FIFO communication channels to the transmission of future values (instead of just having this restriction for request sending), then we extend the causal ordering of messages so that it also applies to future updates. This prevents any observable inconsistency in the future updates. This solution also ensures a correct behaviour with respect to the possible behaviours in ABS, because any operation made on a resolved future would arrive after the update operation. Still, this solution avoids blocking the original activity until the future's value is propagated to all the locations where the future was transmitted. Thus, it seems a reasonable and relevant solution in a distributed context.

Overall, our translational semantics fully respects the ABS semantics and simulates exactly all ABS executions that comply to the aforementioned restrictions. The restrictions are quite light, since either they already exist in the Java backend for ABS, or they can easily be coped with.

* * *

A similar work to this one [Ser+16] is being conducted and aims at simulating the ABS semantics with the paradigms of Java 8, by implementing a lightweight thread continuation mechanism (representative of cooperative scheduling). This work makes a particular focus on efficiency, as opposed to the seminal Java backend for ABS. Preliminary results [1] showed that the Java 8 backend for ABS can scale much better than the existing Java backend, in number of COGs in local execution environments, i.e. in number of threads collocated on the same machine.

Another backend for ABS, the Haskell backend [BB16] focuses like our work on a distributed translation of ABS programs. As opposed to Java-based backends, lightweight thread continuations are natively available in Haskell, which makes the Haskell backend for ABS very efficient even with a high degree of local parallelism: much more COGs can be hosted on the same machine than with programs generated by the ProActive backend. In this work, the ABS compiler is extended to integrate the notion of deployment components within the ABS language. The Haskell backend for ABS also has support for garbage collection of distributed objects, built on top of the Haskell garbage collector. In ProActive, activation of distributed garbage collection is optional, but it less crucial since all passive objects are normally garbage collected by the garbage collector of the JVM. Nevertheless, to the best of our knowledge, the ProActive backend for ABS is the only backend that generates a distributed and executable code that is formally proven to be correct with respect to the ABS semantics.

In general, the effort to port our results to other active object languages depends on the application domain they are the closest to: an active object language that is close to ProActive will certainly be easier to translate. The active objects languages that feature the same language constructs as ABS also are very easy to adapt for the translation. Typically, because of their similarities with ABS, and because we base the translation on the different aspects of the language, our work can be straightforwardly adapted to any active object language featuring cooperative scheduling. For example, adapting our work to Creol or JCoBox raises no difficulty, and porting our results on AmbientTalk only requires minor design adaptation. Similarly, the whole part of Encore that regards objects and thread scheduling is straightforwardly translatable with our approach. Overall, we have found that active object abstractions are most often a special instance of multiactive objects. This already proves the expressiveness of this programming model.

All active object languages share the same basis but show them under different abstractions. The intrinsic differences between active object-based programming languages define what is possible and not possible to do with the language. The range of possibilities is often defined at design time, and is hardly re-adaptable once a choice has been made. This is why this work should help the designers of active object languages, and more generally, the designers of concurrent programming constructs and abstractions. Indeed, we have given in this chapter all the means to design and implement efficient distributed active object languages, dominated by three aspects that are object addressing, request scheduling, and synchronisation.

Chapter 5

Execution Support for Multiactive Objects

Contents

Beyond the adequacy of an active object programming language with the expectation of the programmer in terms of functionalities (language constructs, supported execution platforms), what attracts and retains the programmer is the quality of the tool set that is built around the programming language, that facilitates the development and the execution of applications. This is especially true for the languages and frameworks that offer distributed execution and massive parallelism: abstract away the complexity of an application deployment and help with the debugging of concurrent events. In this chapter, we focus on non functional aspects of the multiactive object programming model. Non functional aspects represent the parts of an application that are not relevant to its business logic, but that help the application to reach its objectives. We start by introducing how non functional aspects of a distributed application are handled by the ProActive library. Then, we present a debugger tool that helps the programmer spotting what happens in parallel in an application that is built with multiactive objects. The advantage of this tool is that it is well integrated with multiactive object notions. Finally, we take interest in the reliable execution of distributed multiactive objects, by adapting a fault tolerance protocol, and by giving directions for future work.

5.1 Context: ProActive Technical Services

The main purpose of the ProActive library is to execute on distributed infrastructures such as clusters, grids and clouds. As such, setting a suitable environment for the execution of a distributed application is a costly operation. In order to ease the deployment phase, ProActive provides a model that is based on deployment descriptors. The objective of the deployment model of ProActive is to decouple the business logic of the application from the infrastrcuture that supports it. This way, deployment descriptors can be modified at will without requiring the modification, the re-compilation and the re-deployment of the application. Along with the deployment model, deployment descriptors can be attached additional capabilities that regard the non functional aspects of applications, especially fault tolerance, communication security, load balancing, logging. The concept of defining non functional requirements for a particular deployment is known in ProActive as a *technical service* [CCD07], and is part of the Grid Component Model of ProActive [Bau+09]. The general objective of technical services is to increase the quality of service of distributed ProActive applications.

In practice, ProActive deployment and technical services appear in configuration files (XML files). The first step of the configuration of a deployment is to define the *virtual nodes* that are involved in the distributed execution. The concept of virtual nodes abstracts the physical machines such that they can be generically accessed in the ProActive program, that is without their physical location being solved in the program. Virtual nodes define a capacity on the number of JVMs/active objects they can host. The concept of virtual nodes also enables a clusterised configuration of distributed entities. More precisely, when technical services are attached to virtual nodes, they apply to all active objects that are deployed on them. An example of configuration of a virtual node with two technical services is displayed on Listing 5.1.

GCM/ProActive

```
<virtualNode id='node_identifier' capacity='1'>
  <technicalServices>
    <class name='TechnicalServiceOneClass'>
      <property name='first_name_TS1' value='first_value_TS1' />
      <property name='second_name_TS1' value='second_value_TS1' />
    </class>
    <class name='TechnicalServiceTwoClass'>
      <property name='name_TS2' value='value_TS2' />
    </class>
  </technicalServices>
</virtualNode>
```

Listing 5.1 – Configuration of a virtual node with two technical services.

The configuration file gives the list of technical services that apply to all the active objects that are deployed on the virtual node. The virtual node can be referenced in the ProActive program through the given identifier (here `node_identifier`). The ProActive class corresponding to the technical service must be mentioned, as well as all the parameters that are needed by the technical service, using key-value pairs. Then, the behaviour of the technical service must be programmed in the corresponding ProActive class, where the given parameters and their value can be accessed. Non functional aspects have been engineered in ProActive such that new technical services can be easily added to the library.

Another aspect of ProActive is to support the specification of non functional requirements for componentised applications [Hen+14]. Such non functional aspects are related to the application lifecycle and they are handled by non functional components. However, non functional components involve concerns that are different from the ones of technical services. They apply at the application level, like the reconfiguration of the application, or its replication, whereas technical services deal with low-level aspects: they do not require the adaptation of the application, but of the middleware instead.

In the next two sections, we present two non functional aspects that we have developed as technical services: they can be added to an application deployment, and parametrised without requiring the re-compilation of the ProActive application.

5.2 A Debugger for Multiactive Objects

The multiactive object programming model is designed such that there is a minimum specification to be written by the programmer. The minimal specification of a multiactive object-based application is the compatibility between the requests. Then, the default values for the scheduling of requests suits to most of applications and give an acceptable performance. However, multiactive objects are also very good for programming systems that need a precise coordination of entities or massive parallelism. For these advanced cases, the default configuration needs to be tuned and to be specifically adapted to the use case in order to hit a high performance. In this context, using advanced multiactive object settings is a difficult task. Indeed, the multiactive object programming model is highly concurrent: multiactive objects executes concurrently and also have intra-concurrency. Even if safety is guaranteed to the application by design (if the model is correctly used), reasoning clearly about the obtained performance is not trivial: listing the set of events that can happen concurrently is already beyond human brain with a few parallel entities.

The objective of the work that is presented in this section is to provide a tool to understand complex multiactive object executions and help in debugging them, in terms of concurrent events. To this end, relevant information is first extracted from the execution of a multiactive object-based application via logging. Second, a debugger tool reads this information and displays the execution of the multiactive objects with notions that stick to the programming model. Especially, in multiactive object executions the flow of concurrent events can create an awkward situation, so we put a particular effort in representing time in the debugger tool with accurate time lines. Our debugger provides a post-mortem feedback: the execution is visible when the application is terminated. However, we give the means to adapt it to a real time monitoring easily and efficiently. This work has been

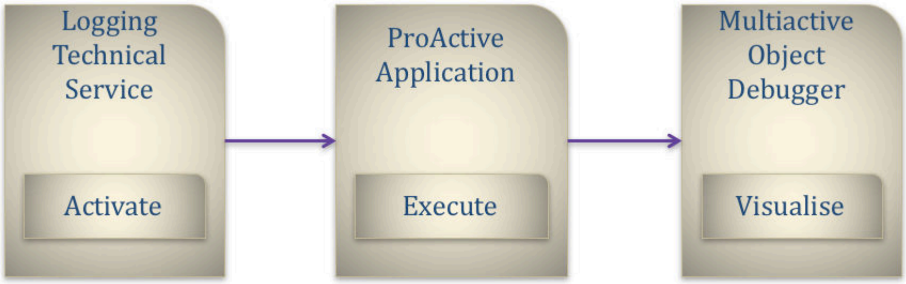


Figure 5.1 – Usage flow of the debugger for multiactive objects.

conducted with Pavlo Khvorostov, who has worked on the implementation of the tool.

5.2.1 Visualisation of Multiactive Object Notions

To develop the multiactive object debugger, we have adapted a part of the ProActive library in order to log and to collect the information needed by the debugger. The basic information that needs to be logged about a multiactive object execution relates to the lifecycle of requests: when a request is sent and from which activity, when a request is received, served, paused, resumed, and finished. Once the required logs are produced by the library, they can be aggregated and interpreted by our debugger tool. Figure 5.1 shows a typical usage of the debugger tool. First, the programmer activates the technical service for logging debug information, by specifying it along with its application deployment. Second, the programmer executes its application, like he would usually do. Then, the programmer uses the debugger tool to view the application's execution, by giving as an input the log files that have been produced during execution.

The ProActive library is extended to produce three kinds of logs required by the debugger tool, as shown in Figure 5.2. Active object logs are meant to trace threads and associate them to a multiactive object. Request logs save the needed information to infer the complete lifecycle of a request. Finally, compatibility logs involve the information that is related to multiactive object notions. We introduce a new technical service in ProActive, whose purpose is to manage the logging of information.

```
<technicalServices>
```

GCM/ProActive

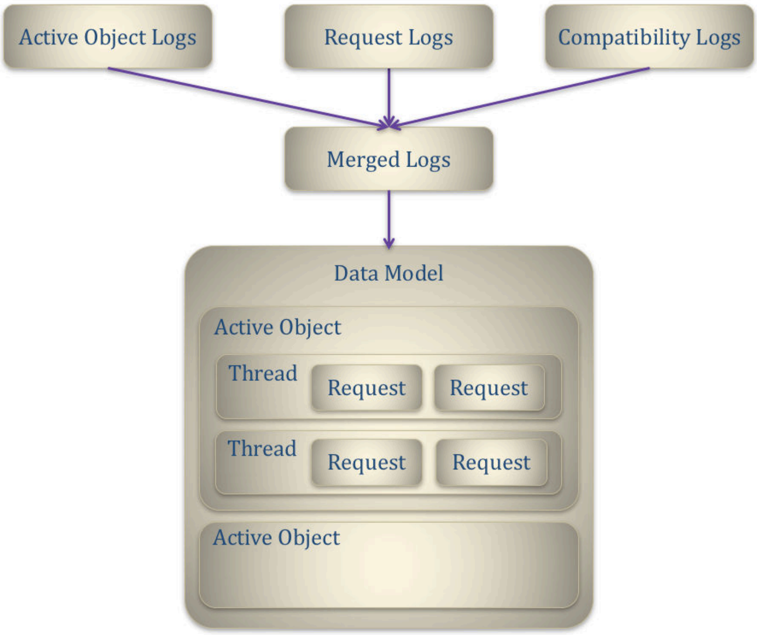


Figure 5.2 – Production and gathering of multiactive object information.

```
  <class name='LoggingTechnicalService'>
    <property name='url_to_log_file' value='URL' />
    <property name='is_enabled' value='true' />
  </class>
</technicalServices>
```

Listing 5.2 – Specification of the logging technical service.

An example of usage of the logging technical service is shown in Listing 5.2. This technical service accepts two parameters: whether the logging mechanism is activated, and the location of the log files in the local file system of the node. The `LoggingTechnicalService` class, whose code is displayed in Listing 5.3, shows the code of the technical service on the library side. In particular, it applies the location of log files as a property of the virtual node on which the active object is deployed, and it sets a boolean value that will activate or deactivate the logging mechanism at different parts of the ProActive library.

ProActive

```
public class LoggerTechnicalService implements TechnicalService {

  private static final long serialVersionUID = 1L;
  public static final String IS_ENABLED = "is_enabled";
```

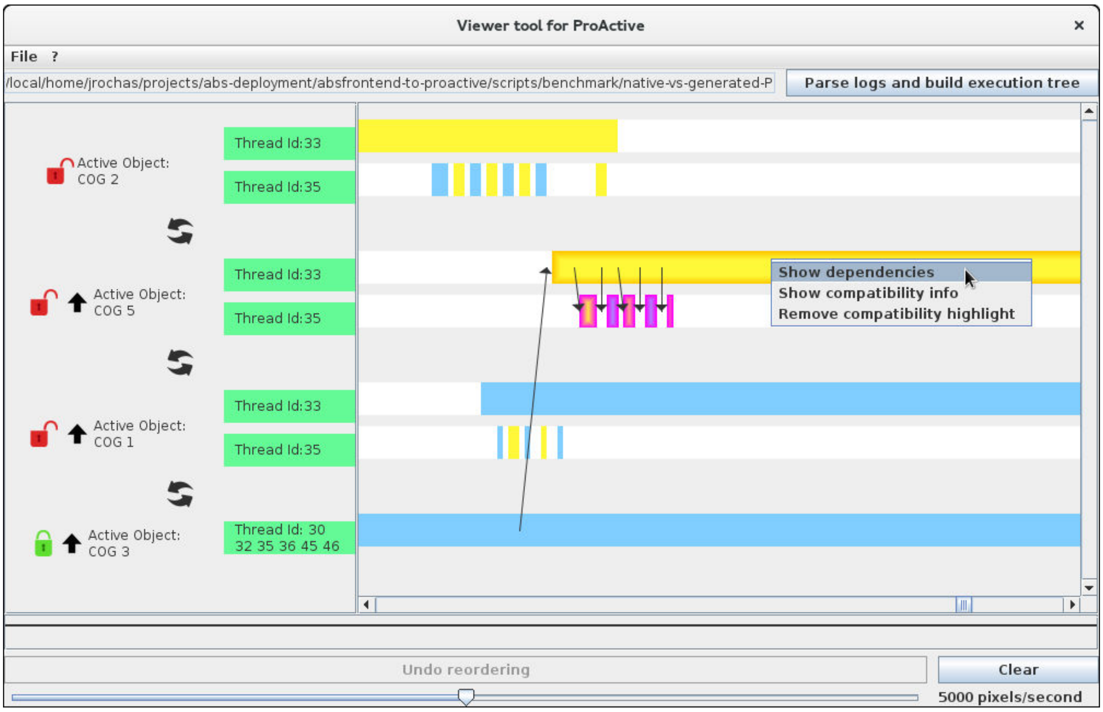
```
 public static final String URL_TO_LOG_FOLDER = "url_to_log_file";
 private boolean isEnabled;
 private String logUrl;

  @Override
  public void init(Map<String, String> argValues) {
      this.isEnabled = Boolean.parseBoolean(argValues.get(IS_ENABLED));
      this.logUrl = argValues.get(URL_TO_LOG_FOLDER);
  }

  @Override
  public void apply(Node node) {
      try {
          if (isEnabled) {
              node.setProperty(IS_ENABLED, Boolean.toString(isEnabled));
              node.setProperty(URL_TO_LOG_FOLDER, logUrl);
          }
      } catch (ProActiveException e) {
          e.printStackTrace();
      }
  }
}
```

Listing 5.3 – Technical service class for logging.

The first step of the debugger tool is to ask the user to select a location where the log files can be loaded. Then, the debugger tool builds a data model from the logs, in order to link efficiently the different information. This step is likened to an in-memory loading of serialised data. We implement the loading phase such that the log files are read in parallel. This is possible because the order in which the file are read is not significant. Depending on the number of events that are involved in the application, treating the logs in parallel is useful to avoid some latency when using debugger tool. Figure 5.2 shows the main components of the data model of the debugger tool: multiactive objects are displayed sequentially and, within a multiactive object, its threads are also displayed sequentially. As requests execute on threads, a thread is represented by a sequence of requests. A screenshot of the debugger tool is displayed on Figure 5.3; it shows the main frame of the tool. The threads of a multiactive object (labelled in green) alternate yellow and blue

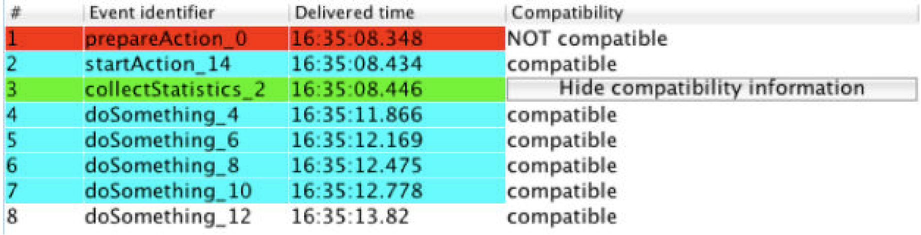


/local/home/jrochas/projects/abs-deployment/absfrontend-to-proactive/scripts/benchmark/native-vs-generated-P

Figure 5.3 – The main frame of the debugger tool.

sections to distinguish two requests that are next to each other. The idle time of a thread is shown in white. In addition to display the components of the data model, Figure 5.3 shows how the debugger tool represents the communications between multiactive objects: an arrow is drawn from the thread that created a request to the thread that processed the request (on the same or on different multiactive objects). As every event is timestamped, the way the event is displayed is representative of the duration of the event. A mouseover on a request displays the name of the request, the identity of the sender, and the timestamp of its reception.

Besides displaying execution information, the debugger tool offers controls in order to re-organise the components of the frame: active objects can be swapped or moved to the top thanks to arrow-shaped controls. The threads of a multiactive object can also be packed on the same time line in order to save vertical space, thanks to the red/green lock, as can be seen with the last active object of Figure 5.3. In this case, only the idle/busy time of the multiactive object can be distinguished. Our implementation of the debugger tool also offer various customisation possibilities such as zoom, scroll, undo, clear events, focus on a selected



#	Event identifier	Delivered time	Compatibility
1	prepareAction_0	16:35:08.348	NOT compatible
2	startAction_14	16:35:08.434	compatible
3	collectStatistics_2	16:35:08.446	Hide compatibility information
4	doSomething_4	16:35:11.866	compatible
5	doSomething_6	16:35:12.169	compatible
6	doSomething_8	16:35:12.475	compatible
7	doSomething_10	16:35:12.778	compatible
8	doSomething_12	16:35:13.82	compatible

Figure 5.4 – Listing of requests for a given multiactive object and compatibility information.

request, and automatic re-ordering.

The added value of the multiative object debugger is to view the information that is related to multiactive object notions. In particular, we are interested in the compatibility of the requests. A selected request is highlighted with a border. On demand of the user, the debugger tool can highlight the compatibility of the request with respect to others. The full information about a request can also be displayed in a pop-up frame, as shown on Figure 5.4 (taken from [Khv15]). This precise listing of requests is meant to focus on a particular timestamp: the requests that are being executed at this timestamp are displayed in green; the requests that are in the queue are in blue; the completed requests are in red; finally, the requests that have been received after the focused timestamp are in white. The pop-up frame also allows the programmer to focus on a particular request in order to show its compatibility with respect to the other requests; this is another view provided by the tool for the same notion shown in the main frame. Since request compatibilities drive request parallelism, one can detect pretty easily the part of the multiactive object configuration that led to an unwanted behaviour, and thus can directly apply the modifications to the configuration in the program.

Since the multiactive object debugger aims at detecting the applications that do not have an adapted multiactive object setting, it provides a feedback on erroneous loading of the logs. An error message is displayed to the user, and the path to the file which caused the error is given. Three kinds of error are rendered to the user. The 'wrong file format' error is shown when the file does not satisfy the format of the logs. The 'wrecked file format' error indicates that the format of the file is correct, but the information that lies in it is not sound, which may happen in case of a partial failure of the application. Finally, the 'request never ends' error

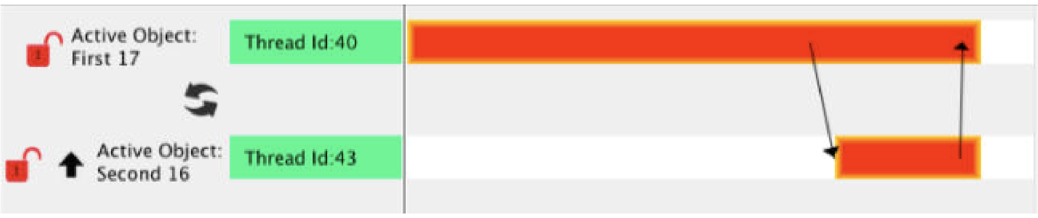


Figure 5.5 – Representation of a deadlock by the debugger tool for multiactive objects.

is precise enough to detect the origin of the error; we show an example of this error in the next subsection. By default, all ProActive applications produce the log files that are suitable for the debugger, in the default temporary location. If the logs cannot be produced for any reason, then the logging mechanism turns off automatically and silently. Finally, the logging technical service must be used in order to explicitly turn off the logging mechanism.

5.2.2 Illustrative Use Cases

In this subsection, we show the usefulness of the multiactive object debugger by reasoning on two common problems of coordinated and concurrent systems: the case of a deadlock and the case of a data race. In active objects, a deadlock is generally caused by a circular dependency in nested requests. With multiactive objects, a deadlock also involves reentrant requests, but it is triggered by a special constraint on the multiactive object: lack of compatibility or lack of threads. We show in Figure 5.5 an example of a deadlocked multiactive object execution. For this execution, the debugger produces the ‘request never ends’ error, that is displayed for deadlocked executions, but that also can be displayed for other reasons. More importantly, the debugger displays in red the requests that did not end, which in our case helps in spotting the reason for the deadlock. In Figure 5.5, we displayed the communications that occurred between the incriminated requests. There is indeed a circular communication pattern involved, where active object `First` calls a method of active object `Second`, and where `Second` makes a request back to `First`, that is not executed. Here, two cases are possible: if the request received by `First` is not compatible with the request that executes, then it will never execute. Otherwise, here the deadlock means that there is a lack of threads:

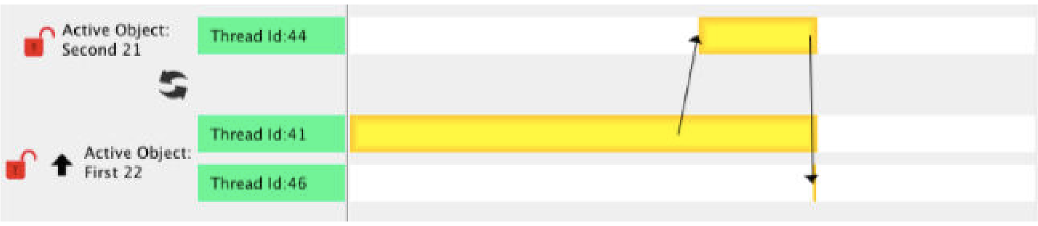


Figure 5.6 – Fix of a deadlock.

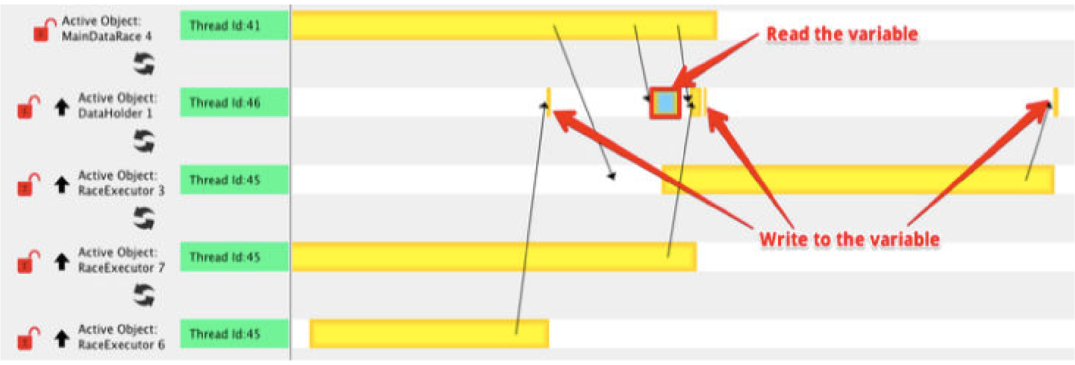


Figure 5.7 – Detection of a data race thanks to the debugger tool.

all threads are busy, and no more can be created. Thus, we can conclude that the soft thread limit is not activated, because in this case another thread would compensate the thread that is in wait-by-necessity.

In order to fix a deadlock in a multiactive object execution, one needs to make sure that nested requests that come back to a sender are compatible with each other, and that the thread limit of the multiactive object complies with nested request execution: either the thread limit is at least as big as the maximum number of callbacks to the multiactive object, or the kind of thread limit must be a soft thread limit to handle those cases[1]. Figure 5.6 shows a fixed version of the deadlocked execution, where no more errors are produced. For this execution, the thread limit has been kept to one thread, but the kind of thread limit has been changed to a soft thread limit for multiactive object `First`.

Another common situation that is source of unwanted behaviours in concurrent systems is the race conditions. In this situation, the state of the system basically

[1] Alternatively, the kind of thread limit can remain strict (hard limit), but then the programmer must set the reentrance parameter of the multiactive object, in order to process reentrant requests on the same thread.

depends on the uncontrollable order of events. One possible setting in which a race condition can occur is when a single variable is modified and read by many threads. We have created the particular setting in which an active object holds a variable that can be accessed (written and read) by many other active objects. Since it is an active object that holds this variable, there cannot exist a data race on this variable. However, the order in which the reads and writes are performed depends on concurrent request sendings. We have analysed the situation with our tool, as shown in Figure 5.7. The red arrows are added on the picture in order to show our interpretation of the execution; they do not belong to the graphical frame of the debugger tool. The `MainDataRace` object is responsible for orchestrating the application. The `DataHolder` object contains the single variable that can be accessed concurrently by three `RaceExecutor` objects. In the particular execution that is shown in Figure 5.7, we can see that the reading of the variable happens in between the first and second writing of the variable. Consequently the value that is read cannot reflect the last written value. Knowing what sequence of actions is performed by an active object is very informative for the programmer. Note that since the debugger gives a faithful representation of time, the debugger tool can also help in spotting race conditions that happen within a multiactive object.

To conclude, the debugger tool that we have developed is completely integrated with the notions of multiactive objects. Going back and forth from the application to the debugger is simpler than doing so with existing debugging tools. For example, YourKit [Gmb03] is a powerful profiler of Java programs. It gives a holistic description of an application's execution, such as standard CPU monitoring to memory leaks. However, it cannot reveal a particular multiactive object setting. In the context of multiactive objects, this kind of debuggers forces the programmer to match the threads to multiactive objects by himself. With our debugger, the threads are automatically matched to multiactive objects, which enables a more comprehensive representation. Also, besides helping in solving true bugs of the application, the multiactive object debugger can also help in determining its bottlenecks, and thus improve the optimisation process.

5.3 A Fault Tolerance Protocol for Multiactive Objects

By nature, active objects are well adapted to a distributed execution of applications, thanks to the fact that they communicate with message passing. As stated many times in the previous chapters, our implementation of multiactive objects, ProActive, targets distributed environments, like clusters, grids, and clouds. Such environments are known for being unpredictable and to have a significant failure rate, that increases along with the scale of the distributed infrastructure [SG10]. As such, providing a fault tolerant distributed execution to multiactive objects is relevant. A fault tolerant distributed application has the ability to avoid the failure or to handle the failure one or several entities that participate to the distributed application.

There are two main ways of dealing with failures: either the initial breakdown must be avoided or the error that follows a breakdown must be intercepted and corrected before leading the system to an erroneous state. The first case corresponds to fully specified and proven systems. We rather focus on the second case of failure management, because distributed systems involve too many parties to be completely modelled and proven safe. In order to cope with a fault, when the fault is detected[2] two solutions can be considered: either the system backups on a sane replicated instance of the faulty process, or the system is recovered to a previous, not faulty state. The right solution to pick often depends on the characteristics of the system. For example, a good metric is the mean time between failures [Jon87], but the choice also depends on the application structure: replication can reveal being costly. In this thesis, we do not focus on replication but rather on the recovery of faulty applications: fault tolerance is embedded in all instances of a ProActive application without requiring any modification in the source code. As a rule of thumb, in the following we always make the assumption that the algorithms are fully distributed, relying on no centralised authority, if not specified otherwise. This is a guarantee of scalability.

In order to recover an application from a previous stable state, one need to

[2] Note that how to detect a fault is out of the scope of this thesis.

collect the state of all the processes regularly. A collected state of a process is called a *checkpoint*. A *recovery line* is a set of collected states for all the processes, plus additional information that make this set *consistent*. Consistency is defined here as the fact to have a final situation where no messages are lost, and no messages are duplicated in case of recovery from a consistent recovery line.

In our case, the processes are likened to active objects, so the checkpoint of an active object consists in a persistent view of the active object's state. ProActive already offers resilience and fault tolerance for ProActive applications that are only composed of active objects. However, the novel implementation of multiactive objects in the ProActive library breaks the fault tolerance protocol for ProActive applications that involve multiactive objects, because of their multi-threaded execution. In this section, we first explain the existing fault tolerance protocol of ProActive and we detail its limitations in multiactive object settings. Then, we propose a preliminary adaptation of the protocol that handles the fault tolerance of multiactive objects for well-defined kinds of applications. Finally, we state the current limitations of the new protocol and give directions for future work.

5.3.1 Existing Protocol and Limitations

The fault tolerance of ProActive active objects [Bau+05] is based on a hybrid protocol that combines two protocols for recovering a faulty application: checkpointing and message logging. A checkpoint is a persistent view of a process at a given moment in the application's execution. In our case, a checkpoint represents the state of an active object at a given moment. Since we focus this section on the fault tolerant protocol and not on an efficient storage mechanism, we assume a centralised authority for storing the checkpoints. Checkpointing regularly, which is, after some timeout, or Time To Checkpoint (TTC), the active objects' state of an application enables building up-to-date recovery lines. Then, if an active object fails, all active objects can be restarted with the states that are available in the last recovery line[3]. The checkpointing protocol that is implemented in ProActive is based on Communication-Induced Checkpointing (CIC). CIC protocols normally

[3] And not with the state of the last checkpoint, since we are not sure that it is part of a recovery line.

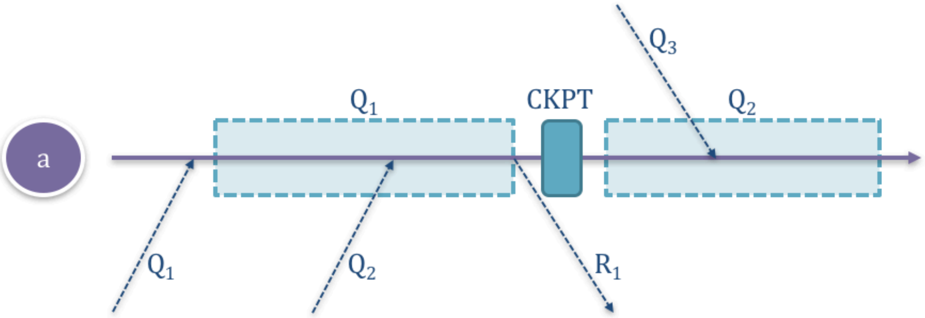


Figure 5.8 – Checkpoints are only located in between the processing of two requests.

ensure consistent recovery lines without having to block all the entities during a checkpoint. To this end, checkpoint indexes are piggybacked on messages, and the message recipient updates its checkpoint index if it is late compared to the message's sender. The consistency of recovery lines is an important property that prevents the system from restarting from the beginning in case of unsound recovery lines (also known as the *domino effect* [DBS84]). However, to this end CIC protocols require that the processes can checkpoint at any time, especially before a request is received. Since ProActive is a Java library, one of its constraints[4] is that the state of a thread cannot be persisted (the `Thread` class is not `Serializable`), thus checkpointing at any time is not possible. This is the reason why the CIC protocol has been adapted in ProActive in order to cope with this constraint. In ProActive, the checkpoint of an active object is always taken when the active object does not process a request, i.e. when it is idle or in between the processing of two requests, because the state of the execution thread is not relevant at this point. Therefore, when a communication-induced checkpoint must be taken, this information is flagged to be taken as soon as the current request finishes.

Active object checkpoints are represented on Figure 5.8. Similarly to the representation of multiactive object executions given by the debugger introduced in Section 5.2, each active object is associated to a time-line representation. The requests that are sent between two active objects are represented with arrows (as well as replies), and labelled Q_i (R_i for replies). The service of a request is represented with a dashed rectangle on the time-line. In between two request processing

[4] Actually the constraint of the JVM.

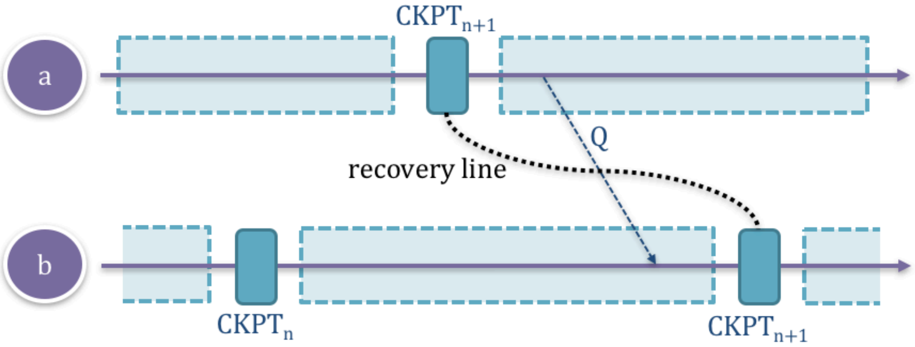


Figure 5.9 – An orphan request.

(where the time-line is empty), the active object is in a stable state, and we can see on Figure 5.8 that a checkpoint can be taken there, noted $CKPT$. The impossibility of taking checkpoint at any time raises two consistency problems regarding the recovery lines that are built. They are explained below, focusing on requests, but they apply the same way for replies.

Orphan requests. They represent requests that have been sent by the sender after the recovery line $CKPT_{n+1}$, and that have been received by the recipient before the recovery line $CKPT_{n+1}$, as shown in Figure 5.9. In case of a recovery from such a recovery line, the orphan requests are duplicated.

In-transit requests. They represent requests that have been sent by the sender before the recovery line $CKPT_{n+1}$, and that have been received by the recipient after the recovery line $CKPT_{n+1}$, as shown in Figure 5.10. In case of a recovery from such a recovery line, the in-transit requests are lost.

In order to solve these problems, an additional message logging mechanism has been implemented in ProActive. It only deals with the orphan and in-transit requests, since the rest of the requests are handled by the adapted CIC protocol. Then, upon re-execution of logged requests during the recovery process, the ordering of execution must be enforced, in order to ensure execution equivalence with the fault-free execution. To this end, the request reception history of an active object is maintained until the next recovery line is built. The reception history allows the correct re-execution of an orphan request: a logged orphan request must

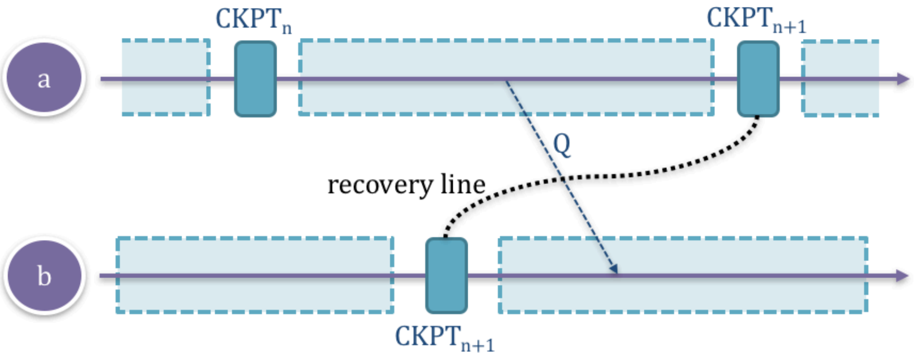


Figure 5.10 – An in-transit request.

wait until it has been received before being executed. In this case, it is called a *promised request*, and it will be able to execute when we are sure that the sender has re-sent it in the re-execution. To handle in-transit requests, the solution is to log these requests and resend them in the right order upon recovery. Thus, the resulting protocol constitutes a hybrid CIC/message logging protocol for the fault tolerance of active objects.

Using checkpointing forces all entities to recover to a past stable state if only one of the entities fails, whereas message logging only make the faulty process restart. Consequently, a recovery with checkpointing protocols can be costly for applications involving many entities, that are active objects in our case. In order to solve this issue, the above protocol has also been extended to handle groups of processes. It represents a mix of the above protocol with a pure message logging protocol [Del07]. Indeed, in case of large systems, the fault tolerance can be handled hierarchically with groups of processes. Checkpointing can be applied for the active objects that are inside a group and message logging can be applied between the groups. This enables restarting only the active objects of one group in case of a failure, and the message logging mechanism is used between different groups indifferently from the communication that happen inside the group. Hereafter, we only focus on the adaptation of the hybrid CIC-message logging protocol detailed before, since the hierarchical composition remains valid.

The described protocol is implemented and proven in ProActive, but it has been designed for active objects, i.e. with a single thread of execution. When using multiactive objects, multiple threads execute multiple requests at the same

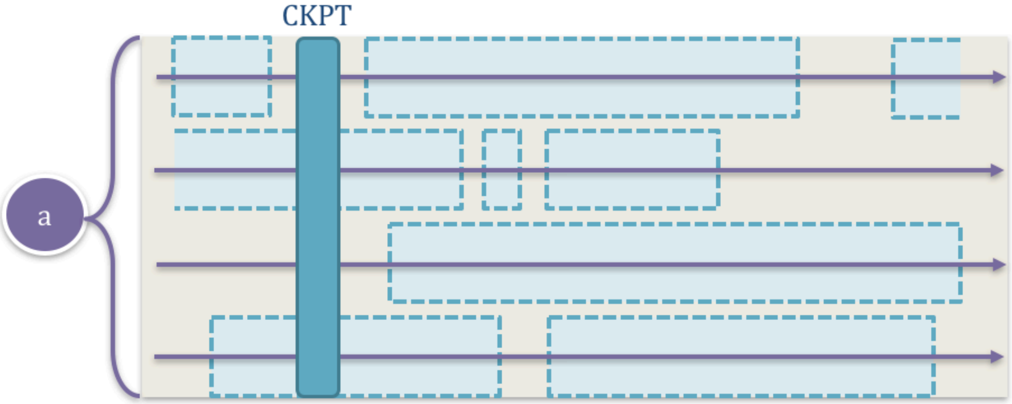


Figure 5.11 – Unsafe checkpoint (in the middle of the processing of two requests).

time, so a checkpoint can no longer be taken safely in between the processing of two requests without additional conditions, because a request might still be executed by another thread. In this case, the remaining of such a request would be lost if a checkpoint were taken at this time. Figure 5.11 represents this situation where two requests execute during the checkpoint. This is why the fault tolerance protocol of ProActive needs to be adapted to multiactive objects' execution.

Fault tolerance of ProActive active objects can be activated through a technical service. Its main purpose is to define the location of the storage server and to define additional nodes to be used in case of a node failure. We keep the fault tolerance technical service unchanged when adapting the protocol to multiactive objects.

5.3.2 Protocol Adaptation and Implementation

Principles

In this work, the first contribution is to re-implement the existing fault tolerance protocol as a set of multiactive object notions. Basically, we process checkpoints like multiactive object requests, that have a dedicated compatibility and priority. Firstly, we generate checkpoint requests when they are needed. Since a checkpoint must be taken as soon as possible when it is triggered, we arrange a suitable compatibility and priority for checkpoint requests. Secondly, the execution policy of multiactive objects automatically handles the scheduling of the checkpoint requests, taking into account the multiactive object setting specially made for checkpoint requests. The guarantees that are given by multiactive object execution ensure that a checkpoint is correctly taken under the conditions we set. This approach makes the implementation of the fault tolerance protocol for active objects easier and more generic, because it is less entangled with the ProActive middleware.

The second contribution of this work is to propose an approach to handle the checkpointing of multiactive objects, by adapting the existing protocol. In order to correctly checkpoint a multiactive object, we need to look at the first property of the fault tolerance protocol: to checkpoint an active object, it must be in a stable state. The only time when an active object is in a stable state is when it is not processing any request. For a multiactive object, it means that all its threads must be idle. Consequently, in order to checkpoint a multiactive object, we must find the moment of the execution when all its threads are idle. If one waited for this situation to happen without forcing it, there would be a high probability that no checkpoint could ever be taken. Therefore, we propose an approach that consists in flushing all the currently processed requests to reach the stable state before a checkpoint is taken. The particularity of this approach is that we only rely on multiactive object notions in order to reach this stable state. The principle is to temporarily constrain the threads of a multiactive object when a checkpoint request is triggered. By setting the thread limit to one thread, we can ensure that, eventually, when all currently executed requests are completed, only one thread is left to process a request. In addition, by setting the kind of thread limit to a hard thread limit, we can ensure that no requests remain in wait-by-necessity when checkpointing. These two manipulations leave room for a checkpoint request to be executed in an isolated manner, and to be executed when the state of the multiactive object is stable, because no applicative requests are being executed nor paused meanwhile. As our approach presents limitations, we will see in the next subsection the restrictions of our solution and some directions to improve it. But first, we focus on the first contribution of this work which is a generic implementation of the existing fault tolerance protocol, an that prepares the context for the new protocol we propose.

Checkpointing as a Request

In the new implementation of the existing fault tolerance protocol, we turn the checkpointing action into a special request, that all fault tolerant active objects can serve. Representing the checkpointing action like a request allows us to parametrise it with multiactive object annotations. The method to execute for serving this request has the reserved name `__checkpoint__`, and this method is put in a multiactive object group of the same name, programmatically. In practice, when the annotations of a multiactive class are interpreted, the group, the compatibility and the priority of the `__checkpoint__` group are automatically inserted, in addition to what the programmer specified through annotations. Our goal is to execute the checkpoint requests as quickly as possible, so we use a specific compatibility and priority for that. In order to lift the checkpoint requests to the ready queue (see Subsection 3.3.1), we make the `__checkpoint__` group compatible with all the other groups of a multiactive object. In order to have the checkpoint requests first in the ready queue, we assign to this group the highest priority, that is uniquely defined for this group[5]. Thanks to this multiactive object setting, checkpoint requests can overtake any request and can always be first in the ready queue, thus a checkpoint is always taken as fast as possible. In the end, this approach is easier to implement and to reuse because a checkpoint is treated like any other request, and it also complies to the same multiactive object rules.

Triggering Checkpoints

Since the checkpoints can be faithfully represented as multiactive object requests, we must figure out now when and how to generate them and put them in the request queue. Firstly, the checkpoint requests are generated at the same place as before when the checkpoint index is detected to be out of date. If it is the case, in our re-implementation of the existing protocol we trigger a checkpoint by creating a new checkpoint request and by putting it in the request queue. By the multiactive object configuration explained in the previous paragraph, we know that this checkpoint request will be the next request to be selected for execution

[5] More precisely, the `__checkpoint__` group has a priority dependency to all the roots of the defined priority graph; thus it becomes the new single root of the new priority graph.

by the multiactive object scheduler. Compared to the previous implementation of the checkpointing action, this implementation requires less additional information to carry together with the active object's state when saving a checkpoint[6]. Indeed, in the previous implementation the request that was responsible for the triggering of a checkpoint had to be saved 'manually' along with the checkpoint, because it was already removed from the request queue for execution. Now, in the same situation, no additional information needs to be saved along with the active object's state, because this request is still part of the request queue, therefore part of the active object's state. Thus, the new implementation of the existing fault tolerance protocol involves less special cases to consider, both when checkpointing and when recovering. It fully handles the fault tolerance of active objects, and sets a propitious context for adapting the protocol to the special case of multiactive objects.

Flushing Parallel Requests & Checkpointing

We present now the adaptation of the fault tolerance protocol to handle the checkpointing of multiactive objects. The main challenge in the case of multiactive objects is to create sound checkpoints. Since this is a local problem, related to the internal scheduling of requests and threads, we can again face it by using the abilities of multiactive objects. The key to be able to checkpoint a multiactive object is to ensure that none of its associated threads execute a request when the checkpoint request is executed, that is to *isolate* the checkpoint request. In order to isolate a request using multiactive object notions, one could think that making the checkpoint requests incompatible with all other requests would fit. However, since we want the checkpoint requests to be in the ready queue as soon as possible (to execute them as soon as possible), we configured the checkpoint request such that it is compatible with any other request. Still, in order to ensure the isolated execution of checkpoint requests, the approach that we have chosen uses the multiactive object API that manipulates the kind of thread limit, seen in Subsection 3.2.1. When a checkpoint must be taken, we immediately set the thread limit of the multiactive object to one thread, and we apply a hard thread

[6] As shown in [Del07], the checkpoint size has a significant impact on the performance of the fault tolerance protocol.

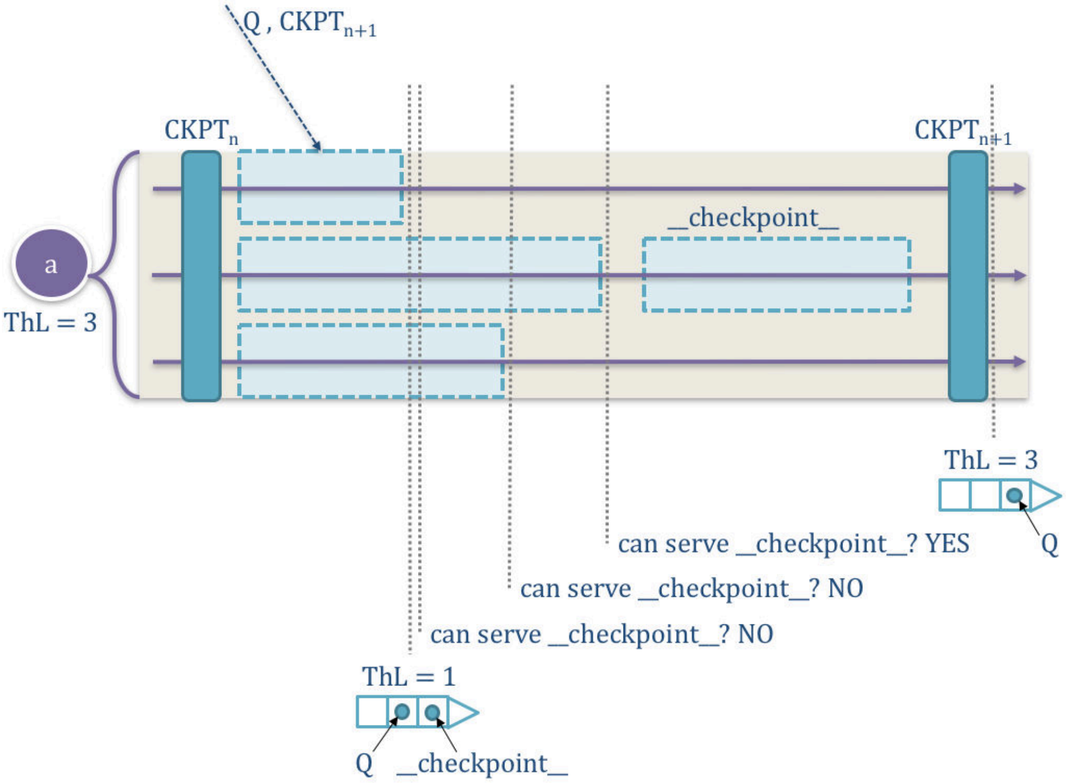

Q , CKPT$_{n+1}$
CKPT$_n$
CKPT$_{n+1}$
a
ThL = 3
__checkpoint__
ThL = 3
Q
can serve __checkpoint__? YES
can serve __checkpoint__? NO
can serve __checkpoint__? NO
ThL = 1
Q
__checkpoint__

Figure 5.12 – Flushing of requests thanks to the thread limit of multiactive objects. The illustrated queue represents the ready queue.

limit (i.e. all the threads, even those in wait-by-necessity, are taken into account in the thread limit). Consequently, the checkpointing request is first in the ready queue, and it cannot execute until the other requests complete. Note also that the checkpointing request cannot execute upon the wait-by-necessity of the other requests, because the hard thread limit prevents the compensation of a thread in wait-by-necessity. Indeed, we do not want to checkpoint if a request is not finished. Assuming that all requests can progress, this mechanism ensures that we can only have a decreasing number of threads that process requests. This number decreases until only one executed request is left, and allows the checkpoint request to be executed just after the completion of this last request.

An example of scheduling of the `__checkpoint__` request is displayed on Figure 5.12. Request Q first triggers the need for checkpointing. The thread limit is changed just after the completion of a request (`ThL = 1`), and the checkpoint request ends up first in the ready queue. Then, each time a request is completed, the scheduling process checks if a request can be executed. Since the thread limit is exceeded, no scheduling can occur until all currently processed requests are completed. When the last executed request terminates, the next selected request is the `__checkpoint__` request. Finally, when the checkpoint completes, the original thread limit is restored, as well as the original kind of thread limit, and a normal execution can resume. Note that changing the thread limit and the kind of thread limit in a dynamic manner at runtime is always legitimate as long as the modifications are more restrictive than the specifications given by the programmer. Indeed, more restrictive modifications are harmless in the sense that they only reduce the set of the executions that are possible (no new behaviours are introduced), except when this reduction leads to a deadlock; this case discussed in Subsection 5.3.3. In the case of our fault tolerance protocol, the modifications that we bring at runtime always reduce the set of possible executions since the restrictions we apply are the strongest possible restrictions related to threads.

Recovery

In order to complete the adaptation of fault tolerance to multiactive objects, not only the checkpointing phase of the protocol must be adapted, but also the recovery phase. In particular, the existing implementation of the recovery phase takes a particular care in preserving the request ordering during a re-execution. To this end, a special class is used to handle the re-execution of orphan requests. In a re-execution, orphan requests are promised: they are in the request queue before having been received. In ProActive, promised requests are wrapped in an `AwaitedRequest` class, so that a promised request contains the history information that allows the scheduler to provide an equivalent re-execution upon recovery. This wrapper also allows the scheduler to wait for the reception of the awaited request. However, in multiactive objects, the notions of compatibility, priority and thread limit are the notions that massively control the order of execution of requests. As such, promised requests must behave like the requests they enclose, in terms of multiactive object execution, in order to ensure the equivalence of the re-execution. This is not a major problem because we know the type of the request that we are waiting for. Consequently, we can easily obtain the static multiactive object information of the request, that is mandatory for its scheduling.

5.3.3 Restrictions and Open Issues

So far, we have adapted the fault tolerance mechanism of the ProActive middleware such that it can be applied to applications that are built with multiactive objects. First, our adaptation is fully compliant with old-style active objects. Second, multiactive object features allow a neater implementation of the fault tolerance protocol of active objects. This provides a clearer framework to reason about the protocol, extract its properties and prove them. However, the proposed protocol is still seminal in the context of multiactive objects, because some restrictions apply for a subset of multiactive object-based applications. In particular, two limitations must be discussed.

The first limitation regards the recovery phase in case of dynamic compatibilities. We have seen that looking at the static multiactive object information of a promised request is easy, because the static information is embedded in the type of the request. However, when a promised request features a dynamic compatibility (that is decided upon a method execution, possibly using the request's parameters, see Subsection 2.4.2), some part of the knowledge is missing to be able to decide on the scheduling of this promised request. Without further modifications of the recovery phase, no decision can be taken until the promised request is actually received by the recipient in the re-execution. One possible solution to handle dynamic compatibilities during a recovery could be to save more information in the reception history, so that it can handle the orphan requests that feature dynamic compatibilities. Nevertheless, this solution seems to be non trivial to design and to implement, and it could have a significant runtime cost, that should be evaluated.

The second limitation regards the main mechanism of the new fault tolerance protocol and more precisely, the way we flush the requests that are being executed in order to take a checkpoint. Our approach to ensure that the multiactive object is in a stable state before checkpointing is brute-force. In particular, we do not check if, in order to complete a request, the multiactive object needs to execute other requests. For example, this is the case for recursive calls: if the need for

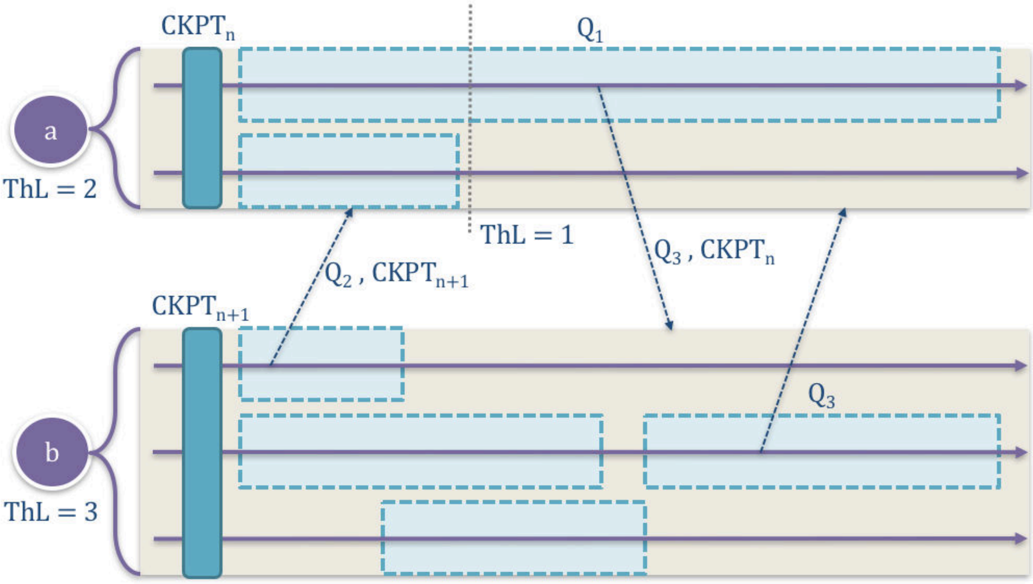


Figure 5.13 – Example of a deadlocked execution. The reply of Q_3 is needed to complete Q_1.

checkpointing comes during the processing of a recursive request (this request sends requests to the current multiactive object), then the application deadlocks. A deadlock is also caused if a callback to the current multiactive object originates from a request that must be flushed by our checkpointing mechanism, as shown in Figure 5.13. In this example, the completion of request Q_1 on multiactive object a is required to perform the checkpoint. Also, the completion of request Q_1 depends on the completion of request Q_3, that is executed on multiactive object b. However, the completion of request Q_3 depends on the reply of another request sent to multiactive object a. Since a is temporary in a hard thread limit of one thread, it cannot process the required request. In the end, request Q_1 will never complete, and multiactive object a cannot execute requests any more.

In general, any request that leads to a dependency cycle between multiactive objects will lead to a deadlock if this request is processed at the same time as a checkpointing request is attempted to be served. A number of solutions can be designed such that multiactive object checkpoints can be taken without a risk of deadlock. We are reviewing below possible solutions or directions from the most restrictive to the lightest approach.

Cycle-free applications. A first solution is to remove all request dependencies in the application, which means, never wait for a reply. This solution is pretty restrictive because it forces applications to be refactored to this end, and because the modifications must be applied at development time. However, this solution is adapted for the applications that are based on a pure actor model, when no synchronisation through futures is allowed.

Subset of execution cases. A second option could consist in avoiding the processing of cyclic requests at the same time as a checkpoint is needed. However, how to detect this situation a priori seems difficult, and can be very inefficient if the cycle involves many activities. Since finding a consistent recovery line in this case is probably impossible, the only solution that is left is to cancel this particular checkpoint, and retry later. However, the implementation of this behaviour might still be costly, and the checkpoints are taken less regularly.

Dynamic deadlock detection. If a multiactive object cannot progress for a long time, although there are available threads and the ready queue is not empty, then the thread limit could be relaxed progressively in order to solve the deadlock. But this solution is first not elegant and second not reliable, since we must prevent all requests that are not involved in the deadlock to be executed meanwhile.

Multiactive object configuration. Lastly, a more refined solution uses the threading setting of multiactive objects where reentrant requests are treated on the same thread (see Subsection 3.2.1). By enabling the reentrance parameter of a multiactive object (with the parameter `hostReentrant=true` in the `@DefineThreadConfig` multiactive object annotation), we can make sure that cyclic requests are processed on the same thread, that is, without requiring any additional thread. In this case, the requests that are executing when a checkpoint request is in the queue could be flushed even if the thread limit is already set to one strict thread for the checkpoint. This solution requires that all requests involved in a cyclic dependency are compatible in the same multiactive object. However, the compatibility of such requests must also be fulfilled for a regular execution of the application (that is without fault), thus it seems a reasonable prerequisite.

In the end, the most plausible solution for handling the integrity and the feasibility of a multiactive object checkpoint is the last proposed solution. For now, there is no implemented solution that both guarantees that a multiactive object checkpoint can be taken in a bounded time and that the checkpointing process cannot deadlock. Providing the two guarantees at the same time for a fault tolerant multiactive object-based application is still unresolved and unimplemented, although we have given some directions for it, that should be further explored.

* * *

In this chapter, we had a look at two non functional aspects related to the support of development and execution of applications that are developed with the multiactive object programming model. Firstly, we have focused on the debugging of complex multiactive object-based applications, by visualising the execution of multiactive objects through a debugger tool. What is special about this debugger is that it keeps a close relationship between the notions that are manipulated trough the development of the application with multiactive objects, and what can be displayed by the debugger. Other debugging tools are very popular in the Java community, but their are related to notions that are external to the programming model. In ProActive, the IC2D debugging tool [Bau+01] has been developed previously in the context of distributed active objects. It enables the dynamic visualisation of the deployment of a ProActive application: active objects, virtual nodes, and JVMs. IC2D was primarily introduced in the context of active object migration [Bad+06; CH05]. Thus, IC2D is oriented towards the localisation and wiring of active objects rather than towards their execution flow and interaction. A merge of IC2D and our debugger tool should be addressed in future work. However, to this end our debugger must be adapted to support a dynamic actualisation. Already, the way we process the logs (based on timestamps) fits such a dynamic update of the visualisation frame without recomputing the information related to the global execution.

Secondly, in this chapter we have focused on the reliable execution of multiactive objects, by adapting the fault tolerance protocol to make it adapted to a multi-threaded execution. The new protocol that we have described makes an extensive use of multiactive object notions in order to coordinate the threads and enable a safe checkpointing. The presented approach, that handles isolation of execution of checkpointing requests, is similar to the one we take in the ProActive backend for ABS (Subsection 4.2.1) for the simulation of a blocking awaiting (get statement). This proves that the manipulation of multiactive object notions can form a generic approach to implement specific execution policies, that can also adapt during the execution of the application. Our adaptation of the fault tolerance protocol is nevertheless limited to applications that do not have recursive calls during the checkpointing process. In particular, our protocol can introduce a deadlock upon specific executions. Removing such deadlocked executions is a short-term future work, and we have given in this thesis some directions to progress on this issue. Afterwards, ProActive will be able to provide transparent fault tolerance for active and multiactive object-based applications, without distinction and with a better implementation of the protocol. In summary, our new implementation of the fault tolerance protocol is better integrated with the ProActive middleware and takes advantage of the multiactive object programming model, as a high-level abstraction. This generic approach makes it also fitted to other active object languages, as well as to other multi-threaded paradigms.

Both of the aspects that were presented in this chapter are supported at deployment time, using ProActive technical services as the standard configuration mechanism for easily adding non functional aspects. Thanks to the various tuning mechanisms that are offered by multiactive objects, the multiactive object programming model is adapted to develop and execute high performance distributed applications. However, this kind of applications requires powerful tool sets, that allow the programmer to analyse and to tune the application iteratively. Also, this kind of applications is usually long-running, so a failure might represent a huge loss in terms of computing time. Consequently, the failure of the whole application must be prevented as much as possible. We have addressed in this chapter these two concerns and this way, we believe that we have raised the multiactive object programming model at the level of expectation of high performance computing programmers.

Chapter 6

Application: Fault Tolerant Peer-to-Peer

Contents

In this penultimate chapter, we present an application that aggregates several works developed in this thesis. First, we introduce the context of this chapter by exposing the challenge of broadcasting in peer-to-peer networks, focusing more particularly on one of them: the CAN. We present an efficient broadcast algorithm for CAN that is published in [2]. Then, we present an application that consist in a resilient broadcast in a CAN made of peers that are implemented with fault tolerant multiactive objects, through the ProActive library. This contribution can be seen both as a practical application of the fault tolerance protocol for multiactive objects introduced in Chapter 5, and as a middleware approach for dealing with faults in distributed broadcasts.

6.1 Broadcasting in Peer-to-Peer Networks

6.1.1 Context: CAN Peer-to-Peer Networks

Structured overlay networks [EAH04] are defined as peer-to-peer networks that operate at the application-level, and where the nodes are virtually organised according to structural properties. In such networks, the nodes are called *peers*, because they organise independently and only according to a local view of the network. A peer can only communicate with a subset of the peers that belong to the network. This subset represents the *neighbours* of a peer. The well-defined properties of structured overlay networks enables reaching a particular peer from any other peer in a bounded number of hops. Providing guarantees on reachability is the main advantage of structured overlay networks. When structured overlay networks are used to store and retrieve content, data are deterministically consigned to a peer by following a precise computation of the data location. Consequently, each peer in the network manages a well-defined set of data, such that content retrieval complies to nice properties. The operation of retrieving content is always bounded and gives a deterministic answer, although the guarantees depend on the type of the considered structured overlay network. The CAN [Rat+01a] is a type of structured overlay networks whose structure is ruled by a multidimensional Cartesian coordinate space.

Structure

In CAN, a finite Cartesian coordinate space of dimension D is entirely partitioned among the peers of the network. Each peer is responsible for a part of the coordinate space, called a *zone*. Each zone is defined with an upper-bound coordinate and lower-bound coordinate for all the dimensions of the space. Therefore, a zone has the shape of an hyper-rectangle. Since the coordinate space of a CAN is entirely partitioned, the zones abut each other in one or more dimensions. The set of peers whose zone abuts the zone of a given peer represents the neighbours of this peer. A peer has at least two neighbours per dimension, one in each direction (increasing or decreasing, considering the coordinates), but the number of neighbours of a peer is not bounded. A peer can only communicate with its neighbours, i.e.

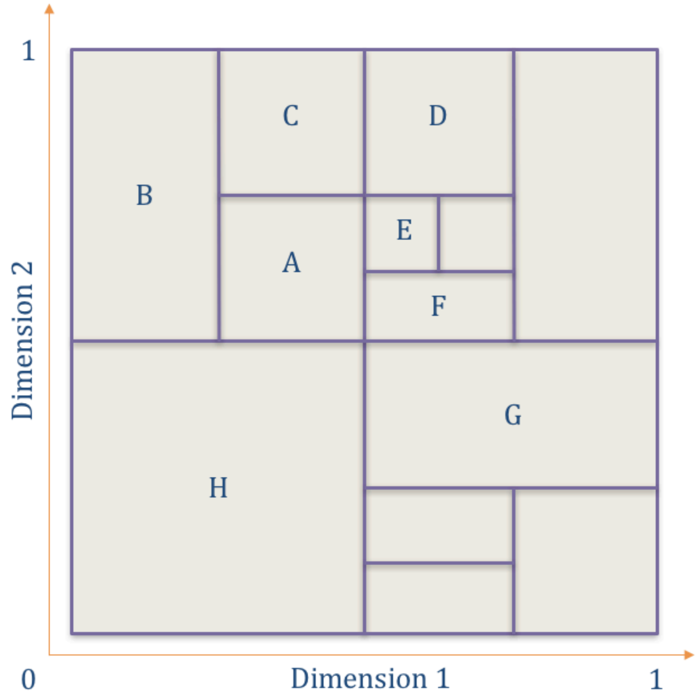


Figure 6.1 – A 2-dimensional CAN network.

with the peers whose zone abuts its own zone. Thus, there cannot exist a gap in the coordinate space between a peer's zone and its neighbours' zones. Figure 6.1 displays a CAN in two dimensions with 13 peers in a $[0,1]\times[0,1]$ coordinate space. In this example, peer A has 5 neighbours. Peer D and peer G are not part of the set of neighbours of peer A, because they share no edge.

Data Storage and Retrieval

In order to store content in a CAN, one should first determine the location of data in the data space. The usual way of storing content is by using a key-value representation: the key determines the storage location while the value is the content to be stored. To store data in a CAN, the data key it must first be hashed to a point of the coordinate space. Once this point is computed, we can determine the unique peer of the network whose zone contains the point. This peer is responsible for storing the content. Searching for stored data is done in a similar way: the corresponding key is hashed and used to find the peer that should store it. If the targeted peer does not have the searched data, then it guarantees that it is not present in the network. Storing and retrieving are performed through an iterative routing process. The process starts from any peer that initiates the query, and then the query is forwarded by successive hops among neighbours until the targeted peer is reached. Approaching the right zone step-by-step is finite and bounded, because the distance from the searched point is reduced at each hop (at least reduced in one dimension). In general, structured peer-to-peer networks give better bounds than unstructured ones.

Construction

The general steps to build a CAN are the following. At the beginning, one peer owns the entire coordinate space. When a new peer wishes to join the network, it picks a point in the coordinate space. The zone that contains this point is split into two parts: the new peer takes the responsibility of one part, and the peer that owned this zone before takes the responsibility of the second part. The coordinates of the zones are updated accordingly, as well as the sets of neighbours. Several policies for selecting the peer to join can be used, for example to produce balanced zones. Splitting is usually done alternatively in each dimension. For example, for a CAN in two dimensions, the splitting is done alternatively vertically and horizontally, as peers join the network.

6.1.2 Efficient Broadcast in CAN

Structured overlay networks are powerful underlying structures for communication-oriented and storage-oriented distributed systems in large-scale settings. This power is enabled by a fully decentralised management. As such, when a peer needs to share an information with all the other peers, it cannot send this information to a shared channel, nor communicate it directly to all the other peers. Instead, it has to rely on its local view of the network, which is the set of its neighbours. The basic approach to broadcast information under these conditions is to progressively cover the network by forwarding the broadcast massage from neighbours to neighbours. However, this approach has a high overhead in terms of duplicated messages. Indeed, it does not use the properties of the underlying structure of the network to route the broadcast more efficiently. The goal of the work that is presented in this subsection is to design a broadcast algorithm for CAN that takes advantage of the CAN structure, so that a minimum number of exchanged messages covers the whole network, and without any global knowledge. This work is published in [2]. In existing broadcast algorithms for CAN, either the broadcast operation is still noisy with a few duplicated messages, like the M-CAN algorithm [Rat+01b], or it requires to maintain an additional overlay, e.g. a spanning tree [Per85]. Here we propose an algorithm that is optimal in number of messages: each peer of the CAN receives the broadcast message exactly once, and no additional structure needs to be maintained for that.

Principle

Our algorithm extends M-CAN [Rat+01b]. It systematically removes the duplicated messages that are left by the M-CAN algorithm, by specifying additional forwarding constraints. We then add specific constraints that remove all of the duplicates in any CAN configuration. In M-CAN, the broadcast starts from an initiator peer that sends the broadcast message to all of its neighbours. Then, the broadcast is propagated according to a given dimension (from dimension 1 to dimension D), and to a given direction (either in ascending direction if the coordinates are higher or in descending direction if the coordinates are lower). A peer that receives the message from a neighbour along dimension i in direction dir only forwards the message to all the neighbours that are located on dimensions that are lower than i, plus to the neighbours that are located on dimension i but only in the propagation direction dir. The broadcast message is never propagated to neighbours that are located on higher dimensions than the dimension of reception of the message. An example of M-CAN broadcast is given in Figure 6.2. In addition to these general rules, in M-CAN a deterministic condition is used to remove some duplicates: a peer forwards the broadcast to a neighbour that fulfil the above rules only if it abuts the lowest corner of this neighbour. The lowest corner is defined as the corner that is along the dimension of propagation and that minimises all the coordinates in other dimensions. This deterministic condition is called the *corner criteria*. In Figure 6.2, the corner criteria prevents 6 duplicated

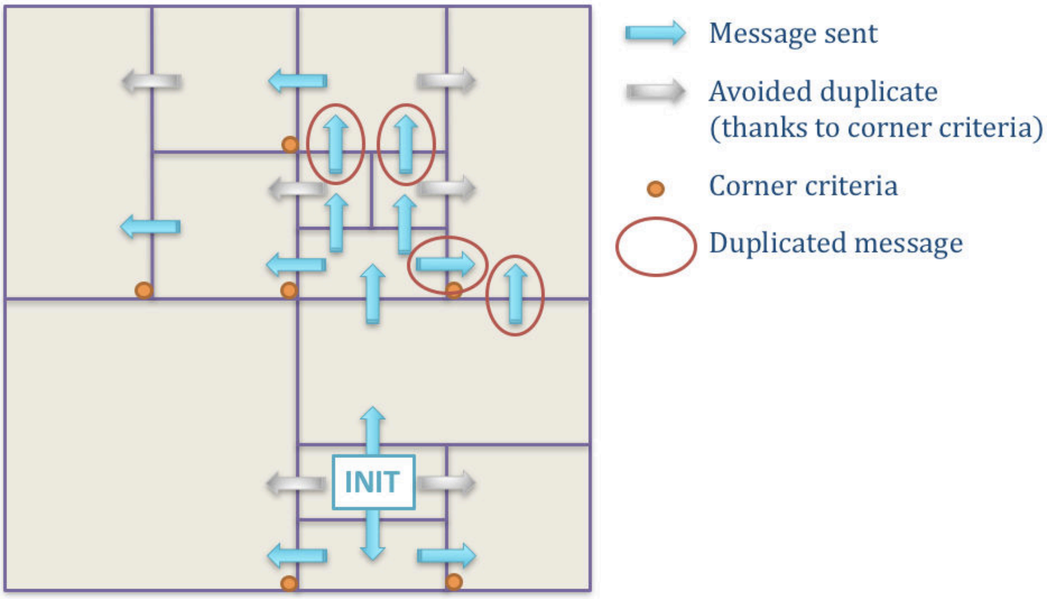


Figure 6.2 – M-CAN broadcast.

messages from being sent in the CAN. However, the M-CAN algorithm can still lead to duplicated messages, notably when the CAN has a lot of dimensions. Indeed, M-CAN only removes the duplicates that arise in the first dimension, which is where most of the duplicates are present. The corner criteria cannot be applied in higher dimensions without jeopardising the correctness of the broadcast, which stipulates that all peers must received the broadcast message[1].

In the efficient broadcast algorithm that we propose, we use the corner criteria in addition to another constraint that we specially introduce to remove all the duplicates. More precisely, our broadcast removes the duplicated messages that arise in higher dimensions than the first one, and thus completes the M-CAN. To this end, we introduce a *spatial constraint*, whose main idea is to reason on a sub-CAN, in order to progress recursively, as if the broadcast was always propagating on a single dimension. Our efficient broadcast algorithm for CAN is proven to be correct and optimal in number of messages [BH13; 2].

[1] Note that in Figure 6.2 the peers' zone are always split in the middle. If it were not the case, for example for load balancing reasons, then the probability to have duplicates would be higher, because the neighbours' coordinates would not match exactly the peer's coordinates.

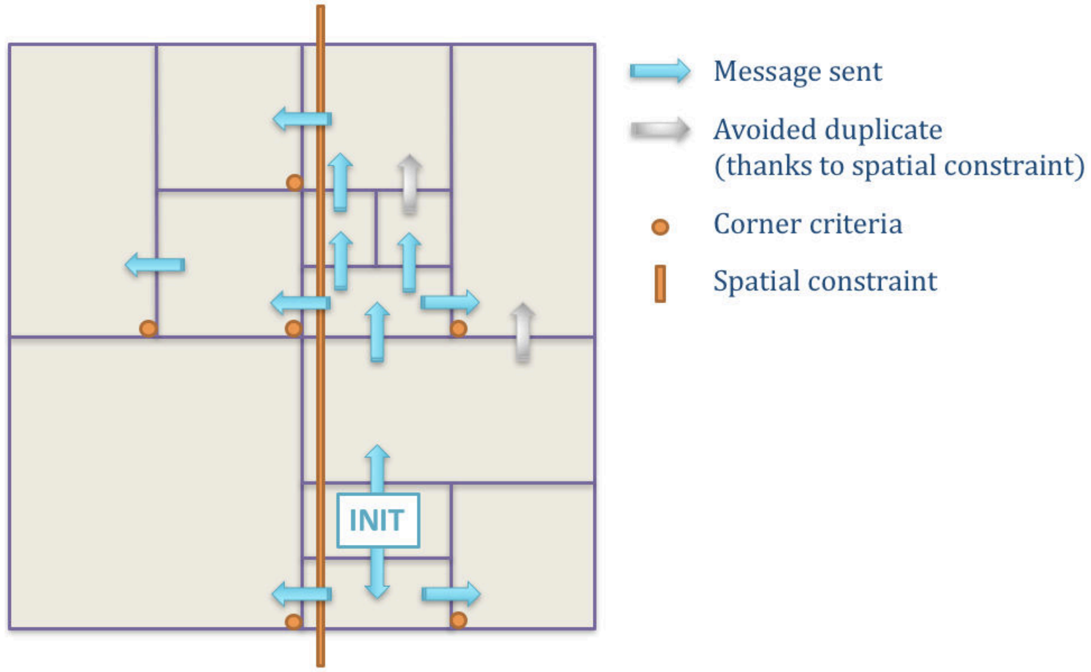


Figure 6.3 – Optimal broadcast.

Algorithm

In practice, to prevent having duplicates in high dimensions, we constraint the peers to only consider the neighbours that belong to a particular hyperplane of the coordinate space. We define the hyperplane as a set of fixed values in all the dimensions apart from dimension D. The spatial constraint is thus a hyperplane in dimension $D-1$. In order to bootstrap the algorithm, the hyperplane must be contained in the zone of the initiator of the broadcast. Then, the neighbours of the initiator that belong to this hyperplane form a sub-CAN of dimension $D-1$. Therefore, we can apply the same selection process on this sub-CAN with a hyperplane of dimension $D-2$. By recursion, the hyperplane eventually becomes a line. Therefore, we can propagate the message along this line without any duplicate, because the line can only be contained by one peer in each direction. The peers that did not receive the message on a high dimension because of the spatial constraint are ensured to receive the message later from another peer on a lower dimension. This is what happens in Figure 6.3. The fact that the top-right peer did not receive the message at hop number 2 (on dimension 2) is not a

```
 1: upon event reception of message M on dimension dim0 and direction dir0 on peer
 2:   for each k $\leq$ dim0 do
 3:     if $k = D + 1$ then
 4:       direction $\leftarrow \varnothing$
 5:     else
 6:       if $k <$ dim0 then
 7:         direction $\leftarrow$ {descending, ascending}
 8:       else
 9:         direction $\leftarrow$ dir0
10:       end if
11:     end if
12:     for each $dir$ in direction do
13:       for each neighbour on dimension $k$ and direction $dir$ do
14:         for each $i$ in $1 \ldots k-1$ do                          ▷ Spatial Constraint
15:           if not ( neighbour.$LB[i] \leq$ constraint$[i] <$ neighbour.$UB[i]$) then
16:             skip neighbour
17:           end if
18:         end for each
19:         for each $i$ in $k+1 \ldots D$ do                           ▷ Corner Criteria
20:           if not( peer.$LB[i] \leq$ neighbour.$LB[i] <$ peer.$UB[i]$) then
21:             skip neighbour
22:           end if
23:         end for each
24:         send message on dimension $k$ and direction $dir$ to neighbour
25:       end for each
26:     end for each
27:   end for each
28: end event
```

Algorithm 2 Efficient broadcast algorithm for CAN.

problem since it receives the message at hop number 3 (on dimension 1).

The precise algorithm is given in Algorithm 2. LB is the short for lower bound and applies on a dimension i to retrieve the corresponding coordinate. Similarly, UB is the short for upper bound and is applied in the same way as LB. Overall, when scanning the neighbours, all the dimensions are either checked against the spatial constraint (Lines 14-18) or against the corner criteria (Lines 19-23), except the dimension on which the message was received (dimension k in Algorithm 2). These two conditions filter the neighbours such that the ones that remain at the end of the algorithm are the ones to which the broadcast message must be forwarded.

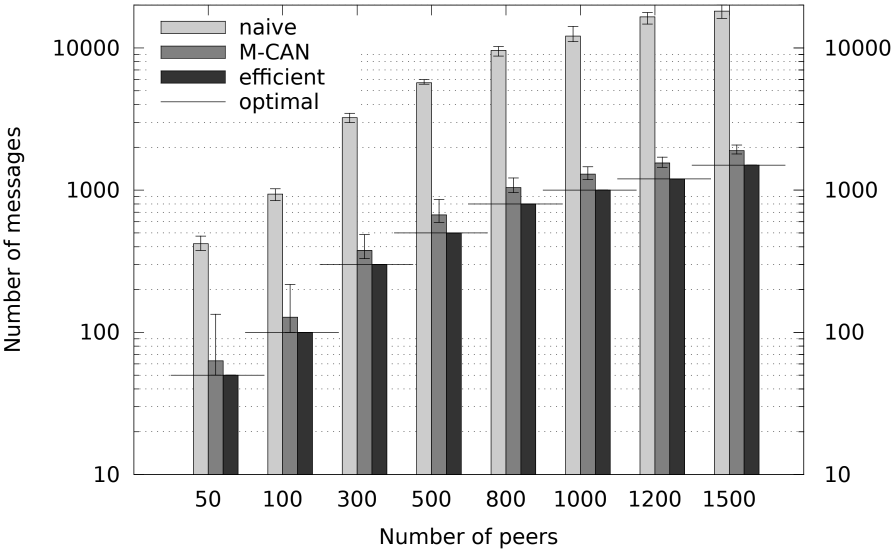


Figure 6.4 – Average number of exchanged messages and optimal number of exchanged messages with a CAN in 5 dimensions.

In order to apply the conditions, the message piggybacks the spatial constraint as well as the dimension and direction onto which the broadcast message is sent.

Experiments

We briefly present some experiments that highlight the benefits of our efficient broadcast algorithm for CAN. The details of the experimental setup and platform are further developed in [2]. The experiments represent realistic scenarios considering the distribution degree (using up to 200 machines, spread across a grid), the number of peers in the CAN[2] (up to 1500 peers), and the number of dimensions of the CAN (up to 15 dimensions). The objective of the experiments is to show that, in realistic situations, using a decentralised broadcast that exchanges the minimum number of messages significantly reduces the volume of exchanged data, and thus impacts positively on the execution time of the broadcast. We mainly compare the performance of our efficient broadcast algorithm to the M-CAN algorithm, that represents the best improvement of the broadcast operation in CAN, besides our algorithm. We also show in comparison the performance of the naive broadcast

[2] We use an implementation of CAN in Java, the EventCloud [Pel14].

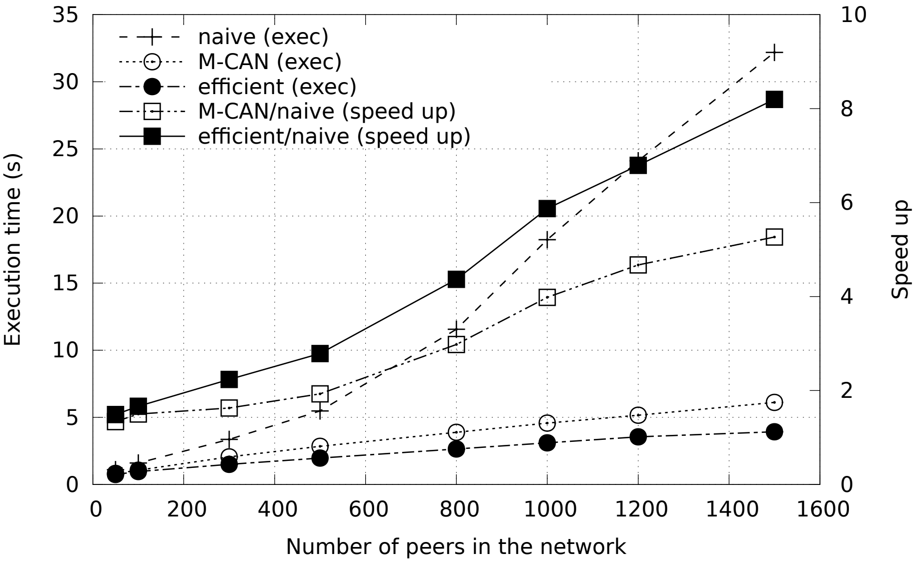


Figure 6.5 – Average execution time of broadcast and speed up from the naive broadcast with a CAN in 5 dimensions.

algorithm, that covers the CAN by solely using the neighbour relationship.

Figure 6.4 shows the average number of exchanged messages for each broadcast algorithm, where the optimal number of exchanged messages is highlighted by a horizontal line. M-CAN already reduces greatly the number of duplicated messages compared to the naive broadcast. However, for 1500 peers, 395 duplicated messages are recorded for M-CAN on average, and this number varies depending on the shape of the CAN and on the location of the broadcast initiator. On the other hand, our efficient broadcast always takes the minimum number of messages in order to cover the CAN. We have also measured the total volume of exchanged data for this experiment[3]. When the CAN is made of 1500 peers, M-CAN generated 25.6 MB of data on average. Our efficient broadcast generated only 20.3 MB of data on average, i.e. it led to a 20% reduction of exchanged data volume. Besides exchanged messages, we also measure the execution time of the different broadcast, i.e. the time needed for all the peers to receive the first broadcast message, shown in Figure 6.5. The performance of the naive broadcast algorithm suffers from the huge number of duplicates. On the contrary, both M-CAN and our

[3] The messages have no useful payload in these experiments.

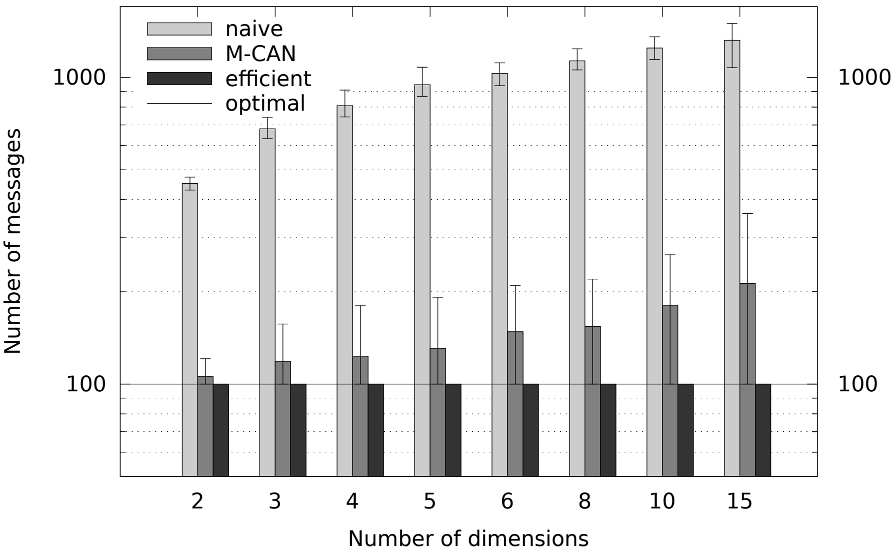


Figure 6.6 – Average number of exchanged messages and optimal number of messages with 100 peers in the CAN.

broadcast maintain a good performance as the CAN gets larger. However, compared to the execution of the naive broadcast over 1500 peers, M-CAN reaches a speed up of 5 whereas our efficient broadcast reaches a speed up of 8, again due to the presence of duplicates in M-CAN.

Apart from varying the number of peers, we also take interest in varying the number of dimensions of the CAN from 2 to 15 dimensions, with a fixed number of 100 peers, as shown in Figure 6.6. We can observe a regular increase of duplicates with M-CAN: from 5% of duplicated messages in dimension 2, to 112% of duplicated messages in dimension 15, in average. This is due to the fact that the number of dimensions is correlated to the average number of neighbours of the peers, and by transitivity this impacts the propensity of duplicates.

The previous experiments show that the remaining duplicates of the M-CAN algorithm have a clear impact on realistic systems. Our efficient broadcast algorithm for CAN offers a significant improvement in terms of bandwidth and execution time by totally avoiding duplicated messages, especially when the number of dimensions of the CAN is high.

6.2 Fault Tolerant Broadcast with ProActive

The efficient broadcast algorithm for CAN that was presented in the previous section has the particularity of being optimal in number of messages, which leads to an efficient execution. In particular, the broadcast message is received only once per peer, meaning that for each peer there exists a single peer from which the message must be received. In this context, without preventive measures the crash of a peer during the efficient broadcast inevitably leads the system in an erroneous state, since some peers will end up in receiving the broadcast and some other peers not. Alternative broadcast algorithms do not perform better in presence of faults, because the peers that receive duplicated messages cannot be determined in advance, thus duplicates cannot serve to robustness. Moreover, as the crash of a peer generally entails the consistency of the data space of the peer-to-peer system, relying on the fault tolerance of a distributed algorithm is not enough to carry out a sane execution of the application. In this section, we propose to make the broadcast operation for CAN completely reliable at the middleware-level, and to automatically recover the data space consistency after the crash of a peer. To this end, we rely on the fault tolerance mechanism of multiactive objects, introduced in Section 5.3 and implemented in the ProActive library.

6.2.1 A CAN Implementation with Multiactive Objects

First of all, we propose an implementation of CAN realised with the ProActive library. The purpose of this application is to offer a high performance distributed system for data storage. In this application, each peer of the CAN is implemented with a multiactive object that is instantiated from a `Peer` class. The most interesting part of the `Peer` class is shown in Listing 6.1.

```
@DefineGroups({
  @Group(name="gettersOnImmutable", selfCompatible=true),
  @Group(name="dataManagement", selfCompatible=true),
  @Group(name="broadcasting", selfCompatible=false),
  @Group(name="monitoring", selfCompatible=true)
})
@DefineRules({
  @Compatible({"gettersOnImmutable", "broadcasting", "dataManagement"}),
  @Compatible({"gettersOnImmutable", "monitoring"})
})
@DefineThreadConfig(threadPoolSize=2, hardLimit=false)
public class Peer implements Serializable, RunActive {
  // CAN matter
  private Tracker tracker;
  private LongWrapper identifier;
  private Zone zone;
  private RouteManager routeManager;
  private DataManager dataManager;
  // Broadcast matter
  private int broadcastReceptionChecker;
  private boolean broadcastAlreadyReceived;
  ...
  @MemberOf("gettersOnImmutable")
  public LongWrapper getIdentifier() {
    return this.identifier;
  }
  ...
  public BooleanWrapper join(Peer p, int dimension) { ... }
  ...
  @MemberOf("dataManagement")
  public AddResponse add(AddQuery query) {
    AddResponse response;
    if (this.manages(query.getKey()).getBooleanValue()) {
      this.dataManager.store(query);
      response = new AddResponse(this.identifier);
    } else {
      query.addtraversedPeerID(this.identifier);
      response=this.routeManager.getPeerClosestTo(this.zone, query).add(query);
      response.addtraversedPeerID(this.identifier);
    }
    return response;
  }
  ...
  @MemberOf("broadcasting")
  public void broadcast(Key spatialConstraint, RoutingPair routingPair,
              int broadcastIteration, BroadcastRecoveryTest coordinator) {
    // Implementation of Algorithm 2
```

```
48   }
49   ...
50 }
```

Listing 6.1 – The Peer class of the fault tolerant broadcast application.

A peer has operations that are related to the construction of the CAN and its maintenance, as well as operations that are related to data management. These two concerns are respectively dealt with a `RouteManager` class and a `DataManager` class. More precisely, a peer can join the CAN, split its zone in order to make space for a new peer, add some data in its store, and return some data. Also, one major responsibility of a peer is to route the queries that are meant for other peers in the CAN.

The `Peer` class contains multiactive objects annotations, that allow some operations of a peer to be parallelised on two threads[4]. More precisely, four groups of requests are defined, for each potentially parallelisable concerns of the class. The `gettersOnImmutable` group gathers the methods that return the attributes of the `Peer` class. The returned attributes are immutable by convention according to the name of the group, like the identifier of the peer. This allows us to parallelise such getters with other operations of the `Peer` class. Similarly, the `monitoring` group regards peer dumping and does not interfere with getters. However, peer dumping cannot be performed when operations are done on the data store of the peer, which is the matter of the methods that are contained in the `dataManagement` and `broadcasting` groups. In our use case, broadcasting and adding data are independent actions, which is why we allow to perform them in parallel trough the annotations. Similarly, as we only add distinct data in the data store, concurrent adds are permitted, thanks to the specification of self compatibility in the `dataManagement` group. Adding data in the CAN is performed through routing from peers to peers. Thus, an add query is routed through asynchronous invocations of method `add` on successive peers (Line 41 of Listing 6.1). The implicit future that is created at this point (`response` variable) enables tracing the route that was taken by the add query. As in our system, only one broadcast operation can be realised at a time, we do not make the `broadcasting` group self compati-

[4] Since the devices that support a peer-to-peer system can be of various nature, we focus on commodity hardware that is assumed to have at least two parallel processing units.

ble. The implementation of the broadcast method reflects the efficient broadcast algorithm for CAN that we have introduced in Subsection 6.1.2. Other methods of the Peer class are not in any multiactive object group, like the join method in Listing 6.1. It means that the corresponding requests never execute other requests at the same time.

In addition to the Peer class and all its associated classes, we have a class that represents the entry point of the CAN: the Tracker class. The tracker is responsible for settling a new joining peer on a ProActive virtual node (see Section 5.1) and for picking a peer to join in the CAN. The tracker is also responsible for recording centralised data (or metadata), for example if all the peers have received an information in order to synchronise and start a new phase.

Our standard policy to build the CAN is deterministic: we split a peers' zone into two equal parts in a round robin manner, and we change the dimension to be split at each round. This policy eventually creates zones of equal size. The building policy is isolated such that other policies can be plugged easily. In our use case, the CAN is static, which means that once it is built, no new peers join the network nor existing peers leave the network. Also, when we build the CAN, we load it with an initial dataset. Our implementation of CAN together with the implementation of the use case represent 27 Java classes and roughly 2000 lines of code (including tests).

6.2.2 Recovery of an Efficient Broadcast in CAN

Our implementation of CAN uses some of the multiactive objects features and mechanisms that have been introduced this thesis. In particular, this implementation benefits from the fault tolerance mechanism that is the subject of Section 5.3. We propose in this subsection to see the fault tolerance mechanism in action, by forcing a crash during the realisation of an efficient broadcast algorithm in our CAN. We focus on a functional evaluation of the mechanism, which means that we are interested in the result of the application, i.e. whether a faulty execution gives the same result as a sane execution.

Compared to the previous description of our CAN implementation, we add to the application an external coordinator that manages the killing a ProActive vir-

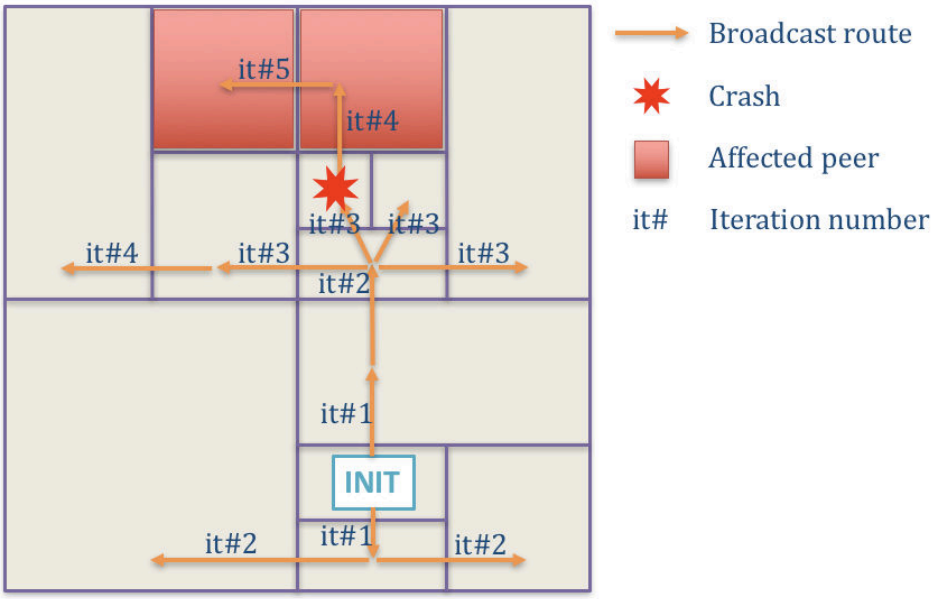


Figure 6.7 – Peer crash at hop number three.

tual node, and that checks the result of the application afterwards. The external coordinator is the only active object that is not fault tolerant in the application, so that when the coordinator kills a node, all active objects (peers and tracker) recover besides the coordinator. Therefore, the coordinator can monitor the application recovery after the fault and it can perform some sanity checks after the recovery.

The particular scenario we study is the following. We initially add 10 000 key-values pairs in a CAN in two dimensions such that keys are evenly distributed throughout the data space. In order to distribute data evenly, we create one key per possible discrete point of the data space, and associate a unique value of integer type. Once all initial data are loaded in the CAN (we synchronise at this point with a wait-by-necessity), we trigger an efficient broadcast (see Algorithm 2). The broadcast is started from a particular peer that we get from the tracker. The purpose of the broadcast is to add one to all the values that are initially stored in the CAN. At the same time as the broadcast is performed, we continue adding new data in the CAN, in parallel with the broadcast. Our application focuses on the case where only new data are added, therefore adding new data does not interfere with the modifications performed by the broadcast.

In order to simulate a failure of a peer during the broadcast, we explicitly crash a peer in the middle of the routing of the broadcast. The crash is triggered with an abrupt virtual node termination that is available in the ProActive library through a method. In order to drive the crash, we detect a particular iteration of the broadcast (in number of hops from the initiator), and we randomly crash one of the peers that receive the broadcast at this iteration. The broadcast is thus received but not forwarded by the crashed peer, as shown in Figure 6.7, where a peer located at the third iteration crashes (`it#3` in the figure). Consequently, all the peers that lie in the remaining of the broadcast route can no longer be reached by the broadcast message until the crashed peer recovers. For the sake of the test, we ensure that the last consistent recovery line is built during the broadcast, precisely before the crash happens and before the end of the broadcast. Indeed, in the other cases a successful broadcast would not bring any insight on the recovery mechanism, since either no peers or all the peers would have correctly received the broadcast when the system restarts. This is why it is important for our test that a consistent recovery line is built when some peers have received the broadcast message and some others not yet.

Our first objective is to perform a sanity check of the application's execution: we monitor the recovery of the application after the crash and we wait the re-execution to run to the end; we can then know if it was successful. The main steps of our methodology are summarised in five steps:

1. We build the CAN and we load it with initial data. We compute the sum of the data values and keep it for reference.

2. We start the efficient broadcast from a peer that is deterministically chosen. The purpose of the broadcast is to add one to all the values initially loaded. New data are inserted in the CAN in parallel with the broadcast execution. Meanwhile, checkpoints of the peers and of the tracker are taken.

3. We terminate a peer's virtual node in the middle of the broadcast, where some peers still need to be reached by the broadcast. The virtual node terminates which leads to a peer crash that is caught by our fault detector. The fault detector triggers the recovery mechanism.

4. We wait for the recovery to be completed. We observe the particular checkpoints at which the application restarts in order to check that they were actually taken after the beginning of the broadcast and before its end. This ensures the relevance of the scenario.

5. We check if the broadcast was successful (discussed below), and that the application continues to checkpoint normally after the recovery.

In order to evaluate the successfulness of the broadcast, we focus on three properties which allow us to determine the correctness of the broadcast in the presence of a fault during its execution. We mainly need to define these properties because the execution of our application is not fully deterministic. Indeed, the program uses the busy waiting mechanism of Java (`Thread.sleep`) in order to plan the crash of a peer. Thereby, the crash cannot be deterministically correlated with the progression of the application. Also, the checkpointing mechanism is buffered such that it cannot be performed too many times in a time frame. These two aspects have an impact on the number of peers that have been reached by the broadcast message before the crash. Therefore, the number of peers that need to be monitored for checking that they successfully received the broadcast in the re-execution can vary, and we have to adapt to this hazard in order to check the result of our application. In order to observe the behaviour of our distributed application, we chose the less intrusive coordination mechanism to trigger a crash and to collect information upon recovery, which leads to a not fully deterministic scenario.

We first verify that the broadcast message has been received by all the peers: *the recovery helped the application reaching its objectives.* In our case, the result of our application is correct if the final sum of all values initially loaded in the CAN is equal to the sum that was computed at the beginning plus the number of values. Secondly, we verify that the broadcast message has not been received twice: *the recovery did not introduce new messages.* In order to know about duplicates, we set a boolean value belonging to the peer's state when the broadcast request is executed (this value is checkpointed). By optimality of our efficient broadcast algorithm for CAN (in terms of number of messages), we know that, if a peer received the broadcast message more than once, it is not due to the broadcast algorithm but to an erroneous re-execution. Thirdly, we verify that new values that are added at the same time as the broadcast are correct after the broadcast: *the recovery did not introduce concurrency that entailed the data sanity.* This means that, for this execution, the checkpoints from which the peers recovered are sound. In the end, we successfully observed our scenario's execution and we verified the three properties above, that ensured a correct broadcast with fault tolerant peers.

* * *

In this chapter, we aimed at a practical setting for showing multiactive objects in action. To this end, we have developed an application in the context of peer-to-peer systems, and more specifically in the context of Content-Addressable Networks (CAN). CAN bases the network structure on geometrical properties that make any part of the network reachable in a bounded number of steps. We have first presented a contribution to the peer-to-peer ecosystem with an efficient broadcast algorithm that specially operates in CAN. This contribution settled the context of the application that we have developed in a second section. The application combines the efficient broadcast algorithm for CAN and the fault tolerance mechanism of multiactive objects introduced in Section 5.3. Multiactive objects are particularly adapted to represent the peers of a peer-to-peer system: they are independent, they can communicate asynchronously, and they can perform multiple tasks at the same time. Moreover, all of these aspects can be represented in a single class and driven with multiactive object annotations, since all peers behave similarly. Conversely, peer-to-peer systems represent challenging settings to test the effectiveness of our middleware: the nodes of a peer-to-peer system are by nature unreliable, which brings the need for fault tolerance.

We have presented a use case where the fault tolerance of a distributed broadcast is not embedded in the broadcast algorithm but rather carried by the middleware, in our case ProActive. This is an approach that is rarely explored when dealing with fault tolerance of distributed algorithms. In the context of distributed broadcast, duplication of messages can be seen as a form of fault tolerance. However, duplicates are often uncontrollable for most of the distributed broadcast algorithms. Consequently, no one can ensure that a duplicate will be produced for a failed message. For example, in the case of M-CAN [Rat+01b], only a small ratio of duplicates are produced, so they are not reliable for fault tolerance. Furthermore, if duplicates can serve to fault tolerance, one should still implement this non-trivial behaviour efficiently. In the context of peer-to-peer systems, the notion of fault tolerance is often directly embedded in the system specification. This is the case of Chord [Sto+01], that delegates the queries for a crashed peer to another peer that is responsible for relaying the crashed peer. A similar approach also exist in CAN [Rat+01a]. The advantage is that the management of faults can then be adapted to any underlying implementation of the peer-to-peer structure, and also be optimised at will. However, we argue that the support of a distributed system is a middleware matter: in our case it allows the business of the peer-to-peer system to be less entangled with its implementation, respecting a clear separation of concerns. Indeed, the philosophy of peer-to-peer is to provide a multitude of light but efficient nodes, so charging peers with more and more concerns can end up in entailing the scalability of the peer-to-peer system.

To conclude, we have shown in this chapter that using the non functional features of the ProActive middleware is a relevant approach for building scalable and reliable distributed systems. Multiactive objects have proven to be well-adapted to implement the presented application, and to support its faulty execution. The peer-to-peer application that was proposed in this chapter gathered several challenges that we have tackled throughout this thesis, and thus prepared us for the conclusion of the thesis.

Chapter 7

Conclusion

Contents

7.1 Summary

7.1.1 Methodology

For a decade, active objects and actors have represented safe concurrent computing, coming in an intuitive abstraction for the programmer. Lately, they have become more relevant in the parallel era thanks to successive enhancements made to the models and to the frameworks, such that active objects and actors also tend now to be highly efficient locally. However, the increments that have been proposed have raised new questions, such as the best way to express local parallelism and local data sharing. They have also raised new challenges, such as the adaptation of the local scheduling of requests. The tools that were built around the model also have to be re-thought. In this thesis, we study the active object programming model raised with internal parallel execution, controlled through an intuitive meta-programming model: the multiactive object programming model. We bring a complete framework that offers advanced scheduling policies, debugging, and fault tolerance, integrated in a library. Overall, we attach a particular interest in three objectives: usability, correctness, and performance of our developments. These objectives cover the broad spectrum of a programming model: design, specification, and execution. We give the guidelines of the programming model's usage through practical applications. Testing our developments in realistic settings is important for us, because being efficient is part of our objectives, and the fact that we target a distributed execution of applications makes their evolution less predictable. This is why we systematically evaluate the solutions we propose with experiments. On the other hand, we use a formalisation of our developments to deal with their correctness and to highlight their properties. By this mean, we believe that we reinforce the guarantees offered by the programming model, on which the programmer can strongly rely on.

7.1.2 Contributions

In this thesis, our first contribution is to come with advanced scheduling mechanisms that are integrated with the multiactive object programming model, and implemented in the ProActive library. The concerns of local parallelism cannot be completely handled by default policies, especially for the applications that target a high performance. This is why we offer convenient abstractions to deal with the parallelised execution of multiactive objects. We give to the programmer a new set of annotations that affect the priority of requests and the allocation of threads. Behind the scene, the request scheduler interprets these annotations in order to select the next request to execute. We expose the properties of the priority mechanism so that the behaviour of the program that derived from the programmer's annotations is easily predictable and intuitive. For example, the programmer must be aware that the priorities will be applied only over the requests that are compatible. We also experiment it with micro-benchmarks and we expose this way the advantages and the drawbacks of representing the priorities with a priority graph. We propose a solution to cache the priority relationship between groups of requests so that the scheduling of requests can be accelerated.

Priorities and thread management, combined with multiactive object compatibilities, reveal to be convenient to implement internal scheduling patterns. We use these features in a second and in a third part of the thesis. We encode the ABS language, that is based on active objects, into a precise setting of multiactive objects. We release the ProActive backend for ABS, that automatically transforms ABS models into distributable applications. We formalise the translation in order to prove its correctness. To this end, we use MultiASP (the calculus of ProActive) together with the operational semantics of ABS. We establish an equivalence relation between ABS and MultiASP configurations. We prove two theorems that corroborate the correctness of the translation. Despite the expressiveness that allows MultiASP to embody the ABS language, some configurations cannot be properly represented. Notably, the future update mechanism is different in the two languages: a future resolution is not remarkable in a MultiASP program, whereas it basically drives an ABS program. Such an event cannot be directly observed in MultiASP, because MultiASP relies on a data-driven synchronisation. This thesis provides a thorough study of active object languages and their implementations, and the backend that we presented in the case of ABS and ProActive/MultiASP illustrate those differences.

A third contribution covers the support of applications that are build with the multiactive object programming model. More precisely, we take interest in the development phase of such applications and in their execution phase. We propose a debugger tool that allows the programmer to visualise the execution of a ProActive application and see what happened in parallel in multiactive objects. This is the first visualisation tool that is in accordance with the notions of multiactive objects. Regarding the support of the execution of multiactive object-based applications, we also come with an adapted fault tolerance protocol that allows faulty multiactive objects to recover after a crash, thanks to regular checkpoints. We generically implement a seminal version of this protocol by only relying on multiactive object abstractions. This makes a clear separation between the middleware and the fault tolerance protocol.

Finally, we complete our developments with a concrete application, that takes place in the particular setting of peer-to-peer systems. We focus on Content-Addressable Networks (CAN), and more particularly on an efficient way of broadcasting information in such networks. We implement a CAN with multiactive objects for embodying fault tolerant parallel peers. We play a scenario where a peer runs an efficient broadcast, and where one of the peer crashes during the routing of the broadcast. We rely on the fault tolerance mechanism adapted to multiactive objects in order to enable the recovery of the faulty peer. The broadcast can thus complete successfully even in the presence of a fault during its execution. This constitutes a valid approach to implement robust distributed algorithms at the middleware level.

Overall, we build a proven and thorough framework around multiactive objects. We implement scheduling patterns and non functional features by using this programming model. We tackle in this thesis the questions raised by a multithreaded execution of active objects: the efficient local scheduling of requests, the management of parallel execution flows with threads, and the support made to the framework.

7.2 Perspectives

7.2.1 Impact and Short-Term Enhancements

The results we have obtained and presented in this thesis are promising for the multiactive object programming model, both on practical aspects and on theoretical aspects. As it is now, the multiactive object programming model and the work we have done around it is a suitable starting point of several research topics, that we will discuss in the next subsections. The first impact of this thesis is immediate: the multiactive programming model and all the the works that are presented in this thesis are completely implemented and usable. A broader impact of this thesis is enclosed in the deep understanding of active object programming languages. It can serve as a guideline on the various ways to design and implement active object models and frameworks. In a short-term perspective, the ProActive middleware and its surrounding tools can be further enhanced. Regarding the set of multiactive object annotations that have been introduced in this thesis, having dynamic priorities should be considered in future work. For now, the priorities that are given to the requests of a multiactive object are static: they are defined by the programmer once and they apply the whole execution time. In some situations, following the prioritised specification of the programmer could lead to a starvation of requests, for example if a continuous flow of high priority requests takes over low priority requests. In this case, the multiactive object runtime should be able to detect the starvation of low priority requests and to adopt a behaviour that solves this issue. One of the possible solutions to implement this behaviour would be to make the multiactive object runtime increase progressively the priority of low priority requests, up to the point where the starvation is solved. This is an approach that is used to handle the priority of tasks in operating systems. Indeed, the starvation of requests is a situation that the beginner programmer is unlikely to foresee. This is why dynamic priorities by default would be relevant in the multiactive object framework. In order to implement this behaviour, a priority Application Programming Interface (API) can be created, like for the manipulation of the thread limit. In general, more systematic policies and guardrails like this can be introduced in the middleware to prevent unexpected executions. Considering the large possibilities of multiactive objects, such policies could likely be implemented using the abstractions offered by multiactive objects, in the same spirit as we have driven most of our developments in this thesis.

7.2.2 Fault Tolerance and Debugging

Short and mid-term works will be carried out in the debugging and fault tolerance of multiactive objects. We first need to augment the fault tolerant protocol of multiactive objects in order to make it work in larger settings of multiactive objects. Indeed, currently the proposed protocol creates a deadlock if there are recursive requests when the checkpoint is triggered. We proposed many directions to solve this issue in Subsection 5.3.3. Most of them imply a core modification of the multiactive object runtime, but we advocate one of the solutions that elegantly uses the generic multiactive object notions. To this end, the parameter of multiactive object annotations that activates the reentrance of requests must be set: the requests that generate new requests on the same multiactive object are treated on the same thread. This solves the deadlocks that arise from having a checkpoint triggered at the same time as a reentrant request is processed. However, when this solution is implemented in the future, one should analyse if having this configuration does not create a side effect on the execution of the application's requests.

Regarding the tool set built around multiactive objects, the debugger tool could provide even more information regarding an application's execution. For example, the kind of thread limit is an information that is missing in the current visualisation, as well as the dynamic adaptation of the thread limit. Another promising improvement of the debugger tool is to support runtime visualisation, with instantaneous updates on the application's status. This is possible thanks to the efficient parsing of events that we provide.

A mid-term perspective that is conceivable is to provide a real-time debugging of multiactive object-based application with breakpoints. Indeed, the debugging of applications through breakpoints can be done by persisting the state of the application at the breakpoint's place. Then, continuing the execution from the breakpoint is like restarting the application from a persisted state. The fault tolerance mechanism of multiactive objects can be used in order to achieve this behaviour. Indeed, the state of the application is regularly persisted with checkpoints. Thus creating breakpoints from checkpoints can be made by raising the checkpointing action from the middleware level to the application level. To this end, the fault tolerance protocol for multiactive objects must be strengthened beforehand, and the solutions presented in Subsection 5.3.3 could be implemented in this objective.

7.2.3 Multiactive Components

VerCors[1] is a platform that allows the programmer to graphically specify distributed component-based applications. VerCors enables analysis, verification and

[1] https://team.inria.fr/scale/software/vercors/vcev4-download/

validation of applications by generating a behavioural model from the specification of an application, which is then suitable for a model checker [HKM16]. The communication model that prevails in VerCors is based on asynchronous requests and replies. Executable code can be automatically generated from the specification. In particular, VerCors generates ProActive code behind the scene. Indeed, the ProActive library provides components; they are implemented with ProActive active objects. In this context, a perspective that arises from this thesis is the integration of the advanced features of multiactive objects in multiactive components. This objective regards the evolution of the ProActive library, but also of the VerCors platform. In the future, it is planned to make the multiactive features accessible from the graphical component editor. More details about the evolution of the VerCors platform and of the usage of the ProActive library in the code generation will be available in the forthcoming thesis of Oleksandra Kulankhina.

7.2.4 Formalisation and Static Analysis

Besides practical developments regarding multiactive objects, debugging, and fault tolerance, a lot of theoretical works can be grounded on the proposed multiactive object framework. Firstly, the work that has been started for the fault tolerance of multiactive objects needs to be reinforced. The formalisation of the fault tolerance protocol can help in its design, and help in defining the guarantees that the protocol should provide. Doing this work should also help in reasoning, more globally, on the fault tolerance of distributed systems that involve local concurrency. Moreover, formalising the fault tolerance protocol of multiactive objects would build a stronger basis for the design and implementation of a dynamic debugger through breakpoints.

Another orientation that the multiactive object framework will probably take in the future is the static analysis of programs made of multiactive objects. For now, the multiactive object annotations that are given by the programmer are interpreted at runtime. Apart from a few guardrails that we have set when the annotations are interpreted, the annotations are not verified, and we trust the programmer for most of the behaviour that is driven by the annotations. A refined analysis of multiactive object annotations could provide a stronger confidence in the application's execution. For example, being able to validate the compatibilities given by the programmer with non-interference techniques (in order to spot data races) would give a precious feedback to the programmer.

Finally, an on going work is pursued on the static detection of deadlocks in multiactive object-based applications. The complete technique and results will be available in the thesis of Vincenzo Mastandrea. The context of this work starts from and effective deadlock analyser for the ABS language [GLL15]. However, performing deadlock analysis in the case of a transparent active object language like ProActive and MultiASP raises new challenges. The current results on this work [Gia+16] include an effective parser for the MultiASP programming language, and a seminal adaptation of the deadlock detection algorithm of ABS. The outcome of this work could be particularly relevant in the case of the ProActive backend for ABS, that we have introduced in this thesis, in order to confront the deadlocks that can be found in the ABS source program and in the translated ProActive program.

The synergy of these works, specially focused on active object languages, builds solid bridges and opens long-term collaborations with all the major players of active object-based programming languages and models.

Appendix A

Proofs

A.1 Proofs of Lemmas

Lemma 4.3.1, page 131 (Correspondence of activated objects). In an ABS configuration, if an object *ob* has an active request that is not `idle`, then there exists a COG in which *ob* is the current active object.

$$ob(i\_\alpha, \overrightarrow{x \mapsto v}, p, q) \in Cn \wedge p \neq \texttt{idle} \Rightarrow cog(\alpha, i\_\alpha) \in Cn$$

Proof of Lemma 4.3.1. The proof is done by induction on the ABS reduction rules.

□

Lemma 4.3.2, page 131 (Equivalence of values).

$$v \approx_{\sigma}^{Cn} v' \Rightarrow v \approx_{\sigma}^{Cn} [\![v']\!]$$

Proof of Lemma 4.3.2. We prove this lemma by recurrence on the proof of $v \approx_{\sigma}^{Cn} v'$ (Definition 2). This corresponds to a recurrence on the number of indirections of references that are followed to evaluate v'. We consider three cases:

- If v is a primitive value (case $v \approx_{\sigma}^{Cn} v$), then $v' = v$ and the equivalence $v \approx_{\sigma}^{Cn} v$ is direct because $v = [\![v]\!]$.

- If v is not a primitive value ($\dfrac{v \approx_\sigma^{Cn} \sigma(o)}{v \approx_\sigma^{Cn} o}$), then $v' = o$ and $v \approx_\sigma^{Cn} \sigma(o)$. Therefore, we need to handle three cases:
  - $\sigma(o) = f$. In this case, since $[\![o]\!] = o$ by the evaluation function of Figure 4.9, then $v \approx_\sigma^{Cn} o$.
  - $\sigma(o) = [\overrightarrow{x \mapsto v}]$. In this case, since the evaluation of o is also equal to o, we also have $v \approx_\sigma^{Cn} o$.
  - $\sigma(o) = v''$. In this case, since $v \approx_\sigma^{Cn} v''$, we have $v \approx_\sigma^{Cn} [\![v'']\!]$ by recurrence hypothesis. Then $v \approx_\sigma^{Cn} [\![o]\!] = [\![\sigma(o)]\!] = [\![v'']\!]$ by the evaluation function of Figure 4.9.
- The other cases of Definition 2 are not applicable here, because they are not a valid MultiASP value (e.g. a future is not a MultiASP value).

□

Lemma 4.3.3, page 131 (Equivalence of evaluation functions). Let Cn be an ABS configuration and suppose $Cn \,\mathcal{R}\, \underline{Cn}$. Let $ob(o\_\alpha, a, \{l|s\}, q) \in Cn$. By definition of $\mathcal{R}$, there exists a single activity $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$, with $\sigma(o) = [\,cog \mapsto \alpha, myId \mapsto i, a']$ and $(f, execute, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \wedge \ell'(\texttt{this}) = o$. For any ABS expression e we have:

$$[\![e]\!]^A_{a+l} \approx_\sigma^{Cn} [\![e]\!]_{\sigma+\ell'}$$

Proof of Lemma 4.3.3. We only consider here the case where x is a variable, because the other cases are not different from what is done in the Java backend for ABS. In particular, the evaluation of arithmetic expressions relies on the equivalence of variables and on the fact that the evaluation of the arithmetic expression itself is the same.

We prove the equivalence of evaluation functions on variables by case analysis.

- Either the variable belongs to the fields of the object $ob(o\_\alpha, a, \{l|s\}, q)$, and not to the local variables: $a(x) = v \wedge v \notin dom(l)$. In this case, we have $ob(o\_\alpha, a, \{l|s\}, q) \in Cn$ with $x \mapsto v \in a$. By definition of equivalence, we have $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$, $\exists(f, execute, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} \in p$ with, $\forall x \in dom(l)\backslash destiny$, $l(x) = v \approx_\sigma^{Cn} \ell'(x) = v'$ by Line 8 of equivalence condition, and $\sigma(o) = [mycog \mapsto \alpha, myId \mapsto i, a']$ by Line 2. Also by definition of equivalence, $a'(x) = v'$ and $v \approx_\sigma^{Cn} v'$, because σ is the store of α. We have $[\![x]\!]_{\sigma+\ell'} = [\![v']\!]$ by the definition of the evaluation function of Figure 4.9, with $\ell'(this) = o$ by the equivalence condition of Figure 4.12 (because $x \notin dom(l)$ and hence, $x \notin dom(\ell')$). Thus we have $v \approx_\sigma^{Cn} [\![x]\!]_{\sigma+\ell'} = [\![v']\!]_{\sigma+\ell'}$ by Lemma 4.3.2.

– Or the variable belongs to local variables: $l(x) = v$. In this case, we have $ob(o\_\alpha, a, \{l|s\}, q) \in Cn$ with $l(x) = v$ and, by definition of equivalence, $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$, $\exists(f, execute, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} \in p$ with, $\forall x \in dom(l)\backslash destiny$, $l(x) = v \approx_\sigma^{Cn} \ell'(x) = v'$ by Line 8 of equivalence condition. Finally, $[\![x]\!]_{a+l}^A \approx_\sigma^{Cn} \ell'(x)$ and by Lemma 4.3.2 $[\![x]\!]_{a+l}^A \approx_\sigma^{Cn} [\![\ell'(x)]\!]_{\sigma+\ell'}$.

□

Lemma 4.3.4, page 131 (Equivalence after serialisation and renaming).

$$v \approx_\sigma^{Cn} v' \wedge \sigma' = \text{serialise}(\sigma, v') \Rightarrow v \approx_{\sigma'}^{Cn} v'$$
$$\overline{v} \approx_\sigma^{Cn} \overline{v'} \wedge (\overline{v''}, \sigma') = \text{rename}_\sigma(\overline{v'}, \sigma) \Rightarrow \overline{v} \approx_{\sigma'}^{Cn} \overline{v''}$$

Proof of Lemma 4.3.4. The proof relies mostly on the definition of equivalence $\mathcal{R}$. Concerning serialisation, the main difference is the fact that we check the equivalence in a smaller (or equal) store. However, this store is self-contained and every reference that could be followed by the equivalence always exist in the serialised store. Concerning renaming, the proof is trivial as only the case where the references are followed is changed and the renaming ensures the equivalence of the reached values. □

Invariant Reg, page 131. *For every activity α such that o_α is the location of the active object of activity α in its store σ_α, if the current task consists in an invocation to o_α.retrieve(i), then this invocation always succeeds, because the object has been registered first. The invocation returns some object o' such that $i\_\alpha \approx_\sigma^{Cn} o'$.*

Proof of Invariant Reg. To prove the **Invariant Reg** we rely on the translational semantics provided in Figure 4.11. Suppose that a call to $o_\alpha.\mathit{retrieve}(i)$ is performed. We want to show that this call succeeds and returns an object equivalent to i_α. If we have such a call, it means that we are in the body of an $\mathit{execute}(id, m, \overline{parameters})$ call, with $id = i$. Since the *execute* method is a reserved method name, we know that we are executing a remote method call, corresponding to an expression: $e.m(\overline{parameters})$ with $e.\mathit{cog}() = \alpha$ and $e.myId() = id$. Also, we have e such that $[\![e]\!]_{\sigma_\beta+\ell} = o$ and $\sigma_\beta(o) = [\mathit{cog} \mapsto \alpha, myId \mapsto id, \cdots]$. Then, we know that we cannot have method calls on the temporary variable *no*, that is used to temporarily store a newly created object until it is registered to its COG, because this variable is only known in the instantiation thread and no method call is performed on it. Thus, since e cannot be *no*, it is necessarily a variable x that has been assigned after a call to $t.\mathit{register}(no)$ (where $t \mapsto \alpha$), that makes the COG t aware of the new object referenced by *no*. Consequently, the call to *register* precedes the remote call $e.m(\overline{parameters})$[1]. Now, we have two cases.

- Either the *register* call is local because it comes with a new local object instantiation (with the `new local` ABS keyword). In this case, since the *register* call is synchronous, it is necessarily finished before $e.m(\overline{parameters})$ is invoked, so the later $o_\alpha.\mathit{retrieve}(i)$ succeeds naturally.

- Or the *register* call is remote because it comes with a new remote object instantiation (with the `new` ABS keyword). In this case, since the *register* call is asynchronous, it may happen that the *register* and the *execute* requests are both awaiting in the queue. However, thanks to causal ordering and to FIFO service policy, we are guaranteed that the *register* request is in the queue before the *execute* request, and that the *register* request will be served first. In addition, considering the multiactive compatibilities defined for the groups of *register* and *execute*, we see that an incompatibility is raised between the two requests because they access the same identifier. Therefore, the $o_\alpha.\mathit{retrieve}(i)$ request cannot be served before the *register* request finishes, and thus, the $o_\alpha.\mathit{retrieve}(i)$ that is performed in the *execute* request succeeds.

[1] We also use the fact that a pointer could not be forged.

Finally, in both cases we have: $i\_\alpha \approx_{\sigma}^{Cn} o'$. □

A.2 Proofs of Theorems

A.2.1 From ABS to MultiASP

Theorem 4.3.5, page 132 (ABS to MultiASP). The translation simulates all ABS executions with FIFO policy and rendez-vous communications, provided that no future value is a future reference.

$$Cn_0 \overset{\text{A}}{\to}{}^{*}\ Cn \text{ with a FIFO policy } \wedge \nexists f, f'.\, \mathit{fut}(f, f') \in Cn \ \Rightarrow\ \exists \underline{Cn}.\ \underline{Cn_0} \to^{*} \underline{Cn} \wedge Cn\, \mathcal{R}\, \underline{Cn}$$

Proof of Theorem 4.3.5. Firstly, we look at the starting configuration. Let $P = \overline{IC}\ \{\overline{x\ ;}\ s\}$ be an ABS program, let Cn_0 be the initial configuration corresponding to this ABS program: $ob(start, \varnothing, p, \varnothing)$, where the process p corresponds to the activation of the program's main block: $p = \{l|s\}$. The initial MultiASP configuration corresponding to program $[\![P]\!]$ is: $\textsc{act}(\alpha_0, o, [o \mapsto \varnothing], q_0 \mapsto \{l|[\![s]\!]\}, \varnothing)$. It is easy to see that this initial configuration has the same behaviour as: $\textsc{act}(\alpha_0, o, [o \mapsto \varnothing, start \mapsto [cog \mapsto \alpha_0, myId \mapsto 0], (f, execute, start, m) \mapsto \{l|[\![s]\!]\} :: \{\varnothing|x = \bullet\}, \varnothing)$. We denote this new MultiASP initial configuration $\underline{Cn_0}$. This way, we have the ABS initial configuration Cn_0 that is equivalent, considering our equivalence relation $\mathcal{R}$, to the MultiASP configuration $\underline{Cn_0}$; this fact is denoted $Cn_0\, \mathcal{R}\, \underline{Cn_0}$. We also use $\to^{*}$ to denote the transitive closure of $\to$.

Secondly, Theorem 4.3.5 is proven by induction on the ABS reduction rules. It relies on the definition of equivalence $\mathcal{R}$ shown in Figure 4.12, and on the following induction step: If $Cn_0 \overset{\text{A}}{\to}{}^{*}\ Cn$, $Cn\, \mathcal{R}\, \underline{Cn}$, $Cn \overset{\text{A}}{\to} Cn'$ with a FIFO policy, and $\nexists f, f'.\, \mathit{fut}(f, f') \in Cn'$, then $\exists \underline{Cn'}.\ \underline{Cn} \to^{*} \underline{Cn'} \ \wedge\ Cn'\, \mathcal{R}\, \underline{Cn'}$. First, note that in the definition of equivalence between values and between statements ($\approx_{\sigma}^{Cn}$), the ABS configuration Cn is only used to track futures. Thus, in all the cases except Return, $\approx_{\sigma}^{Cn}$ and $\approx_{\sigma}^{Cn'}$ are the same. Each case is concluded by arguments on observability. We are interested in observing remote method invocations, return of asynchronous method calls, assignments to field variables, and assignments to local variables that were not introduced by the translation into MultiASP. Other reduction rules are considered to be not observable and will be used transparently as many times as necessary to simulate any ABS reduction.

1) Case of the [Assign-Local] ABS rule.

We first consider the ABS ASSIGN-LOCAL rule, to show that we can observe equivalent configurations on ABS and MultiASP side after reduction. In this rule, a particular ABS configuration Cn leads to a configuration Cn', noted $Cn \overset{\text{A}}{\rightarrow} Cn'$, as follows:

ABS

$$\frac{x \in \text{dom}(l) \quad v = [\![e]\!]^{A}_{(a+l)}}{ob(i\_\alpha, a, \{l \mid x = e; s\}, q) \overset{\text{A}}{\rightarrow} ob(i\_\alpha, a, \{l[x \mapsto v] \mid s\}, q)}$$

Suppose that Cn is also related to a MultiASP configuration $\underline{Cn}$ by the definition of equivalence, noted $Cn\,\mathcal{R}\,\underline{Cn}$. Then, we want to show that a configuration $\underline{Cn'}$, equivalent to Cn', can be obtained by MultiASP semantics from $\underline{Cn}$. In the case of ASSIGN-LOCAL rule, it means that a local variable x must exist in $\underline{Cn'}$ and that its value after assignment must be equivalent to the value assigned to it in ABS. To begin with, by Lemma 4.3.1 there must be a COG in the ABS starting configuration where i_α is the current active object: $\exists cog(\alpha, i\_\alpha) \in Cn$. Then, by definition of the equivalence relation $\mathcal{R}$, we have the following key points:

- First, there exists a corresponding MultiASP activity in the MultiASP starting configuration: $\exists o_\alpha, \sigma, p, Rq.act(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$.

- Second, this activity has an active request initiated from an *execute* method call, and maps to a current statement to execute: $\exists f, i, m, \overline{v''}, \ell', s', \ell'', s''.((f, execute, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p$ with $(x = e; s) \approx^{Cn}_{\sigma} s'$ and $\forall x \in dom(l) \smallsetminus destiny.l(x) \approx^{Cn}_{\sigma} \ell(x)$. Thus, by the definition of equivalence between statements, either $s' = (x = e; [\![s]\!])$, or $e = v$ (because ABS statement is $x = v; s$) and $s' = (x = e'; [\![s]\!])$ with $v \approx^{Cn}_{\sigma} [\![e']\!]_{\sigma+\ell'}$. In both cases, there is e' such that $s' = [x = e'; [\![s]\!]]$

Those key points allow us to find the starting MultiASP configuration $\underline{Cn}$ and to apply the MultiASP ASSIGN-LOCAL reduction rule:

MultiASP

$$\frac{x \in \text{dom}(\ell') \qquad v' = [\![e']\!]_{(\sigma+\ell')}}{\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \rightarrow \qquad \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell'[x \mapsto v'] \mid [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p', Rq)}$$

We have now to verify that the obtained MultiASP configuration Cn' is in relation $\mathcal{R}$ with the ABS configuration $\underline{Cn'}$. First, we must have at least as many corresponding terms in MultiASP as in ABS. In this case, we have $cog(\alpha, i\_\alpha) \in Cn'$ on one hand, and $\textsc{act}(\alpha, o_\alpha, \sigma, p'', Rq) \in \underline{Cn'}$ on the other hand. Recall that we only consider the terms that changed in the reduction, not all the terms of a configuration. Second, we must check equivalence of each element by checking the equivalence of their content. We have on the ABS side the term $ob(i\_\alpha, a, \{l[x \mapsto v] | s\}, q)$ and on the MultiASP side the activity α which has in particular $q_A \mapsto \{\ell'[x \mapsto v'] | \llbracket s \rrbracket\} :: \{\ell'' | s''\}$ in the method execution stack, with $v' = \llbracket e' \rrbracket_{\sigma+\ell'}$. Then, to have $v \approx_\sigma^{Cn'} v'$ we must have $v \approx_\sigma^{Cn'} \llbracket e' \rrbracket_{\sigma+\ell'}$. Now, we have two cases:

1. either $e' = e$ by definition and therefore, by Lemma 4.3.3: $\llbracket e \rrbracket_{a+l}^A \approx_\sigma^{Cn'} \llbracket e \rrbracket_{\sigma+\ell'}$.
2. or $v \approx_\sigma^{Cn'} \llbracket e \rrbracket_{\sigma+\ell'}$ and $\llbracket e' \rrbracket_{\sigma+\ell'} = v'$.

Both cases lead to the fact that the assigned values are equivalent. The rest of the elements of the activity are unchanged except the remaining statements, but in this case we have $s \approx_{\sigma+\ell'}^{Cn'} \llbracket s \rrbracket$ by definition of equivalence.

2) **Case of the [Assign-Field] ABS rule.**

Like the previous case, suppose $Cn \xrightarrow{\text{A}} Cn'$ with the Assign-Field ABS reduction rule, as follows:

$$\frac{x \in \text{dom}(a) \setminus \text{dom}(l) \quad v = \llbracket e \rrbracket_{(a+l)}^A}{ob(i\_\alpha, a, \{l \mid x = e; s\}, q) \xrightarrow{\text{A}} ob(i\_\alpha, a[x \mapsto v], \{l \mid s\}, q)} \text{ABS}$$

With the same strategy as the previous case, namely by having first $Cn \, \mathcal{R} \, \underline{Cn}$, then $\underline{Cn} \to \underline{Cn'}$ and finally $Cn' \, \mathcal{R} \, \underline{Cn'}$, we want to show that an object field x exists in $\underline{Cn'}$ and that its value after assignment is equivalent to the value assigned to it in ABS. First, we have again by Lemma 4.3.1, $\exists cog(\alpha, i\_\alpha) \in Cn$. Second, by definition of the equivalence relation $\mathcal{R}$, we have:

– A corresponding MultiASP activity: $\exists o_\alpha, \sigma, p, Rq.act(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$.

– Values in the store σ that are equivalent to the values contained in a: $\exists o, \overline{v'}.\sigma(o) = [cog \mapsto \alpha, myId \mapsto i, \overrightarrow{x \mapsto v'}]$ where $\overline{v} \approx_\sigma^{Cn} \overline{v'}$ and $\overrightarrow{x \mapsto v} = a$. Consequently, $x \in dom(\sigma(o))$ because $x \in dom(a)$.

- A statement currently executed that belongs to the current active request of the activity: $\exists f, i, m, \overline{v''}, \ell', s', \ell'', s''$ such that: $(f, \mathit{execute}, i, m, \overline{v''})_A \mapsto \{\ell' | s'\} :: \{\ell'' | s''\} \in p$ and $\ell'(\texttt{this}) = o$ with $(x = e; s) \approx_{\sigma}^{Cn} s'$. Like the previous case, either $s' = (x = e; [\![s]\!])$ and $e' = e$, or $e = v$ and $s' = (x = e'; [\![s]\!])$ with $v \approx_{\sigma}^{Cn} [\![e']\!]_{\sigma+\ell'}$.

Then, this leads to the following MultiASP Assign-Field rule:

MultiASP

$$\frac{\ell'(\mathit{this}) = o \qquad x \in \mathrm{dom}(\sigma(o)) \qquad x \notin \mathit{dom}(\ell') \qquad \sigma' = \sigma[o \mapsto (\sigma(o)[x \mapsto [\![e']\!]_{(\sigma+\ell')}])]}{\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \to \qquad \textsc{act}(\alpha, o_\alpha, \sigma', \{q_A \mapsto \{\ell' \mid [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p', Rq)}$$

Like in the previous case, we must now check that $Cn' \mathcal{R} \underline{Cn'}$. First, we have $\mathit{cog}(\alpha, i\_\alpha) \in Cn'$ and $\textsc{act}(\alpha, o_\alpha, \sigma', p'', Rq) \in \underline{Cn'}$, which are equivalent terms. Second, we have to compare the content of $\mathit{ob}(i\_\alpha, a[x \mapsto v], \{l \mid s\}, q)$ to the content of activity α with $\sigma(o) = [\mathit{cog} \mapsto \alpha, \mathit{myId} \mapsto i, \overrightarrow{x \mapsto v}[x \mapsto [\![e']\!]_{\sigma+\ell'}]]$. Then, we must have $\overline{v} \approx_{\sigma}^{Cn'} \overline{v'}$, which is unchanged except for the particular value v that must be equivalent to the evaluation of e'. Then, similarly to the Assign-Local case, we have again $v \approx_{\sigma}^{Cn'} [\![e']\!]_{\sigma+\ell'}$. Finally, we have $s \approx_{\sigma+\ell'}^{Cn'} [\![s]\!]$ by the definition of equivalence $\mathcal{R}$.

3) Case of the [Await-True] ABS rule.

We start from the ABS Await-True reduction rule, in which $Cn \xrightarrow{\mathrm{A}} Cn'$:

ABS

$$\frac{f = [\![x]\!]_{(a+l)}^{A} \qquad v \neq \bot}{\mathit{ob}(i\_\alpha, a, \{l \mid \texttt{await}\, x\,\texttt{?}; s\}, q)\ \mathit{fut}(f, v) \xrightarrow{\mathrm{A}} \mathit{ob}(i\_\alpha, a, \{l \mid s\}, q)\ \mathit{fut}(f, v)}$$

From the configuration Cn, we know that, by Lemma 4.3.1, there exists in ABS α such that $\mathit{cog}(\alpha, i\_\alpha) \in Cn$. We also know, by equivalence relation $\mathcal{R}$ and by translational semantics, that there exist in MultiASP o_α, σ, p, Rq such that $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$. Besides, there exist ℓ', s', ℓ'', s'' such that, by translational semantics: $\{q_A \mapsto \{\ell' \mid w = x.\mathit{get}(); [\![s]\!]\} :: \{\ell'' \mid s''\}\} \in p$. Also, we know that there are v', σ' such that $\textsc{fut}(f, v', \sigma') \in \underline{Cn}$, with $v \approx_{\sigma'}^{Cn} v'$. Thus, we have the following MultiASP configuration $\underline{Cn}$, where $Cn \mathcal{R} \underline{Cn}$:

MultiASP

$$\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell' \mid w = x.\mathit{get}(); [\![s]\!]\} :: \{\ell'' \mid \mathsf{s}''\}\} \uplus p', Rq) \quad \textsc{fut}(f, v', \sigma')$$

By Lemma 4.3.3, we have $[\![x]\!]^A_{(a+l)} \approx^{Cn}_{\sigma} [\![x]\!]_{(\sigma+\ell')}$, and, by case analysis on the definition of $\approx^{Cn}_{\sigma}$ (Definition 2), we have $[\![x]\!]_{(\sigma+\ell')} = o$. Then, two cases are possible in MultiASP: either $\sigma(o) = f$ (fourth case and second case of the definition of $\approx^{Cn}_{\sigma}$), or $\sigma(o) = v_f$ where $v \approx^{Cn}_{\sigma} v_f$ (fourth case and fifth case of the definition of $\approx^{Cn}_{\sigma}$). We consider the two possible cases below:

- In the first case, where o points to a future, we first perform a future update on the MultiASP configuration $\underline{Cn}$ (the point here is to 'catch up' with the ABS execution). Thus, we have $\underline{Cn} \to \underline{Cn'}$ where in $\underline{Cn'}$ activity α is: $\textsc{act}(\alpha, o_\alpha, \sigma'', \{q_A \mapsto \{\ell' \mid w = x.get(); [\![s]\!]\} :: \{\ell'' \mid \mathsf{s}''\}\} \uplus p', Rq)$ with $\sigma'' = \sigma[o \mapsto v_r] \cup \sigma_r$ and $(v_r, \sigma_r) = \text{rename}_{\sigma}(v', \sigma')$. After that, the current local method call $(f.get())$ can be performed. Consequently, after applying a local invocation and a local assignment, we have a MultiASP configuration $\underline{Cn''}$, such that $\underline{Cn'} \to^* \underline{Cn''}$, as follows:

$$\textsc{act}(\alpha, o_\alpha, \sigma'', \{q_A \mapsto \{\ell'[w \mapsto v_r] \mid [\![s]\!]\} :: \{\ell'' \mid \mathsf{s}''\}\} \uplus p', Rq) \quad \textsc{fut}(f, v', \sigma') \qquad \text{MultiASP}$$

  Now, we must compare Cn' and $\underline{Cn''}$ to prove that they are equivalent. The only change concerns the value pointed to by the variable x. Suppose x is a field of the current object (the case where x is a local variable is similar). To prove Line 3 of the definition of equivalence (Figure 4.12), we must ensure that $f \approx^{Cn'}_{\sigma''} v_r$. Indeed, by definition of $\approx^{Cn'}_{\sigma''}$, we need to have $\sigma(o) = v_r$ and also $v \approx^{Cn'}_{\sigma''} v_r$. In our case, this is true because first, $v \approx^{Cn'}_{\sigma'} v'$ and second, by Lemma 4.3.4, we have $v \approx^{Cn'}_{\sigma'} v' \ \wedge \ (v_r, \sigma_r) = \text{rename}_{\sigma'}(v', \sigma') \Rightarrow v \approx^{Cn'}_{\sigma_r} v_r$ with $\sigma_r \subseteq \sigma''$. Regarding activity α, the elements to be considered are the local variables and the statements; the other elements did not change in the reduction. MultiASPconfiguration $\underline{Cn'}$ contains an additional local variable w that comes from the *get* invocation; this local variable is not used. The other local variables did not change, which preserves equivalence. Finally, the remaining statements in ABS and in MultiASP are guaranteed to be equivalent by the fact that $s \approx^{Cn'}_{\sigma''+\ell'} [\![s]\!]$, according to the definition of equivalence $\mathcal{R}$.

- In the second case, where o points to a value, the future has already been updated in the past. Consequently, no preliminary future update is necessary and the last steps of the first case can be performed in the same way, but directly with $\underline{Cn}$ instead of $\underline{Cn'}$ and with v_f instead of v_r. The same arguments in favour of the equivalence $Cn' \mathcal{R} \underline{Cn'}$ can be applied.

4) **Case of the [Await-False] ABS rule.**

We start from the ABS Await-False reduction rule, where $Cn \overset{\text{A}}{\rightarrow} Cn'$:

ABS
$$\frac{f = [\![x]\!]^A_{(a+l)}}{ob(i\_\alpha, a, \{l \mid \texttt{await}\, x\,?; s\}, q)\ \mathit{fut}(f, \bot) \overset{\text{A}}{\rightarrow} ob(i\_\alpha, a, \texttt{idle}, q \cup \{l \mid \texttt{await}\, x\,?; s\})\ \mathit{fut}(f, \bot)}$$

By Lemma 4.3.1, we know that there exists in ABS α such that $cog(\alpha, i\_\alpha) \in Cn$. By the definition of equivalence $\mathcal{R}$, we also know that there exist in MultiASP o_α, σ, p, Rq such that $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$. Furthermore, we have $\textsc{fut}(f, \bot) \in \underline{Cn}$ and, by translational semantics, we have: $\exists \ell', \ell'', s''.\{q_A \mapsto \{\ell' \mid w = [\![x]\!].get(); [\![s]\!]\} :: \{\ell'' \mid s''\}\} \in p$. Thus, we have the following MultiASP configuration $\underline{Cn}$, where $Cn \mathcal{R} \underline{Cn}$:

MultiASP
$$\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell' \mid w = x.get(); [\![s]\!]\} :: \{\ell'' \mid \mathsf{s}''\}\} \uplus p', Rq) \quad \textsc{fut}(f, \bot)$$

By Lemma 4.3.3, we have $[\![x]\!]^A_{(a+l)} \approx^{Cn}_{\sigma} [\![x]\!]_{(\sigma+\ell')}$. By case analysis on the definition of $\approx^{Cn}_{\sigma}$, we have $[\![x]\!]_{(\sigma+\ell')} = o$. Then, only one case is possible in MultiASP: $\sigma(o) = f$, according to the fourth and second case of the definition of $\approx^{Cn}_{\sigma}$. Indeed, no other case is possible because the future has not been resolved yet. Thus, the current statement of activity α is in fact a local method call to $x.get()$, where $\sigma(o) = f$. Since f has not been resolved yet in MultiASP neither, the current request of activity α switches to a passive status (q_P) by the MultiASP rule Invk-Future[2]. Thus, we have the following MultiASP configuration $\underline{Cn'}$, such that $\underline{Cn} \rightarrow \underline{Cn'}$:

MultiASP
$$\textsc{act}(\alpha, o_\alpha, \sigma, \{q_P \mapsto \{\ell' \mid w = x.get(); [\![s]\!]\} :: \{\ell'' \mid \mathsf{s}''\}\} \uplus p', Rq) \quad \textsc{fut}(f, \bot)$$

Now we have to prove $Cn' \mathcal{R} \underline{Cn'}$. In MultiASP, the only element that changed in activity α is the status of the current request, from q_A to q_P. This corresponds

[2] Recall also that the activity has by default a soft thread limit status, only restraining the number of threads with an active status.

to the fact that the current task became `idle` in ABS. Indeed, in the definition of equivalence Figure 4.12, as soon as there is an ABS term $\{l \mid s\}$ in p, there must be a corresponding active thread in MultiASP (Line 4). Similarly, passive threads in MultiASP correspond to some of the requests that are in the queue in ABS. Thus, the passivation of the MultiASP thread corresponds exactly to the transfer of the task to the request queue in ABS. Then, the equivalence of the two tasks is trivial, because $\texttt{await}\, x; s \approx_{\sigma}^{Cn'} w = x.get(); [\![s]\!]$.

5) Case of the [Release-Cog] **ABS** rule.

We have the following ABS reduction $Cn \overset{A}{\to} Cn'$ with the Release-Cog rule:

$$ob(i\_\alpha, a, \texttt{idle}, q)\ cog(\alpha, i\_\alpha) \overset{A}{\to} ob(i\_\alpha, a, \texttt{idle}, q)\ cog(\alpha, \epsilon) \qquad \text{ABS}$$

From configuration Cn, by definition of equivalence $\mathcal{R}$, we know that there exist in MultiASP o_α, σ, p, Rq such that $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq)$ is in the corresponding MultiASP configuration $\underline{Cn}$, where $Cn\,\mathcal{R}\,\underline{Cn}$. Since, in Cn, the current task is `idle` and all other objects in COG α are `idle` too (we know that by Lemma 4.3.1), there exist in MultiASP no $f, i, m, \overline{v''}, \ell', s', \ell'', s''$ such that $(f, execute, i, m, \overline{v''})_A \mapsto \{\ell' \mid s'\} :: \{\ell'' \mid s''\} \in p$. In other words, there is no active *execute* request in the set p of executed requests of activity α. In this case, we directly have $Cn'\,\mathcal{R}\,\underline{Cn}$, because the only term that changed from Cn to Cn' is the *cog* term, and the state of *cog* terms is not relevant in the definition of equivalence $\mathcal{R}$.

6) Case of the [Activate] **ABS** rule.

We have the following ABS reduction $Cn \overset{A}{\to} Cn'$ with the Activate rule:

$$\frac{\alpha = a(\text{cog})}{ob(i\_\alpha, a, \texttt{idle}, q \cup \{l \mid s\})\ cog(\alpha, \epsilon) \overset{A}{\to} ob(i\_\alpha, a, \{l \mid s\}, q)\ cog(\alpha, i\_\alpha)} \qquad \text{ABS}$$

From configuration Cn, by definition of equivalence $\mathcal{R}$, we know that there exist o_α, σ, p, Rq such that $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq)$ is in a corresponding MultiASP configuration $\underline{Cn}$, where $Cn\,\mathcal{R}\,\underline{Cn}$. We also know that there is no active *execute* request in the set p of executed requests of activity α, because in the ABS configuration Cn, the current task is `idle`. As in ABS the $\{l \mid s\}$ request belongs to the pending requests of the ABS object i_α, we have, by definition of equivalence $\mathcal{R}$ (Figure 4.12, Line 5, passive requests case), two possible cases (for the two sides of the disjunction of Line 5), detailed below:

- In the first case, a corresponding passive request in MultiASP is in the set of executed requests p: $\exists f, i, m, \overline{v''}, \ell', s', \ell'', s''. (f, \mathit{execute}, i, m, \overline{v''})_P \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \wedge \ell'(\texttt{this}) = o$, with o the location of the object corresponding to i_α[3]. The requests involved in this case can only be *execute* requests. As we have explained in the translational semantics from ABS to MultiASP (Subsection 4.3.3), *execute* requests belong to group g_2 ($\mathit{group}(\mathit{execute}) = g_2$) and this group has a maximum thread limit of one thread ($\mathcal{L}_{g_2} = 1$). Basically, this means that only one *execute* request can be active at a time. Thus, since no other request is active in p, we have the condition: $\text{Active}(p\big|_{g_2}) < \mathcal{L}_{g_2}$ that is verified. We can apply the Activate-Thread MultiASP rule to be in configuration $\underline{Cn'}$, where $\underline{Cn} \to \underline{Cn'}$:

MultiASP

$$\frac{\mathit{Group}(q') = g_2 \qquad \text{Active}(p\big|_{g_2}) < \mathcal{L}_{g_2}}{\begin{array}{c}\textsc{act}(\alpha, o_\alpha, \sigma, \{q'_P \mapsto \{\ell'|s'\} :: \{\ell''|s''\}\} \uplus p'', Rq)_S \\ \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q'_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\}\} \uplus p'', Rq)_S\end{array}}$$

  Note that in the above reduction, we have $q' = (f, \mathit{execute}, i, m, \overline{v''})$. We now have to prove that $Cn' \mathcal{R} \underline{Cn'}$. The only change in the reduction takes place in the set of executed requests p: there is one less (*execute*) request that is passive (one less request to match in $\mathcal{R}$ in Line 5 of Figure 4.12, but one more (*execute*) request that is active. We must compare the (*execute*) request with the current task in ABS, in Line 4 of $\mathcal{R}$. Proving the condition of Line 4 from the first case of the disjunction of Line 5 is trivial, because all the elements perfectly match.

- In the second case, a corresponding request in MultiASP is in the queue of activity α: $\exists \ell', s', i, m, . (f, \mathit{execute}, i, m, \overline{v''}) \in Rq \wedge o_\alpha.\mathit{retrieve}(i) = o \wedge \text{bind}(o, m, \overline{v''}) = \{\ell'|s'\}$ $\underline{\text{with}}$ $(\forall x \in \mathit{dom}(l) \backslash \mathit{destiny}. l(x) \approx^{Cn}_{\sigma} \ell'(x)) \wedge s \approx^{Cn}_{\sigma+\ell'} s')$. Let $q' = (f, \mathit{execute}, i, m, \overline{v''})$ and $Rq = Rq_0 :: q' :: Rq_1$. In this case, since no other request is active and since *execute* requests are compatible with

[3] Formally, $\sigma(o) = [\mathit{cog} \mapsto \alpha, \mathit{myId} \mapsto i]$.

all other requests (they can run in parallel with any other request), the q' request is ready to be served because the predicate $\text{ready}(q', p, Rq_0)$ is true. Here, we use the fact that we restricted ABS to the FIFO policy for request activation: if a request in Rq_0 could be served, then it would be served first. Thus, we can apply the MultiASP Serve reduction rule to be in configuration $\underline{Cn'}$, where $\underline{Cn} \to \underline{Cn'}$:

$$\frac{\text{ready}(q', p, Rq_0) \quad q' = (f, \mathit{execute}, i, m, \overline{v''}) \quad \text{bind}(o_\alpha, \mathit{execute}, \overline{v''}) = \{\ell'' \mid s''\}}{\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq_0 :: q' :: Rq_1) \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q'_A \mapsto \{\ell'' \mid s''\}\} \uplus p'', Rq_0 :: Rq_1)} \quad \textsf{MultiASP}$$

Note that in the above reduction, s'' is the body of the execute method: $s'' = (w = \mathit{this.retrieve}(i); x = w.\mathtt{m}(\overline{\mathit{params}}); \mathit{return}\ x)$. Furthermore, we also have $o_\alpha.\mathit{retrieve}(i) = o$ thanks the the **Invariant Reg** of Subsection 4.3.4 and thus, after a local method call to *retrieve* and a local assignment, we can perform the main local method call $\mathtt{m}(\overline{\mathit{params}})$ on object o. We can then apply the MultiASP Invk-Passive reduction rule and obtain a last configuration $\underline{Cn''}$, where $\underline{Cn'} \to^* \underline{Cn''}$, as follows:

$$\frac{[\![w]\!]_{(\sigma+\ell'')} = o \qquad [\![\overline{\mathit{params}}]\!]_{(\sigma+\ell'')} = \overline{v''} \qquad \text{bind}(o, m, \overline{v''}) = \{\ell' \mid s'\}}{\begin{array}{c}\textsc{act}(\alpha, o_\alpha, \sigma, \{q'_A \mapsto \{\ell'' \mid x = w.\mathtt{m}(\overline{\mathit{params}}); \mathit{return}\ x\}\} \uplus p'', Rq) \\ \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q'_A \mapsto \{\ell' \mid s'\} :: \{\ell'' \mid x = \bullet; \mathit{return}\ x\}\} \uplus p'', Rq)\end{array}} \quad \textsf{MultiASP}$$

Finally, we have to prove that $Cn' \mathcal{R} \underline{Cn''}$. To this end, there is one less request to match in the second case of the disjunction of Line 5 of $\mathcal{R}$, but the new request has to be compared in Line 4 of $\mathcal{R}$ (Figure 4.12). We have:
$\exists l, s. p' = \{l|s\}$ $\underline{\text{iff}}$ $\exists f, i, m, \overline{v''}, \ell', s', \ell'', s''.$
$\big((f, \mathit{execute}, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \wedge\ \ell'(\mathtt{this}) = o\big)$
$\underline{\text{with}}$ $\forall x \in \mathit{dom}(l) \backslash \mathit{destiny}.\, l(x) \approx^{Cn}_{\sigma} \ell'(x) \wedge l(\mathit{destiny}) = f) \wedge s \approx^{Cn}_{\sigma+\ell'} s'$
Indeed, we constructed a request specially to match those conditions: each condition is either a direct consequence of the applied reduction rules or was ensured by the initial conditions on the request that was in the queue. Besides, by definition of bind, $\text{bind}(o, m, \overline{v''}) = \{\ell' \mid s'\}$ implies that $\ell'(\mathit{this}) = o$, which confirms the last condition above.

7) **Case of the [Read-Fut] ABS rule.**

We have the following ABS reduction $Cn \overset{\text{A}}{\to} Cn'$ with the Read-Fut rule:

ABS

$$\frac{f = [\![e]\!]^{A}_{(a+l)} \qquad v \neq \bot}{ob(i\_\alpha, a, \{l \mid x = e.\mathtt{get}; s\}, q)\ \mathit{fut}(f, v) \overset{\text{A}}{\to} ob(i\_\alpha, a, \{l \mid x = v; s\}, q)\ \mathit{fut}(f, v)}$$

To begin with, from the ABS configuration Cn, we know that, by Lemma 4.3.1, there exists in ABS α such that $cog(\alpha, i\_\alpha) \in Cn$. By the definition of equivalence $\mathcal{R}$, there exist in MultiASP o_α, σ, p', Rq such that $\textsc{act}(\alpha, o_\alpha, \sigma, p', Rq) \in \underline{Cn}$. There also exist ℓ', s', ℓ'', s'' such that $\{q_A \mapsto \{\ell' \mid s'\} :: \{\ell'' \mid s''\}\} \in p'$. By definition of equivalence on statements, we have $(x = e.\mathtt{get}; s) \approx^{Cn}_{\sigma} s'$. By translational semantics, we have $s' = [\![x = e.\mathtt{get}; s]\!]$. Consequently, we have $s' = (\mathbf{setLimitHard}; w = e.get(); \mathbf{setLimitSoft}; x = e); [\![s]\!])$. Thus, we have the following MultiASP configuration $\underline{Cn}$, where $Cn\,\mathcal{R}\,\underline{Cn}$:

MultiASP

$$\begin{array}{l}\textsc{act}(\alpha, o_\alpha, \sigma, \\ \quad \{q_A \mapsto \{\ell | \mathbf{setLimitHard}; w = e.get(); \mathbf{setLimitSoft}; x = e; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq) \\ \textsc{fut}(f, v', \sigma')\end{array}$$

Note that we have $v \approx^{Cn}_{\sigma'} v'$ by definition of equivalence $\mathcal{R}$. And by Lemma 4.3.3, we have $[\![e]\!]^{A}_{a+l} \approx^{Cn}_{\sigma} [\![e]\!]_{\sigma+\ell}$. Now, we can reduce the configuration $\underline{Cn}$ to consume the **setLimitHard** statement with the MultiASP Set-Hard-Limit rule (see Subsection 4.3.2), to be in configuration $\underline{Cn'}$, where $\underline{Cn} \to \underline{Cn'}$, as follows:

MultiASP

$$\begin{array}{l}\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \\ \quad \{\ell \mid \mathbf{setLimitHard}; w = e.get(); \mathbf{setLimitSoft}; x = e; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq)_S \\ \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid w = e.get(); \mathbf{setLimitSoft}; x = e; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq)_H\end{array}$$

Note that in the above reduction, we have: $[\![e]\!]^{A}_{a+l} = f$ in ABS and $[\![e]\!]^{A}_{a+l} \approx^{Cn}_{\sigma} [\![e]\!]_{\sigma+\ell}$ by Definition 3. Now, two cases are possible, detailed below:

- In the first case, the future has not been updated yet, and we have: $[\![e]\!]_{\sigma+\ell} = o$ and $\sigma(o) = f$. Then, a future update can occur through the MultiASP Update reduction rule, to go in configuration $\underline{Cn''}$, where $\underline{Cn'} \to \underline{Cn''}$, as follows:

MultiASP

$$\frac{\sigma(o) = f \qquad (v_r, \sigma_r) = \text{rename}_{\sigma}(v', \sigma') \qquad \sigma'' = \sigma[o \mapsto v_r] \cup \sigma_r}{\textsc{act}(\alpha, o_\alpha, \sigma, p', Rq)\ \textsc{fut}(f, v', \sigma') \to \textsc{act}(\alpha, o_\alpha, \sigma'', p', Rq)\ \textsc{fut}(f, v', \sigma')}$$

Note that in the above reduction, we have: $p' = \{\ell|w = e.get(); \mathbf{setLimitSoft}; x = e; [\![s]\!]\} :: \{\ell'' \mid s''\} \uplus p$. At this point, the $get()$ method call on object o can be performed and this method call succeeds because the future has been updated. Additionally, thanks to Lemma 4.3.4, $v \approx^{Cn}_{\sigma'} v'$ implies that $v \approx^{Cn}_{\sigma''} v_r$. In other words, the value of the future after serialisation is preserved. Here, we use the fact that we excluded the particular executions where the value of a future is a future (see Subsection 4.3.4), so since v is not a future, then neither v' nor v_r can point to a future. Finally, after applying some local reduction rules (a local method invocation, a local assignment (changing ℓ to ℓ') and a change in the kind of thread limit), we end up in the following MultiASP configuration $\underline{Cn_3}$, where $\underline{Cn''} \to^* \underline{Cn_3}$:

$$\text{ACT}(\alpha, o_\alpha, \sigma'', \{q_A \mapsto \{\ell'|x = e; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq)_S \quad \text{FUT}(f, v', \sigma') \qquad \text{MultiASP}$$

Finally, we have to prove that $Cn' \mathcal{R} \underline{Cn_3}$. We have indeed $v \approx^{Cn'}_{\sigma''} v'$ and we also have: $(x = v; s) \approx^{Cn'}_{\sigma''} (x = e; [\![s]\!])$, because $[\![e]\!]_{\sigma''+\ell'} = v_r$ and $v \approx^{Cn'}_{\sigma''} v_r$. Lastly, the other elements need not being considered because they did not change.

- In the second case, the future has already been updated, and we have: $[\![e]\!]_{\sigma+\ell} = v'$. By Lemma 4.3.3, we also have: $f \approx^{Cn}_{\sigma} v'$. Then, performing get on e has no visible effect, and after applying several local MultiASP reduction rules (like in the first case), we end up in a configuration $\underline{Cn''}$, where $\underline{Cn'} \to \underline{Cn''}$, as follows:

$$\text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell'|x = e; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq) \quad \text{FUT}(f, v', \sigma') \qquad \text{MultiASP}$$

This final configuration is similar to the one of the first case. The only difference is that no future update was needed and thus the local store is unchanged. For the other elements, we can prove $Cn' \mathcal{R} \underline{Cn''}$ similarly to the case above.

8) **Case of the [New-Object] ABS rule.**

We start from the NEW-OBJECT rule in which $Cn \xrightarrow{\text{A}} Cn'$ as follows:

ABS

$$\frac{i'\_\alpha = \text{fresh}(\mathtt{C}) \quad \mathit{fields}(\mathtt{C}) = \overline{x} \qquad \overline{v} = [\![\overline{e}]\!]^A_{(a+l)} \quad a' = [\overline{x \mapsto v}, \text{cog} \mapsto \alpha]}{ob(i\_\alpha, a, \{l \mid x = \mathtt{new\ local\ C}(\overline{e}); s\}, q)\ cog(\alpha, i\_\alpha) \xrightarrow{\text{A}} ob(i\_\alpha, a, \{l \mid x = i'\_\alpha; s\}, q)\ cog(\alpha, i\_\alpha) ob(i'\_\alpha, a', \mathtt{idle}, \varnothing)}$$

By definition of equivalence $\mathcal{R}$, there is a MultiASP activity in $\underline{Cn}$ corresponding to $cog(\alpha, i\_\alpha)$: $\exists o_\alpha, \sigma, p', Rq.\textsc{act}(\alpha, o_\alpha, \sigma, p', Rq)$. By translational semantics, p' contains the statements that create an equivalent new object in MultiASP: $p' = \{q_A \mapsto \{\ell \mid t = \mathit{this.cog}(); \mathit{id} = t.\mathit{freshId}(); no = \mathtt{new\ C}(\overline{e}, t, \mathit{id}); z = t.\mathit{register}(no, \mathit{id}); x = no; [\![s]\!]\} :: E \uplus p''\}$. By definition of equivalence $\mathcal{R}$, there is a MultiASP object in $\underline{Cn}$ corresponding to $ob(i\_\alpha, a, \cdots)$: $\exists o, \overline{v'}.\sigma(o) = [\mathit{cog} \mapsto \alpha, myId \mapsto i, \overrightarrow{x \mapsto v'}]$, and this object maps to the current local variable this: $\ell(\mathit{this}) = o$. In summary, we have the following MultiASP configuration $\underline{Cn}$, where $Cn\,\mathcal{R}\,\underline{Cn}$:

MultiASP

$$\textsc{act}(\alpha, o_\alpha, \sigma[o \mapsto [\mathit{cog} \mapsto \alpha, myId \mapsto i, \overrightarrow{x \mapsto v'}]], \{q_A \mapsto \{\ell[\mathit{this} \mapsto o] \mid t = \mathit{this.cog}(); \mathit{id} = t.\mathit{freshId}(); no = \mathtt{new\ C}(\overline{e}, t, \mathit{id}); z = t.\mathit{register}(no, \mathit{id}); x = no; [\![s]\!]\} :: E \uplus p''\}, Rq)$$

Then, $\underline{Cn}$ can be reduced by evaluating all local reductions of α until a configuration $\underline{Cn'}$ has the object instantiation as current statement. These reduction steps only execute local assignments and local (synchronous) method calls fetching the COG and the identifier of the invoked object. In particular, we focus on $\underline{Cn} \to^* \underline{Cn'}$ where $\underline{Cn'}$ contains the activity α with the current thread executing the object instantiation: $q_A \mapsto \{\ell' \mid no = \mathtt{new\ C}(\overline{e}, t, \mathit{id}); z = t.\mathit{register}(no, \mathit{id}); x = no; [\![s]\!]\}$, with $\ell' = \ell[t \mapsto \alpha, \mathit{id} \mapsto i']$[4]. This configuration $\underline{Cn'}$ can be reduced to a configuration $\underline{Cn''}$, such that $\underline{Cn'} \to \underline{Cn''}$ by the MultiASP NEW-OBJECT rule, as follows:

MultiASP

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{x} \quad o' \ \text{fresh} \quad \sigma' = \sigma \cup \{o' \mapsto [\mathit{cog} \mapsto \alpha, myId \mapsto i', \overrightarrow{x = v''}]\} \quad [\![\overline{e}]\!]_{(\sigma+\ell')} = \overline{v''}}{\underline{Cn'} \to \textsc{act}(\alpha, o_\alpha, \sigma', \{q_A \mapsto \{\ell' \mid no = o'; z = t.\mathit{register}(no, \mathit{id}); x = no; [\![s]\!]\} :: E\} \uplus p'', Rq)}$$

The configuration $\underline{Cn''}$ has now the activity α containing the new object. By now, this object has a proper identifier and points to the right activity α (it points to its COG). The *register* method call is then evaluated. Its effect is handled by

[4] Note that i', the same identifier as the fresh identifier allocated by ABS, can be chosen as a fresh identifier by the method *freshId*. This is due to the definition of equivalence between objects: if i' was not fresh in α, i'_α would already be an existing ABS object and could not be chosen as a fresh ABS object identifier.

the **Invariant Reg**, which makes sure that the object is available to the system after this call. We prove the **Invariant Reg** in Subsection 4.3.4, and it is verified here. This leads us to the final configuration $\underline{Cn'''}$ such that $\underline{Cn''} \to^* \underline{Cn'''}$, that contains the following activity:

$$\text{ACT}(\alpha, o_\alpha, \sigma', \{q_A \mapsto \{\ell'' \mid x = no; [\![s]\!]\} :: E\} \uplus p'', Rq) \qquad \text{MultiASP}$$

Note that in the above reduction, we have: $\ell'' = \ell'[no \mapsto o']$. Now we have to prove $Cn' \mathcal{R} \underline{Cn'''}$. Typically, we have two objects of the same COG affected in ABS (i_α and i'_α), and one activity affected in MultiASP by the reduction. We focus on the state of the two modified elements below:

- Let us first consider i'_α. We have a fresh object i'_α in ABS that must be equivalent to the fresh object o' in the store of activity α in MultiASP. By Definition 4 of $\mathcal{R}$, in Line 2 of Figure 4.12, we have the additional fields of the new object that point to the COG (activity α) and to the identifier i', that corresponds to the ABS identifier i'_α. The equivalence of the other fields of the fresh objects (in ABS and in MultiASP) is obtained by Lemma 4.3.3. The newly created object is `idle` in ABS with no pending request. Consequently, Line 4 and Line 5 of Figure 4.12 are trivially verified.

- Secondly, let us consider i_α. The currently served request is the only element that changed, so only Line 4 of Figure 4.12 has to be checked. The set of local variables did not change, except in MultiASP where the set of local variables contains more variables. Finally, we have to check the equivalence of the remaining statements. We fall in the second case of the definition of the equivalence of statements: $s_{\text{ABS}} = (x = i'\_\alpha; s) \ \wedge \ s_{\text{MultiASP}} = (x = no; [\![s]\!])$ *with* $i'\_\alpha \approx_\sigma^{Cn'} [\![no]\!]_{(\sigma'+\ell'')}$. Indeed, the currently considered statements in ABS and in MultiASP fit with the requirements of the definition. Additionally, we have: $[\![no]\!]_{(\sigma'+\ell'')} = o'$ and we have already shown before the equivalence between i'_α and o'.

9) **Case of the [New-Cog-Object] ABS rule.**

We start from the NEW-COG-OBJECT rule where $Cn \overset{\text{A}}{\to} Cn'$, as follows:

ABS

$$\frac{\beta = \text{fresh}(\,) \quad i'\_\beta = \text{fresh}(\texttt{C}) \quad \mathit{fields}(\texttt{C}) = \overline{x} \qquad \overline{v} = [[\overline{e}]]^{A}_{(a+l)} \quad a' = [\overline{x \mapsto v}, \text{cog} \mapsto \beta]}{\begin{array}{c} \mathit{ob}(i\_\alpha, a, \{l \mid x = \texttt{new C}(\overline{e}); s\}, q) \\ \overset{\text{A}}{\to} \mathit{ob}(i\_\alpha, a, \{l \mid x = i'\_\beta; s\}, q) \quad \mathit{ob}(i'\_\beta, a', \texttt{idle}, \varnothing) \quad \mathit{cog}(\beta, \epsilon) \end{array}}$$

From the ABS configuration Cn, we know that by Lemma 4.3.1, $\exists \alpha.\, \mathit{cog}(\alpha, i\_\alpha) \in Cn$. And, by definition of equivalence $\mathcal{R}$, $\exists o_\alpha, \sigma, p', Rq.\textsc{act}(\alpha, o_\alpha, \sigma, p', Rq) \in \underline{Cn}$ and $\exists f, i, m, \overline{v''}, \ell', s', \ell'', s''.(f, \mathit{execute}, i, m, \overline{v''})_A \mapsto \{\ell' \mid s'\} :: \{\ell'' \mid s''\} \in p'$ and $(x = \texttt{new C}(\overline{e}); s) \approx^{Cn}_{\sigma+\ell'} s'$. By definition of the equivalence on statements, and by translational semantics, we have: $(x = \texttt{new C}(\overline{e}); s) \approx^{Cn}_{\sigma+\ell'} s'$ implies $s' = [\![x = \texttt{new C}(\overline{e}); s]\!]$. Consequently, the translational semantics gives us:

$$\begin{aligned} s' = (&\mathit{newcog} = \mathit{newActive}\ \textsc{cog}(); \mathit{id} = \mathit{newcog.freshId}(); \mathit{no} = \texttt{new C}'(\overline{e}, \mathit{newcog}, \mathit{id}); \\ &z = \mathit{newcog.register}(\mathit{no}, \mathit{id}); x = \mathit{no}; [\![s]\!]) \end{aligned}$$

All of these elements allow us to define the MultiASP configuration $\underline{Cn}$, where $Cn\,\mathcal{R}\,\underline{Cn}$:

MultiASP

$$\begin{array}{c} \textsc{act}(\alpha, o_\alpha, \sigma, \{\mathtt{q}_A \mapsto \{\ell' \mid \mathit{newcog} = \mathit{newActive}\ \textsc{cog}(); \mathit{id} = \mathit{newcog.freshId}(); \mathit{no} = \\ \texttt{new C}'(\overline{e}, \mathit{newcog}, \mathit{id}); z = \mathit{newcog.register}(\mathit{no}, \mathit{id}); x = \mathit{no}; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq) \end{array}$$

The MultiASP configuration $\underline{Cn}$ can be reduced with the NEW-ACTIVE rule to the configuration $\underline{Cn'}$, such that $\underline{Cn} \to \underline{Cn'}$:

MultiASP

$$\frac{\begin{array}{c} \mathit{fields}(\textsc{cog}) = \overline{x_{\text{COG}}} \\ \beta,\, o_\beta \ \ \text{fresh} \qquad \sigma' = \{o_\beta \mapsto [\overrightarrow{\overline{x_{\text{COG}} = v_{\text{COG}}}}]\} \cup \mathit{serialise}(\overline{v_{\text{COG}}}, \sigma) \qquad [[\overline{e_{\text{COG}}}]]_{(\sigma+\ell')} = \overline{v_{\text{COG}}} \end{array}}{\begin{array}{c} \underline{Cn} \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell' \mid \mathit{newcog} = \gamma; \mathit{id} = \mathit{newcog.freshId}(); \mathit{no} = \\ \texttt{new C}'(\overline{e}, \mathit{newcog}, \mathit{id}); z = \mathit{newcog.register}(\mathit{no}, \mathit{id}); x = \mathit{no}; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq) \\ \textsc{act}(\beta, o_\beta, \sigma', \varnothing, \varnothing) \end{array}}$$

Note that we can pick β as a fresh activity name because, by definition of $\mathcal{R}$, as β is free in ABS , it is also free in MultiASP. Now, we can reduce $\underline{Cn'}$ to a configuration $\underline{Cn''}$ by two local assignments and a remote invocation (to get a fresh identifier), until the current statement is the new object instantiation:

$$\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell''' \mid no = \texttt{new C}'(\overline{e}, newcog, id); z = newcog.register(no, id); x = no; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq) \quad \textsc{act}(\beta, o_\beta, \sigma', \varnothing, \varnothing) \quad \textsc{fut}(f, \bot) \qquad \textsf{MultiASP}$$

Note that in the above reduction, we have: $\ell''' = \ell'[newcog \mapsto \beta, id \mapsto f]$. Then, it is possible to reduce this configuration until a fresh identifier is found and returned, and where the future f has been updated in activity α. In particular, we focus on the MultiASP configuration $\underline{Cn_3}$, such that $\underline{Cn''} \to^* \underline{Cn_3}$ with the previous requirements:

$$\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell''' \mid no = \texttt{new C}'(\overline{e}, newcog, id); z = newcog.register(no, id); x = no; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq) \quad \textsc{act}(\beta, o_\beta, \sigma', \varnothing, \varnothing) \quad \textsc{fut}(f, i', \varnothing) \qquad \textsf{MultiASP}$$

Note that in the above reduction, we have: $\ell'''(id) = i'$ and we know that i' is necessarily a fresh identifier in β. On $\underline{Cn_3}$, we can apply the New-Object MultiASP rule to obtain the configuration $\underline{Cn_4}$ such that $\underline{Cn_3} \to \underline{Cn_4}$, as follows:

$$\frac{\begin{array}{c} \mathit{fields}(\texttt{C}) = \overline{x} \\ o \text{ fresh} \qquad \sigma'' = \sigma \cup \{o \mapsto [cog \mapsto \beta, myId \mapsto f, \overrightarrow{x = v''}]\} \qquad [\![\overline{e}]\!]_{(\sigma+\ell')} = \overline{v''} \end{array}}{\begin{array}{c} \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell''' \mid no = \texttt{new C}'(\overline{e}, newcog, id); \\ z = newcog.register(no, id); x = no; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq) \\ \textsc{act}(\beta, o_\beta, \sigma', \varnothing, \varnothing) \quad \textsc{fut}(f, i', \varnothing) \to \textsc{act}(\alpha, o_\alpha, \sigma'', \{q_A \mapsto \{\ell''' \mid no = o; \\ z = newcog.register(no, id); x = no; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq) \\ \textsc{act}(\beta, o_\beta, \sigma', \varnothing, \varnothing) \quad \textsc{fut}(f, i', \varnothing) \end{array}} \qquad \textsf{MultiASP}$$

Then, the final steps consist in reducing the local assignment and evaluating the *register* remote method invocation. This brings us to the final configuration $\underline{Cn_5}$, where $\underline{Cn_4} \to^* \underline{Cn_5}$, which contains the following MultiASP terms:

$$\textsc{act}(\alpha, o_\alpha, \sigma'', \{q_A \mapsto \{\ell_4 \mid x = no; [\![s]\!]\} :: E\} \uplus p'', Rq) \quad \textsc{act}(\beta, o_\beta, \sigma_3, \varnothing, (f', register, o', i')) \quad \textsc{fut}(f, i', \varnothing) \quad \textsc{fut}(f', \bot) \qquad \textsf{MultiASP}$$

Note that in the above reduction, we have: $\ell_4 = \ell'''[no \mapsto o]$. In particular, $\sigma_3(o') = [cog \mapsto \beta, myId \mapsto i', \overline{x \mapsto v_2}]$ is the serialisation of o. Now we have to prove that $Cn' \mathcal{R} \underline{Cn_5}$. Two activities in MultiASP have to be considered: activity α, that has been modified during the reductions, and activity β, that has been introduced by the reductions. The two activities are detailed and checked against the equivalence $\mathcal{R}$ below:

- The first MultiASP activity (α) corresponds to *cog* α in ABS (that exists by Lemma 4.3.1), contains now in its store a copy of the new object o corresponding to object i'_β in ABS. Objects o and i'_β are equivalent for two reasons. First, o contains the required additional fields: *cog* that points the new activity β, and $myId$ with value i'. Second, the other fields are only meaningful in activity β, thus we do not have to consider them in activity α. Regarding the current request, first the set of local variables in MultiASP contains two more variables, but the existing local variables did not change in the reductions. Second, the remaining statements fall in the second case of the definition of the equivalence of statements. Like in the case of [New-Object], we have: $s_{\text{ABS}} = (x = i'\_\beta; s) \wedge s_{\text{MultiASP}} = (x = no; [\![s]\!])$ *with* $i'\_\beta \approx^{Cn'}_{\sigma''} [\![no]\!]_{(\sigma''+\ell_4)}$. Since we have: $[\![no]\!]_{(\sigma''+\ell_4)} = o$, and since we showed before that o is equivalent to i'_β, we can conclude that the remaining statements are equivalent.

- The second MultiASP activity (β) corresponds to the fresh *cog* β in ABS. The content of this activity reflects the fresh object o'. Concerning the object value, we must prove equivalence of object fields other than *cog* and $myId$. Here, we recall the object's fields that are meaningful are the ones that are hosted in β. Precisely, we have: $\overline{v} = [\![\overline{e}]\!]^{A}_{(a+l)}$ and we also have: $(\overline{v_2}, \sigma_3) = \text{rename}_{\sigma'}([\![\overline{e}]\!]_{(\sigma''+\ell_4)}, \text{serialise}([\![\overline{e}]\!]_{(\sigma''+\ell_4)}, \sigma''))$. By Lemma 4.3.3 and by Lemma 4.3.4, we have: $\overline{v} \approx^{Cn'}_{\sigma_3} \overline{v_2}$. This verifies Line 3 of the definition of equivalence $\mathcal{R}$. In ABS, the new object is `idle` with no pending request. Line 4 and Line 5 of the definition of equivalence $\mathcal{R}$ is verified, because no request is currently served in ABS, and accordingly, in MultiASP there is no *execute* request is currently served.

Finally, we have two additional futures in MultiASP concerning the *freshId* and the *register* remote method calls. However, they are not considered in the equivalence because they do not correspond to an *execute* request (i.e. to an applicative request).

10) **Case of the [Rendez-vous-Comm] ABS rule.**

In this case, we only deal with communications that are not performed on the caller itself. Indeed, the self asynchronous method call is similar but requires a specific proof instance. So we start from the following ABS Rendez-vous-Comm reduction rule, where $Cn \xrightarrow{A} Cn'$, and where $\alpha \neq \beta$, as follows:

$$\frac{f = \text{fresh}(\ ) \qquad i'\_\beta = [\![e]\!]^A_{(a+l)} \qquad \overline{v} = [\![\overline{e}]\!]^A_{(a+l)} \qquad p'' = \text{bind}(i'\_\beta, f, m, \overline{v}, \texttt{class}(i'\_\beta))}{\begin{array}{c} ob(i\_\alpha, a, \{l \mid x = e!\texttt{m}(\overline{e}); s\}, q)\ ob(i'\_\beta, a', p', q') \\ \xrightarrow{A} ob(i\_\alpha, a, \{l \mid x = f; s\}, q)\ ob(i'\_\beta, a', p', q' \cup p'')\ fut(f, \bot)\end{array}} \quad \text{ABS}$$

First of all, by Lemma 4.3.1, $\exists \alpha.\ cog(\alpha, i\_\alpha) \in Cn$ and $\exists \beta.\ cog(\beta, i'\_\beta) \in Cn$. By definition of equivalence $\mathcal{R}$ and of translational semantics, there exist $o_\alpha, \sigma_\alpha, p$, $Rq, \ell, \ell'', s'', o_\beta, \sigma_\beta, p', Rq$ such that the following terms belong to the MultiASP configuration $\underline{\text{Cn}}$, where $Cn\,\mathcal{R}\,\underline{Cn}$, as follows:

$$\begin{array}{l} \text{ACT}(\alpha, o_\alpha, \sigma_\alpha, \{q_A \mapsto \\ \qquad \{\ell | t = e.cog(); id = e.myId(); x = t.execute(id, \texttt{m}, \overline{e}); [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq) \\ \text{ACT}(\beta, o_\beta, \sigma_\beta, p', Rq') \end{array} \quad \text{MultiASP}$$

In the MultiASP configuration, there must be an object equivalent to i'_β resulting from the evaluation of e in activity α. This object has two copies: one in activity α and one in activity β, but the value of its applicative fields in activity α are meaningless (they are never used). Indeed, the copy of this object in activity α plays the role of a proxy to its meaningful counterpart in β. More formally, we have:

$$\begin{array}{l} o' \mapsto [cog \mapsto \beta, myId \mapsto i', [\overrightarrow{x \mapsto v'}]] \in \sigma_\alpha \ \text{ with } \ [\![e]\!]_{\sigma+\ell} = o' \\ o'' \mapsto [cog \mapsto \beta, myId \mapsto i', [\overrightarrow{x \mapsto v''}]] \in \sigma_\beta \ \text{ with } \ a' \approx_{\sigma_\beta} [\overrightarrow{x \mapsto v''}] \end{array}$$

The MultiASP configuration $\underline{Cn}$ can be reduced using local rules until a configuration $\underline{Cn'}$ contains the *execute* remote method invocation as current statement. More formally, we have $\underline{Cn} \rightarrow^* \underline{Cn'}$, where the current thread of activity α in $\underline{Cn'}$ is: $q_A \mapsto \{\ell' | x = t.execute(id, \texttt{m}, \overline{e}); [\![s]\!]\} :: \{\ell'' \mid s''\}$, with $\ell' = \ell[t \mapsto \beta, id \mapsto i']$. Then, we can reduce $\underline{Cn'}$ to a configuration $\underline{Cn''}$ by the MultiASP Invk-Active rule, such that $\underline{Cn'} \rightarrow \underline{Cn''}$, as follows:

$$\textsf{MultiASP}\quad \frac{[\![t]\!]_{(\sigma_\alpha+\ell')}=\beta \quad [\![\overline{e}]\!]_{(\sigma_\alpha+\ell')}=\overline{v} \qquad f, o_f \text{ fresh} \qquad (\overline{v_r}, \sigma_r)=\text{rename}_{\sigma_\beta}(\overline{v}, \text{serialise}(\overline{v}, \sigma_\alpha))}{\begin{array}{c} \textsc{act}(\alpha, o_\alpha, \sigma_\alpha, \{q_A \mapsto \{\ell' \mid x = t.\mathit{execute}(id, \mathtt{m}, \overline{e}); [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq) \\ \textsc{act}(\beta, o_\beta, \sigma_\beta, p', Rq') \\ \rightarrow \textsc{act}(\alpha, o_\alpha, \sigma_\alpha[o_f \mapsto f], \{q_A \mapsto \{\ell' \mid x = o_f; [\![s]\!]_\}} :: \{\ell'' \mid s''\}\} \uplus p, Rq) \\ \textsc{act}(\beta, o_\beta, \sigma_\beta \cup \sigma_r, p', Rq' :: (f, \mathit{execute}, i', \mathtt{m}, \overline{v_r})) \qquad \textsc{fut}(f, \bot) \end{array}}$$

We finally have to prove that $Cn' \mathcal{R} \underline{Cn''}$. First, we have as many terms that have changed in Cn' as in $\underline{Cn''}$: there are two *ob* and ACT terms and one added future in both configurations. Regarding the fresh future, it is the same in the two configurations: we conclude by Line 6 of the definition of equivalence $\mathcal{R}$.

Regarding activity α, the set of local variables remains unchanged, except for temporary variables. The element that actually changed is the current statement. We have: $(x = f; s) \approx^{Cn'}_{\sigma_\alpha} (x = o_f; [\![s]\!])$ by the second case of the definition of the equivalence of statements. For that, we need to have: $f \approx^{Cn'}_{\sigma_\alpha} [\![o_f]\!]_{\sigma_\alpha+\ell'}$, which is true because $[\![o_f]\!]_{\sigma_\alpha+\ell'} = o_f$ and because we have: $\sigma_\alpha(o_f) = f$. Thus, we have: $f \approx^{Cn'}_{\sigma_\alpha} \sigma(o_f)$ by definition of equivalence $\mathcal{R}$. We conclude the equivalence between activity and *cog* α using Line 4 of the definition of equivalence $\mathcal{R}$.

Regarding activity β, a new pending request is created in the MultiASP configuration $\underline{Cn''}$, corresponding to the new inactive thread in the ABS configuration Cn'. According to Line 5 of the definition of equivalence $\mathcal{R}$, we have: $\{l|s\} \in q' \wedge l(\mathit{destiny}) = f$, with $\{l|s\} = \text{bind}(i'\_\beta, f, m, \overline{v}, \mathtt{class}(i'\_\beta))$. Then, there is only one case to consider because we know that the request is not served yet. Thus, what we exactly have to prove is the following:

$$\begin{array}{l} \exists i', \mathtt{m}, \overline{v_r}, \ell_\beta, s'.(f, \mathit{execute}, i', \mathtt{m}, \overline{v_r}) \in Rq \wedge o_\beta.\mathit{retrieve}(i')=o \wedge \text{bind}(o, \mathtt{m}, \overline{v_r}) = \{\ell_\beta|s'\} \\ \underline{\text{with}} \quad \forall x \in \mathit{dom}(l)\backslash \mathit{destiny}.\, l(x) \approx^{Cn'}_{\sigma_\beta} \ell_\beta(x) \wedge s \approx^{Cn'}_{\sigma_\beta+\ell_\beta} s' \end{array}$$

Firstly, we have indeed the new request in the queue of activity β in the MultiASP configuration $\underline{Cn''}$. Then, we need to ensure that: $o_\beta.\mathit{retrieve}(i')=o$, which is guaranteed by the **Invariant Reg** (see Subsection 4.3.4). On the other hand, the result of the auxiliary function bind in MultiASP is similar to the one that exists in ABS. The local variables on the ABS side also appear on the MultiASP side equivalently, except for the two following points:

- There is no *destiny* variable in MultiASP. However, this is taken into account by the definition of equivalence $\mathcal{R}$, which compares the *destiny* variable with the future of the corresponding MultiASP request.

- The transmitted parameters $\overline{v_r}$ are the copies of the method parameters in ABS. Ensuring equivalence between request parameters in this case is handled by Lemma 4.3.3 for obtaining the equivalence of the emitted values, and by Lemma 4.3.4 to ensure that equivalence still holds after the values being serialised, sent to β and renamed.

11) **Case of the [Return] ABS rule.**

We start from the following Return ABS reduction rule, where $Cn \overset{\text{A}}{\to} Cn'$, as follows:

$$\frac{v = [\![e]\!]^{A}_{(a+l)} \quad f = l(\text{destiny})}{ob(i\_\alpha, a, \{l \mid \texttt{return}\ e; s\}, q)\ fut(f, \perp) \overset{\text{A}}{\to} ob(i\_\alpha, a, \texttt{idle}, q)\ fut(f, v)} \quad \text{ABS}$$

From the starting ABS configuration, we have, by Lemma 4.3.1, $\exists cog(\alpha, i\_\alpha) \in Cn$. By definition of equivalence $\mathcal{R}$, $\exists o_\alpha, \sigma, p, Rq$ such that $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$, and $\exists f, i, m, \overline{v''}, \ell', s', \ell'', s''$ such that $(f, \mathit{execute}, i, m, \overline{v''})_A \mapsto \{\ell' \mid s'\} :: \{l'' \mid s''\} \in p$, and $\texttt{return}\ e; s \approx^{Cn}_{\sigma+\ell'} s'$. By the definition of the equivalence of statements, we have: $s' = (\texttt{return}\ e; [\![s]\!])$, since the first statement is not in the form of $x = e; s$. Also, by Line 7 of the equivalence $\mathcal{R}$, we have: $\textsc{fut}(f, \perp) \in \underline{Cn}$. Furthermore, we know that in our MultiASP translation, a method call on an object is always wrapped in an *execute* method call on an active object (by definition of $\mathcal{R}$). Therefore, we have $s'' = (x = \bullet; \texttt{return}\ x)$. All of these elements allow us to find the following MultiASP configuration $\underline{Cn}$, where $Cn\,\mathcal{R}\,\underline{Cn}$:

$$\begin{array}{l}\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell' \mid \texttt{return}\ e; [\![s]\!]\} :: \{\ell'' \mid x = \bullet; \texttt{return}\ x\}\} \uplus\ p', Rq) \\ \textsc{fut}(f, \perp)\end{array} \quad \text{MultiASP}$$

From $\underline{Cn}$, we can apply the Return-Local MultiASP reduction rule to be in the configuration $\underline{Cn'}$, such that $\underline{Cn} \rightarrow \underline{Cn'}$, as follows:

MultiASP

$$\frac{v' = \llbracket e \rrbracket_{\sigma+\ell'}}{\underline{Cn} \rightarrow \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell'' \mid x = v'; \texttt{return } x\}\} \uplus p', Rq) \quad \textsc{fut}(f, \bot)}$$

Then, after applying the Assign-Local MultiASP reduction rule on $\underline{Cn'}$, we are in the configuration $\underline{Cn''}$, such that $\underline{Cn'} \rightarrow \underline{Cn''}$:

MultiASP

$$\begin{array}{l}\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell''[x \mapsto v'] \mid \texttt{return } x\}\} \uplus \ p', Rq) \\ \textsc{fut}(f, \bot)\end{array}$$

We now denote $\ell_3 = \ell''[x \mapsto v']$. From $\underline{Cn''}$, we can apply the Return MultiASP reduction rule to finally end up in the configuration $\underline{Cn'''}$, such that $\underline{Cn''} \rightarrow \underline{Cn'''}$, as follows:

MultiASP

$$\frac{\llbracket x \rrbracket_{\sigma+\ell_3} = v'}{\underline{Cn''} \rightarrow \textsc{act}(\alpha, o_\alpha, \sigma, p', Rq) \quad \textsc{fut}(f, v', \mathit{serialise}(v', \sigma))}$$

We further denote: $\sigma_s = \mathit{serialise}(v', \sigma)$, and we recall that: $\llbracket e \rrbracket_{\sigma+\ell'} = v'$. Now, we have to prove that $Cn' \mathcal{R} \underline{Cn'''}$. Regarding the resolved future, we have the future term f both in the ABS and MultiASP configurations. Secondly, we have to ensure that: $v \approx_{\sigma_s}^{Cn'} v'$ and to this end, we have to ensure that: $v \approx_{\sigma}^{Cn'} \llbracket e \rrbracket_{\sigma+\ell'}$. Indeed, by Lemma 4.3.3, we have: $\llbracket e \rrbracket_{a+l}^{A} \approx_{\sigma}^{Cn'} \llbracket e \rrbracket_{\sigma+\ell'}$. Thus, we have: $v \approx_{\sigma}^{Cn'} v'$ and, by Lemma 4.3.4, we have: $v \approx_{\sigma_s}^{Cn'} v'$. Regarding activity α, the request corresponding to future f does not exist any more, which matches with the fact that the ABS current task becomes `idle`: in the definition of equivalence $\mathcal{R}$, there is one less request to compare in Line 4. Finally, the other elements of the activity did not change, thus preserving their equivalence.

12) **Case of the [Skip], [Cond-True], [Cond-False], [Context], [Cog-Sync-Call], [Self-Sync-Call], [Cog-Sync-Return-Sched], [Self-Sync-Return-Sched] and [Rem-Sync-Call] ABS rules.**

To finish, none of these rules are considered in the proof of Theorem 4.3.5, for different reasons. We review the reasons for not treating each of them below.

Firstly, in practice Skip, Cond-True, Cond-False and Context are kept unchanged from the translation provided by the Java backend for ABS to implement the ProActive backend for ABS. Consequently, their translation in an imperative programming language without remote method invocations and futures is already supposed to be sound and correct, and is trivial because it does not involve asynchronous remote method calls and futures. Furthermore, in the case of the ProActive backend, the `await` statement on conjunctive guards and boolean expressions is not handled using these rules. However, none of these four rules raises a particular difficulty because they only involve local computation.

Secondly, all the rules dealing with local synchronous method calls, namely Cog-Sync-Call, Self-Sync-Call, Cog-Sync-Return-Sched and Self-Sync-Return-Sched are not considered in the proof. Indeed, the effect of these rules send back the current task in queue and schedule it right away with the continuation statement `cont` to mimic a synchronous treatment. Instead, we rely on a classical stack of method calls and avoid considering the `cont` keyword. Besides, these rules make use of statements that are treated in other cases, like `await` and `get`. Consequently, considering these rules would be redundant and would not bring any particular insight on the correctness of the translation. Additionally, the practical translation of those aspects is again unchanged from the Java backend for ABS.

Thirdly, in the case of the rule for remote synchronous method calls, namely Rem-Sync-Call, the ABS rule is a composition of a remote asynchronous method call and a future read; in the translation, we directly inline this composition. Thus, the proof for this rule simply inlines the two cases of Rendez-vous-Comm and Read-Fut.

In conclusion, all reduction rules of the ABS semantics ensure that an equivalent final configuration is reachable with the MultiASP translation that we propose. Consequently, the translation simulates all ABS executions with FIFO policy and rendez-vous communications, provided that no future value is a future reference, which completes the proof of Theorem 4.3.5. □

Overall, although the equivalence of the ABS and MultiASP configurations is achieved for all ABS reduction rules, it has some restrictions where it is impossible to have a strong simulation, because of the intrinsic differences between the two languages. Indeed, to this end we must have considered some of the ABS and some of the MultiASP rules as *silent actions*: On one hand, we have some ABS rules that are strictly associated to one MultiASP rule, like for strong simulation. But on the other hand, for some other ABS rules there is an associated MultiASP rule, but also they must be associated with additional rules that are needed to reach the correct simulation: we call these latter rules the silent actions. Also, some ABS rules cannot be simulated by MultiASP rules. More precisely, they do not strictly correspond to any MultiASP rule, so in this case either the equivalence is just maintained, or some not observable transitions must be achieved (silent actions). Table A.1 summarises our results on the simulation and on observability of ABS and MultiASP rules. The '–' character denotes the case where there is no strict correspondence of rules. The '/' character indicates a choice between two rules depending on a criteria detailed in the proof but not in the table. Further comments can be made about Table A.1.

Firstly, we can notice that an additional Invk-Passive and Assign-Local-Tmp MultiASP rules are quite omnipresent. Indeed, we exhibited before that our translation introduces additional communications and temporary variables. We can notice that, if we ignore the introduction of those 'harmless' rules (in terms of what we are interested in observing), most of the important ABS rules can be simulated by a single MultiASP rule, and in this case we achieve a strong simulation.

Secondly, we can notice that we can simulate the Await-False ABS rule with the Invk-Future MultiASP rule, but the Await-True ABS rule has no MultiASP equivalent. Then, for example it might have been more consistent in Table A.1 to consider the Invk-Future MultiASP rule as a silent action for the Await-False ABS rule too.

Thirdly, the New-Object and New-Cog-Object ABS rules are both simulated by the New-Object MultiASP rule, but we can easily distinguish the two cases. Indeed, the New-Object MultiASP rule can distinguish whether the translated ABS cog is empty or not. More precisely, the New-Cog-Object ABS rule is simulated by a New-Object MultiASP on an empty cog, while the New-Object ABS rule is simulated by a New-Object MultiASP rule on a cog that is not empty. There is thus no ambiguity in the simulation. From another point of view, the New-Active MultiASP rule could be the observable rule (and not a

ABS rule	**MultiASP rule**	**Additional MultiASP rules**
Assign-Local	Assign-Local	–
Assign-Field	Assign-Field	–
Await-True	–	Update / – , Invk-Passive, Return-Local Assign-Local-Tmp
Await-False	Invk-Future	–
Release-Cog	–	–
Activate	–	Activate-Thread / (Serve, Invk-Passive,Return-Local, Assign-Local-Tmp)
Read-Fut	–	Set-Hard-Limit, Update / – , Invk-Passive, Return-Local, Set-Soft-Limit Assign-Local-Tmp
New-Object	New-Object	Invk-Passive Assign-Local-Tmp Return-Local
New-Cog-Object	New-Object	New-Active, Assign-Local-Tmp, Invk-Active-Meta, Return
Rendez-vous-Comm	Invk-Active	Invk-Passive, Return-Passive, Assign-Local-Tmp
Return	Return	Return-Local Assign-Local-Tmp

Table A.1 – Summary table of the simulation of ABS in MultiASP. Assign-Local-Tmp represents an Assign-Local on a variable introduced by the translation instead of ABS local variables. In the same way, Invk-Active-Meta means that it is like an Invk-Active but on a method that is not the *execute* method.

silent action) to simulate the New-Cog-Object ABS rule instead of the New-Object MultiASP rule. However, doing so would mean that we create an (active) object in MultiASP that has no equivalent object in ABS. This is why we prefer tracing the New-Object rules in MultiASP.

For the simulation we introduced the Invk-Active-Meta MultiASP rule, that always is a silent action because it only applies to meta-requests that do not exist in ABS, like the *register* and *freshId* requests. Symmetrically, the visible Invk-Active MultiASP rule, that simulates the Rendez-vous-Comm ABS rule, only corresponds to *execute* requests which are the ones that are observable in ABS executions. Again, as we can easily distinguish the requests of the Invk-Active and Invk-Active-Meta rules, then there is no ambiguity in the simulation. A similar remark can be made about the Assign-Local and Assign-Local-Tmp MultiASP rules. The latter is for handling the temporary variables that are introduced by the simulation. Finally, we can now easily list the MultiASP and ABS rules that are involved but not observable in the simulation from ABS to MultiASP:

The silent MultiASP actions are: New-Active, Serve, Invk-Passive, Return-Local, Update, Activate-Thread, Set-Hard-Limit, Set-Soft-Limit, Invk-Active-Meta, and (Assign-Local-Tmp). The other MultiASP rules are the ones that are observable in the simulation.

The silent ABS actions are: Await-True, Release-Cog, Activate, Read-Fut. It is interesting to note that these rules are related to the manipulation of futures and of execution threads in ABS, and that are manipulated differently, or not at all, in MultiASP. For example, a single MultiASP rule is necessary for an object to release a thread whether two rules are necessary in ABS.

Such observable and silent actions in the simulation highlight the aspects that are similarly designed, as well as the aspects that are differently handled in the two active object programming languages. These remarks conclude the insights of the proof of Theorem 4.3.5.

A.2.2 From **MultiASP** to **ABS**

Theorem 4.3.6, page 132 (MultiASP to ABS). Any reduction of the MultiASP translation corresponds to a valid ABS execution.

$$\underline{Cn_0} \to^* \underline{Cn} \Rightarrow \exists Cn.\, Cn_0 \overset{\mathrm{A}}{\to}^{*} Cn \wedge Cn\, \mathcal{R}\, \underline{Cn}$$

Proof sketch of Theorem 4.3.6. This proof sketch verifies that any MultiASP translation of an ABS program corresponds to an ABS execution that is possible. In this direction, the main difficulty is that we introduced additional steps in the reduction. However, for the proof we use the fact that the two active object languages have few concurrent rules: method invocation and future awaiting, and since there is only a single active thread at a time, it prevents local concurrency. Consequently, we know that any sequence of additional MultiASP actions never performs a wait-by-necessity, and runs until the end of the sequence without any observable interruption. Overall, only thread activation, thread passivation, method invocation, and future update can create interleavings. We rely on this knowledge for the proof.

Proving this direction of the simulation is complex, since the translated code is more operational. In particular, we can be in intermediate states in MultiASP, and these must be attached to the right state in ABS. Here, what must be observed is that the translation of an ABS primitive mostly involves a single action that impacts the state of the equivalent ABS program. For example, for remote invocations, solely the *execute* requests impact the request queue of the recipient, and has an effect on the equivalence relation. The principle is to see only those actions and to allow the translation of statements to do the same. For example, in the case of the *execute* remote method invocations, assignments to intermediate variables are ignored in the equivalence, and all the states that precede the execution of the remote method invocation are considered equivalent to the same ABS state. This is mostly handled by ignoring adequately assignments to variables that are not part of the ABS code. A particular other issue is that there is an intermediate state when the request is served. However, the associated ABS method is not started yet, so this case must also be considered in the notion of equivalence. A 'just started' request is equivalent to having the same request in the queue. Finally, it is also important to notice that the theorem no more needs a restriction on future values because when a future access is possible in MultiASP, it can be faithfully simulated in ABS. Indeed, ABS allows the program to have an explicit control on the status of futures.

The goal here is not to do the exhaustive simulation, like for the proof of Theorem 4.3.5, mostly because the technical details will be massively redundant. However, what is important to show here is that the translation ensures that no additional ABS behaviours can be introduced by the translational MultiASP seman-

$$s \approx^{Cn}_{(\sigma+\ell)} [\![s]\!] \qquad\qquad \frac{s \approx^{Cn}_{(\sigma+\ell)} (\textbf{setLimitHard}; s')}{s \approx^{Cn}_{(\sigma+\ell)} s'}$$

$$\frac{\begin{array}{c} s \approx^{Cn}_{(\sigma+\ell)} (\mathit{tmp}{=}z; s') \\ \mathit{tmp} \text{ is a local variable introduced by the translation} \end{array}}{s \approx^{Cn}_{(\sigma+\ell)} s'}$$

$$\frac{\begin{array}{c} s \approx^{Cn}_{(\sigma+\ell)} s' \\ \\ \mathit{tmp} \text{ is a local variable introduced by the translation} \end{array}}{s \approx^{Cn}_{(\sigma+\ell)} (\mathit{tmp}{=}z; s')}$$

$$\frac{\begin{array}{c} s \approx^{Cn}_{(\sigma+\ell+(\mathit{tmp}\mapsto[\![e]\!]_{(\sigma+\ell)}))} s' \\ \mathit{tmp} \text{ is a local variable introduced by the translation} \end{array}}{s \approx^{Cn}_{(\sigma+\ell)} (\mathit{tmp}{=}e; s')}$$

$$\frac{s = (x = v; s_1) \qquad s' = (x = e; [\![s_1]\!]) \qquad v \approx^{Cn}_{\sigma} [\![e]\!]_{(\sigma+\ell)}}{s \approx^{Cn}_{(\sigma+\ell)} s'}$$

Figure A.1 – Refined equivalence of statements for the proof of Theorem 4.3.6.

tics. We provide here the most important points of the proof: the refinement of the equivalence relation $\mathcal{R}$, and a summary of the equivalence between reductions.

1) Adapting the equivalence relation.

To prove that all the MultiASP executions faithfully respect the ABS semantics, the equivalence relation must take into account the fact that several rules are used in MultiASP to simulate a single ABS rule. In practice, it is easy to identify the case where the same statement is triggered both in MultiASP and in ABS, while other MultiASP statements are preparing the main statement being executed, e.g. fetching the identifier and the COG of the targeted object for a method invocation. Thus, we define a new equivalence relation on statements, as shown in Figure A.1.

The first and the last rule of the equivalence are unchanged for the proof of Theorem 4.3.6, but we add new rules in order to discard **setLimitHard** instructions: the thread that have just performed a **setLimitHard** is considered as still in the same state even if one statement is missing. Similarly, the assignments to temporary variables can be added or removed, but when it is removed one can use the assigned value only if it is a simple expression. This is useful to relate in MultiASP the temporary variable no and use it in the remaining. The statement: $no = o; s; x = no$ in MultiASP should be equivalent to: $x = o$ in ABS.

Two other modifications must be made regarding the equivalence of MultiASP and ABS configurations (we recall Figure 4.12). Firstly, considering the direction from ABS to MultiASP, we have as many COG in ABS as active objects in MultiASP. However, in ABS, when a COG is created, it is populated with at least one object, whereas in the translation of the `new` ABS keyword, the instruction that instantiate an active object (the `newActive` MultiASP keyword) is separated from the instantiation of the new object that populates the COG in ABS. Even if, in any case, the thread cannot be interrupted when doing these steps, this particular moment of the execution is a point where the equivalence relation is not verified. Such a situation is ruled out by considering, in Line 1 of Figure 4.12, only the activities that contain at least one object. In other words, the activities that are only populated with a COG active object in MultiASP are not considered to be part of the configuration yet. This can easily be formulated by checking the content of the local store of the activity: if σ only contains the COG object and no object corresponding to an ABS object, then the activity should not be considered in the equivalence. Specifically:

Line 1 of Figure 4.12: $act(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$
has the additional condition: σ *has more than one entry.*

Secondly, there is a floating state when the request has started to be served in MultiASP but the corresponding ABS method has not started to be executed (precisely, when the stack corresponding to this request has a single entry). In this case, the corresponding thread that is currently executed is necessarily the single thread that can be active. This situation must be considered in a way that is similar to the case where the requests are still in queue. Specifically:

Line 4 of Figure 4.12: $\big((f, execute, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \wedge\ \ell'(\mathtt{this}) = o\big)$
has one more case and thus is be replaced by:
$\big((f, execute, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \wedge\ \ell'(\mathtt{this}) = o\ \vee$
$(f, execute, i, m, \overline{v''})_A \mapsto \{\ell''|s''\} \in p \wedge o_\alpha.retrieve(i) = o \wedge \mathrm{bind}(o, m, \overline{v''}) = \{\ell'|s'\}\big)$

Modulo these minor changes in $\mathcal{R}$, the proof of Theorem 4.3.6 is similar to the proof of Theorem 4.3.5, but tedious and without novel aspects. Consequently, we only provide the principles of the proof and Table A.2 for summarising our results on the simulation.

2) **Proof principles.**

Like in the direction of the simulation from ABS to MultiASP, a notion of observability has to be defined, to focus on what is meaningful to consider for equivalence of configurations. In our sense, the most important step to be observable in active object executions is the emission of requests. Indeed, request sending characterises by its own the available objects, the available cogs, the communications, and the variables and statements that enabled the request sending.

We also give proof directions to handle the inconsistencies of future updates for the simulation of MultiASP in ABS. We recall here that the problem is that, due to distributed first class futures, a MultiASP program can have several copies of a future, whereas only one such future exists in ABS. Two directions can be taken to prove that the behaviour of the two programs is still equivalent. In the case we have an atomic future update in MultiASP, then the equivalence is trivial: when an Invk-Future rule in MultiASP is observed, an Await-False in ABS can be exactly observed. However, in the case we do not rely on an atomic future update, as it is likely the case in a distributed setting, an approach in order to still prove the equivalence of the two programs would be to bring the problem back to the trivial case of the atomic future update. To this end, we rely on the causal ordering of events of MultiASP. Indeed, in between the update of the original copy of a future and the update of the last copy of this future, many events could have occurred meanwhile. However, by causal ordering relationship, none of these events are related to the usage of this future's value, because otherwise they would necessarily occur after the considered future update. Consequently, we can safely consider these events as not observable in the current configuration, so the equivalence of the MultiASP and ABS configurations holds in this case. □

We further rely on Table A.2 for commenting the simulation of the MultiASP translation with ABS executions. We review below each of the MultiASP rules. First

MultiASP rule	**ABS rule**	**Additional ABS rules**
Assign-Local	Assign-Local	–
Assign-Local-Tmp	–	–
Assign-Field	Assign-Field	–
Invk-Future	Await-False	Release-Cog
Invk-Passive	–	–/Read-Fut/ Await-True
Activate-Thread	Activate	–
Serve	Activate	–
Invk-Active	Rendez-vous-Comm	–
Invk-Active-Meta	–	–
New-Object in an activity with no ABS object	New-Cog-Object	–
New-Object in a non-empty activity	New-Object	–
New-Active	–	–
Return-Local	–	–
Return	Return	–
Update	–	–
Set-Soft-Limit	–	–
Set-Hard-Limit	–	–

Table A.2 – Summary table of the simulation of MultiASP in ABS.

the Assign-Local and Assign-Field MultiASP rules raise no particular comment, except that we can safely ignore them for the assignment of intermediate variables introduced by the translation. The Invk-Future MultiASP rule only activates if the current thread limit is a soft thread limit, which only happens in the translation of an ABS *await* statement, and solely when the future is not resolved. Indeed, no additional future can be awaited in ABS. Additionally, the cog is released in ABS, which is a silent action that cannot be observed in MultiASP. In the case of the Invk-Passive MultiASP rule, two cases are possible. The corresponding ABS rule can be the Read-Fut rule if the invoked method is $get()$ and the current limit is a hard limit. Alternatively, the corresponding ABS rule can be the Await-True rule if the invoked method is $get()$ and if the current thread limit is a soft thread limit. Lastly, the Invk-Passive MultiASP rule can also be not observable in ABS for all the requests that are introduced by the translation. Here, we use the fact that we do not consider in the proof the ABS local method invocations. Then, the distinction between the Activate-Thread and Serve MultiASP rules is made depending on the status of the request in MultiASP: whether it has started to be served or not. Such a distinction does not exist in ABS. The Invk-Active MultiASP reductions that involve an *execute* request exactly match the Rendez-vous-Comm rule. Here, the preliminary steps and the additional steps performed in the translation are handled by the equivalence relation. Those steps also ensure that the object exists in ABS, and that it can be accessed through its cog. Obviously, the Invk-Active reductions of other methods (that we label Invk-Active-Meta) have no corresponding reduction in ABS. The New-Object MultiASP rule can either correspond to the instantiation of an object in the same cog (the New-Object ABS rule) or in a new cog (the New-Cog-Object ABS rule) if the store of the MultiASP activity in which the object is instantiated is empty (if it only contains the cog object). The Return MultiASP rule is exactly matched with the Return ABS rule, except that in MultiASP the request must be an *execute* request.

Finally, we can now easily list the ABS and MultiASP rules that are involved but not observable in the simulation from MultiASP to ABS:

The silent ABS actions are: Await-True, Read-Fut, Release-Cog. Such rules represent the execution steps that MultiASP does not feature.

The silent MultiASP actions are: New-Active, Invk-Passive, Return-Local, Update, Set-Hard-Limit, Set-Soft-Limit, Invk-Active-Meta, and Assign-Local-Tmp. We can note that except the Invk-Passive MultiASP rule, none of the aforementioned rules needs an additional reduction on the ABS side: the reduced configuration is always equivalent to the corresponding ABS configuration.

Again, observable and silent actions in this direction of the simulation represent the manipulation that are permitted in a program and not in the other, or on the contrary that are intrinsic. These remarks conclude the insights of the proof of Theorem 4.3.6.

Appendix B

Extended Abstract in French

B.1 Introduction

À l'heure du e-commerce, des salons de discussion virtuels, des éditeurs collaboratifs, des plateformes de stockages synchronisées, des jeux en ligne et des objets connectés, chacune de nos habitudes numériques implique des interactions simultatnées et multi-tiers. Ces usages maintenant devenus courants sont supportés par des systèmes informatiques qui évoluent de façon concurrente et distribuée. Les systèmes informatiques ont évolué en profitant de l'avènement du processeur multi-coeur, aujourd'hui présent dans quasiment tous nos ordinateurs et appareils électroniques. Cependant, les langages de programmation qui servent à créer ces systèmes n'ont pas évolué aussi rapidement que le matériel sous-jacent et que les applications que nous utilisons aujourd'hui. C'est tout spécialement le cas pour les langages de programmation qui sont utilisés en production dans l'industrie. En général, les modèles de programmation et les langages de programmation existants manquent de support pour gérer la concurrence des programmes parallèles et pour créer des applications distribuées. Ce sont principalement les *threads*, ou flots d'exécution, qui sont proposés par ces langages de programmation, et qui représentent des séquences d'instructions asynchrones. Tous les langages de programmation modernes offrent un support pour créer et démarrer des flots d'exécution parallèles. Le problème de cette approche est que le programmeur doit avoir une connaissance précise de la manière dont les différents flots d'exécution peuvent être synchronisés de façon sûre. Deux types de bugs peuvent apparaître à cause d'une mauvaise utilisation des threads : les situations de compétitions sur des ressources partagées, résolus par l'utilisation de verrous, et les interblocages résultant la plupart du temps d'une mauvaise synchronisation des verrous. En résumé, l'utilisation des threads place une charge supplémentaire sur les épaules du programmeur.

Les notions de programmation orientée objet et multi-threadée sont souvent embarquées dans un même langage de programmation, bien que leurs concepts soient littéralement opposés. En effet, l'état d'un objet étant encapsulé, il ne devrait être modifié qu'à travers l'exécution des méthodes de l'objet. En autorisant plusieurs flots d'exécution à appeler ces méthodes de façon concurrente, l'état d'un objet devient vulnérable. De plus, lorsque des verrous sont utilisés à l'intérieur d'une classe, ils forcent une exposition des variables sur lesquelles la synchronisation s'applique. Enfin, utiliser les threads dans un environnement distribué n'est pas adapté, car ils peuvent se trouver dans des espaces mémoire différents. Les langages de programmation actuels manquent donc d'abstractions parallèles et distribuées qui soient adaptées aux applications actuelles. Ils ne procurent pas par défaut une exécution parallèle sûre, ni une façon de programmer concurrement qui soit agréable pour le programmeur. Le programmeur devrait être munit de langages et de plateformes d'exécution qui procurent à la fois une sûreté d'exécution et des performances satisfaisantes. Plusieurs moyens d'atteindre ces objectifs sont possibles. Mon travail se focalise sur les moyens qui sont au plus près du programmeur: les modèles de programmation, les langages de programmation et leur APIs. Mon but est de concevoir une expérience de programmation concurrente et distribuée qui soit plaisante, tout en conservant sûreté et performance de l'exécution des programmes.

Dans cette thèse, nous proposons un langage et des outils de développement, réunis dans une même librairie, qui procure les abstractions parallèles et distribuées avec lesquelles le programmeur peut bâtir des applications concurrentes et distribuées robustes. Plus précisément, nous nous basons sur le modèle de programmation à objet actif car il constitue une base solide de programmation concurrente et distribuée dans un contexte orientée objet. Une des caractéristiques des objets actifs est l'absence de mémoire partagée, ce qui les rendent particulièrement adaptés aux environnements distribués. Chaque objet actif possède son propre espace mémoire et ne peut pas accéder celui des autres objets actifs. En conséquence, les objets actifs ne communiquent qu'à travers l'envoi de messages, matérialisés par des appels de méthodes asynchrones. Cette caractéristique rendent les accès à un objet actif plus facile à tracer, et ainsi leur sûreté est aussi plus facile à vérifier. Le modèle de programmation à objets actifs permet formalisation et vérification des programmes car le modèle de communication est bien défini. Cela facilite l'analyse statique des programmes qui sont faits d'objets actifs. Pour ces raisons, le modèle de programmation à objet actif représente une base solide pour la construction de langages de programmation et d'abstractions de haut niveau pour programmer des systèmes concurrents et distribués de plus en plus complexes. Dans cette thèse, nous choisissons une approche globale dans la considération des langages à objets actifs: Nous étudions à la fois l'implantation de ces langages ainsi que leur sémantique. Il existe généralement un décalage entre la formalisation de ces langages et leur implantation, ainsi notre but est de montrer dans cette thèse que cet écart peut néanmoins être comblé en utilisant les bonnes approches et les bons outils.

B.2 Résumé des Développements

Nous démarrons cette thèse avec un aperçu étendu des langages de programmation à objet actif dans le chapitre 2. Nous commençons par présenter les grandes lignes du modèle de programmation et les termes associés. Un objet actif consiste en l'association d'un flot d'exécution privé à un objet. Seul ce flot d'exécution est autorisé à invoquer les méthodes d'un objet actif. Pour invoquer une méthode d'un objet actif, l'appelant doit poster une requête. Cette requête est mise en queue jusqu'à ce qu'il soit possible de l'exécuter : les requêtes sont exécutées séquentiellement et de façon isolée par rapport aux autres requêtes. C'est la raison pour laquelle l'exécution concurrente de plusieurs objets actifs est sûre. En outre, ce modèle permet l'invocation de méthodes asynchrones : l'appelant continue son exécution pendant le temps de traitement de la requête par l'objet actif. En attendant le résultat de la requête, une variable spéciale fait office de conteneur pour le résultat à venir, c'est un futur. Après avoir décrit les principaux avantages et inconvénients des objets actifs, nous entammons la description d'une classification des langages à objet actif, sous trois axes principaux: le modèle d'objet actif utilisé, le modèle d'ordonnancement des requêtes, et la transparence du langage du point de vue du programmeur. Nous présentons ensuite six langages à objet actif majeurs: Creol, JCoBox, ABS, ASP/ProActive, AmbientTalk et Encore. Nous les décrivons en utilisant les notions introduites dans la classification précédemment établie. Ensuite, nous nous focalisons sur les objets actifs multi-threadés. Nous présentons le modèle de programmation à objet multiactif et son implantation dans la librairie ProActive sur laquelle repose cette thèse. Nous présentons aussi brièvement ici la formalisation des objets multiactifs de la librairie ProActive, à travers le language de programmation MultiASP. Enfin, après avoir détaillé les sources variées de déterminisme dans les implantations actuelles des objets actifs, nous positionnons cette thèse dans l'écosystème détaillé précédemment.

Le premier chapitre de contribution de cette thèse, le chapitre 3, concerne l'adaptation de l'ordonnancement des requêtes des objets multiactifs aux attentes du programmeur. Ce chapitre fait l'objet de la publication [3]. En effet, en la présence de plusieurs flots d'exécution au sein d'un objet multiactif, une politique par défaut d'allocation des requêtes aux threads disponibles n'est pas suffisante. Elle ne permet pas d'atteindre les objectifs d'applications haute performance. C'est pour cette raison que nous proposons une extension du modèle de programmation à objet multiactif qui offre au programmeur la possibilité de spécifier, dans un premier temps, le nombre de flots d'exécution disponibles par objet multiactif et de manipuler cette limite dans le but de s'adapter à tout type de situations. Nous proposons dans un deuxième temps un mécanisme de spécification de priorités pour le traitement des requêtes d'un objet multiactif. Pour cela, le programmeur peut appliquer des priorités sous forme d'annotations apportées à la classe concernée. Le modèle de priorité se base sur un graphe de dépendances dont nous donnons les propriétés. Ces mécanismes de spécification, complètement intégrés dans la librairie ProActive, constituent un ordonnanceur de requêtes complet. Nous présentons son architecture logicielle. Pour finir, nous évaluons le mécanisme de spécification de priorités, ainsi que l'efficacité de plusieurs représentations internes pour celui-ci, à travers plusieurs micros bancs de tests. La représentation qui sacrifie une partie de mémoire contre un accès caché aux priorités au moment de l'exécution est la plus performante.

Le chapitre 4 présente un cas d'utilisation particulier des objets multiactifs. Nous proposons dans ce chapitre l'encodage d'un langage de modélisation basé sur un modèle à objet actif, ABS, dans une configuration précise d'objets multiactifs, implanté en ProActive. Ce travail fait l'objet de la publication [4]. Nous implantons un traducteur d'ABS vers ProActive, basé sur un traducteur d'ABS vers Java. La particularité de cette traduction est qu'elle tranforme un programme ABS local en un programme ProActive qui peut s'exécuter de façon distribuée. Les principaux défis de cette traduction se situent dans les différences des deux langages, notamment sur trois points: le modèle d'objet actif utilisé, l'ordonnancement interne des requêtes, et la transparence des appels asynchrones et des futures dans le langage. Ces trois points représentent les axes principaux de la classification que nous donnons au chapitre 2. Dans le but de simuler les objets actifs d'ABS, nous instancions un objet multiactif ProActive par groupe d'objets ABS. Les objets ABS sont représentés par des objets passifs en ProActive. Dans la traduction, l'objet multiactif fait donc office de point d'entrée pour contacter un objet sous-jacent, ce qui rend ce concept particulièrement adapté à la distribution des applications. Nous simulons l'exécution de programmes ABS avec une exécution d'objets multiactifs qui n'ont qu'un seul flot d'exécution actif à la fois. Les autres flots d'exécution, s'il y en a, sont bloqués dans l'attente d'un future. Nous évaluons notre traducteur de façon expérimentale en comparant un programme traduit en ProActive avec un programme traduit par le traducteur Java existant. Nous le comparons aussi avec un programme écrit nativement en ProActive. Le traducteur ProActive introduit un coût additionnel négligeable comparé aux bénéfices de pouvoir exécuter les programmes ABS de façon disitribuée. Dans un deuxième temps, nous prouvons la correction de notre traduction en utilisant la sémantique du langage ABS et la sémantique du langage MultiASP, qui formalise l'exécution des objets multiactifs de la librairie ProActive. Les preuves des deux théorèmes associés sont présentées en annexe de cette thèse. Nous exposons dans cette thèse les éléments pertinents de la preuve de correction : les lemmes utilisés, la relation d'équivalence établie, et les restrictions de la traduction. Nous concluons ce chapitre avec un récapitulatif de la méthodologie de preuve et un retour d'expérience qui montre l'importance des choix de conception qui sont faits au moment de la création d'un langage à objet actif.

Nous nous intéressons ensuite dans le chapitre 5 au développement des applications qui sont faites d'objets multiactifs, ainsi qu'à leur support à l'exécution. Pour cela, nous introduisons dans un premier temps un outil que nous avons développé pour visualiser l'exécution des objets multiactifs d'une application et leur communications. Cet outil permet de réaliser cela après l'exécution de l'application. Les éléments de visualisation sont entièrement intégrés aux notions liées aux objets multiactifs, ce qui en font un outil de débogage adapté à ce modèle de programmation. Tout particulièrement, l'utilisateur de ce débogueur peut visionner la séquence des requêtes qui ont été exécutées par un flot d'exécution particulier d'un objet multiactif, et de voir quelles autres requêtes étaient traitées par celui-ci en parallèle à un moment donné. Notamment, la compatibilité entre les différentes requêtes est mise en valeur et toutes sortes de contrôles permettent à l'utilisateur de se focaliser sur un moment donné de l'exécution d'un objet multiactif. Nous montrons l'utilité de ce débogueur à travers deux cas d'utilisation qui constituent une source d'erreur récurrente dans les systèmes concurrents et distribués : le cas d'un interblocage et le cas d'une situation de compétition pour une ressource partagée.

Dans une deuxième partie de ce chapitre, nous nous intéressons à la tolérance aux pannes des applications basées sur des objets multiactifs. Nous commençons par présenter un protocole de tolérance aux pannes qui prenait en charge les objets actifs de la librairie ProActive. Cependant, l'introduction de plusieurs flots d'exécution dans un objet actif rend ce protocole défaillant. Notre première contribution dans le but d'adapter ce protocole à la tolérance aux pannes des objets multiactifs est de réimplanter le protocole existant uniquement avec des notions liées aux objets multiactifs. Ainsi, les actions de sauvegarde d'état stable (checkpoints) deviennent des requêtes génériques qui peuvent être soumisent aux différentes spécifications possibles des objets multiactifs. Nous définissons des priorités d'exécution adaptées pour ces requêtes, et nous faisons ainsi en sorte qu'elles soient exécutées de façon similaire à l'implantation initiale. Ensuite, nous adaptatons ce protocole de tolérance aux pannes aux objets multiactifs. Le fait qu'un objet multiactif soit constitué de plusieurs flots d'exécution rend plus difficile la détection d'états stable permettant la capture de l'état de l'objet. Nous adaptons le protocole avec une approche qui fait décroître progressivement le nombre de flots d'exécution parallèles d'un objet multiactif. Lorsque que le nombre de flots d'exécution est réduit à un, la capture d'état est alors rendue possible. Cette approche présente des limitations que nous décrivont en fin de chapitre. Notamment, elle peut introduire un interblocage si la demande de capture d'état apparaît lorsqu'une requête réentrante est en train d'être exécutée.

Dans un dernier chapitre, le chapitre 6, nous nous intéressons à un cas d'utilisation pratique qui implique plusieurs des développements qui ont été introduits dans cette thèse. Avant cela, nous présentons le contexte de ce cas d'utilisation qui porte sur les réseaux pair-à-pair de type CAN. Nous introduisons les principales caractéristiques du CAN : l'espace de données de ce type de réseaux est basé sur un espace Cartésien multi-dimensionnel. Chaque pair du réseau gère une partie de cet espace de données. Les requêtes d'insertion et de récupération de ces données sont routées de pair-en-pair, et atteignent leur destination en un nombre borné de sauts. Nous présentons un algorithme de broadcast efficace pour CAN qui fait l'objet de la publication [2]. La particularité de cet algorithme est qu'il génère un nombre optimal de messages, exactement un message par pair appartenant au réseau, et de façon complètement décentralisée. Les propriétés spatiales du CAN sont largement utilisées pour cela. L'idée principale est de restreindre la propagation du message aux pairs voisins qui se trouvent sur une branche de propagation particulière. Pour cela nous raisonnons sur l'espace multi-dimensionnel du CAN de façon récursive. Nous présentons ensuite notre cas d'application qui consiste en une implantation de CAN où les pairs sont représentés avec des objets multiactifs tolérants aux pannes. Nous utilisons pour cela la librairie ProActive, qui contient l'implantation du protocole de tolérance aux pannes pour objet multiactif introduit dans le chapitre précédent. Notre but dans cette application est de lancer un broadcast efficace, et de montrer que, même en présence d'une panne d'un pair durant le broadcast, le broadcast réussit quand même grâce au recouvrement automatique effectué par notre protocole. Nous prouvons ainsi que l'approche qui consiste à rendre les algorithmes distribués tolérants aux pannes au niveau de l'intergiciel est pertinente.

B.3 Conclusion

Depuis plusieurs années, le modèle de programmation à objet actif a représenté la programmation concurrente et sûre, tout en étant contenu dans une abstraction intuitive pour le programmeur. Dernièrement, ce modèle de programmation est même devenu plus adapté à l'ère parallèle grâce aux améliorations apportées à la fois au modèle et à ses différentes implantations. De ce fait, le modèle de programmation à objet actif peut maintenant être aussi efficace localement. Cependant, les améliorations qui ont été apportées dans ce sens ont aussi amené de nouvelles problématiques, telles que la meilleure façon d'exprimer le parallélisme local, sans retomber dans les méandres des threads. De nouveaux défis sont aussi apparus, tels que l'adaptation de l'ordonnancement interne des requêtes. Les outils qui avaient été bâtis autour des objets actifs doivent eux-aussi être repensés. Dans cette thèse, nous avons étudié le modèle de programmation à objet actif augmenté d'exécution parallèle controllée, à travers un méta-langage de programmation. C'est le modèle de programmation à objet multiactif. Nous proposons un cadre de développement complet autour des objets multiactifs qui offre des politiques avancées d'ordonnancement et d'allocation des requêtes, de la tolérance aux pannes et du débogage, tout cela intégré dans une librairie : la librairie ProActive. Nous attachons un intérêt particulier à trois objectifs : l'utilisabilité, la correction et la performance de nos développements. Ces objectifs couvrent le large spectre des modèles de programmation en termes de conception, de spécification et d'exécution. Nous donnons les grandes lignes de notre modèle de programmation à travers des applications concrètes. Tester nos développements dans des environnements réalistes a une importance déterminante, puisque nous nous positionnons sur des applications distribuées, ce qui rend leur performances moins prédictibles. C'est pourquoi nous menons systématiquement une évaluation expérimentale de nos développements. D'autre part, nous formalisons nos travaux afin de démontrer leur efficacité et leurs principales propriétés. De cette manière, nous renforçons les garanties offertes par notre modèle de programmation, en lesquelles le programmeur peut avoir confiance.

Le cadre de travail complet autour des objets multiactifs qui a été amené par cette thèse est prometteur, tant dans un aspect pratique que théorique. Cette thèse peut en effet représenter le point de départ de plusieurs sujets de recherche. Les travaux présentés dans cette thèse sont implantés et utilisables. Les idées apportées autour de l'étude des langages à objet actif peuvent aussi consistuer un point de départ dans la conception de nouveaux modèles de programmation et outils de développement. Dans une perspective à court terme, la librairie ProActive pourra être améliorée. Les priorités appliquées par le programmeur peuvent être rendues plus dynamiques afin de s'adapter aux aléas de l'exécution, par exemple pour résoudre un cas de famine de requête. Concernant les outils construits autour des objets multiactifs, notre débogueur pourra dans le futur procurer plus d'informations sur l'exécution des objets multiactifs. Par exemple, le type de la limite du nombre de flots d'exécution parallèles est une information qui est actuellement manquante dans cet outil. Un support de visualisation en temps réel pourra aussi être développé assez rapidement, grâce à l'implantation modulaire que nous avons fourni pour la classification des événements de l'application. Une évolution à moyen terme que nous planifions est de procurer un débogage d'applications à objet multiactif par points d'arrêt (breakpoints). L'arrêt de l'exécution de l'application quand le point d'arrêt est rencontré peut être matérialisé par la sauvegarde de l'état de ses objets multiactifs. Le mécanisme de sauvegarde d'état qui est inclut dans protocole de tolérance aux pannes peut être utilisé pour cela. Pour que ce mécanisme soit accessible au niveau du programmeur, il faudra élever cette fonctionnalité au niveau de l'API de la librairie ProActive.

Dans sa version courante, la librairie ProActive inclut aussi un support pour l'exécution de composants distribués. Les composants représentent une abstraction plus large que les objets actifs et procurent une plus grande modularité pour la construction de larges applications. Les composants de ProActive peuvent être manipulés graphiquement pour modéliser, vérifier et générer des applications distribuées. L'éditeur graphique VerCors qui permet cela génère du code ProActive en fin de chaîne. Dans ce contexte, l'intégration des notions des objets multiactifs pourra être effectuée au niveau des composants, afin de donner au programmeur la possibilité de manipuler des composants multiactifs.

Enfin, de nombreux travaux théoriques peuvent être basés sur les résultats de cette thèse. Tou d'abord, le travail qui a été démarré sur la tolérance aux pannes des objets multiactifs devra être renforcé par une étude plus formelle. Cette étape fournira une base solide pour résonner plus généralement sur la tolérance aux pannes des systèmes distribués. Cela aidera aussi la conception de l'outil de débogage par points d'arrêt. Une autre piste prévue pour l'enrichissement du cadre de travail que nous proposons est l'analyse statique de programmes à objets multiactifs. Pour l'instant, les annotations appliquées par le programmeur sont interprétées à l'exécution de l'application. En particulier, elles ne sont pas vérifiées avant d'être exécutées, nous faisons principalement confiance au programmeur pour obtenir une exécution correcte de l'application. L'analyse statique des annotations spécifiées par le programmeur donnera plus de garanties à l'exécution, notamment pour détecter des compatibilités qui mèneraient à une situation de compétition sur des ressources potentiellement partagées par plusieurs flots d'exécution. Enfin, des travaux sont actuellement poursuivis pour la détection statique d'interblocages dans les applications basées sur les objets multiactifs de ProActive. Le contexte de ce travail prend racine dans la technique de détection d'interblocage des programmes ABS. L'adaptation de cette technique à MultiASP, le langage qui formalise la librairie ProActive, est pointue, notamment à cause de la transparence du langage qui cache syntaxiquement les principaux emplacements de synchronisation. Notamment, les résultats de ces travaux pourront directement être applicables au backend ABS que nous présentons dans cette thèse.

List of Publications

[1] Frank De Boer and Vlad Serbanescu and Crystal Din and Reiner Hähnle and Ludovic Henrio and Justine Rochas and Einar Broch Johnsen and Ehsan Khamespanah and Marjan Sirjani and Kiko Fernandez Reyes and Albert Mingkun Yang. “A Survey of Active Objects and Actors”. In: *Not Published* () (cit. on pp. 28, 137).

[2] Ludovic Henrio and Fabrice Huet and Justine Rochas. “An Optimal Broadcast Algorithm for Content-Addressable Networks”. In: *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304*. OPODIS 2013. Nice, France: Springer-Verlag New York, Inc., 2013, pp. 176–190 (cit. on pp. 7, 167, 171, 172, 175, 235).

[3] Ludovic Henrio and Justine Rochas. “Declarative Scheduling for Active Objects”. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC ’14. Gyeongju, Republic of Korea: ACM, 2014, pp. 1339–1344 (cit. on pp. 6, 58, 126, 232).

[4] Ludovic Henrio and Justine Rochas. “From Modelling to Systematic Deployment of Distributed Active Objects”. In: *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings.* 2016, pp. 208–226 (cit. on pp. 6, 90, 233).

[5] Ge Song and Justine Rochas and Fabrice Huet and Frédéric Magoulès. “Solutions for Processing K Nearest Neighbor Joins for Massive

Data on MapReduce". In: *23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing*. Turku, Finland, Mar. 2015.

[6] Ge Song and Justine Rochas and Léa El Beze and Fabrice Huet and Frédéric Magoulès. "K Nearest Neighbour Joins for Big Data on MapReduce: a Theoretical and Experimental Analysis". In: *IEEE Transactions on Knowledge and Data Engineering* PP.99 (2016), pp. 1–1.

Bibliography

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986 (cit. on pp. 10, 12, 35, 49).

[Alb+12] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. "COSTABS: A Cost and Termination Analyzer for ABS". In: *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*. PEPM '12. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 151–154 (cit. on p. 27).

[Alb+14] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, German Puebla, and Guillermo Román-Díez. "SACO: Static Analyzer for Concurrent Objects". In: *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Ed. by Erika Ábrahám and Klaus Havelund. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 562–567 (cit. on p. 27).

[Ame+10] Brian Amedro, Denis Caromel, Fabrice Huet, Vladimir Bodnartchouk, Christian Delbé, and Guillermo L. Taboada. "HPC in Java: Experiences in Implementing the NAS Parallel Benchmarks". In: *APPLIED INFORMATICS AND COMMUNICATIONS* (Aug. 2010) (cit. on p. 31).

[And99] Greg R Andrews. *Foundations of Parallel and Distributed Programming*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999 (cit. on p. 2).

[Bac+63] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. Van Wijngaarden, and M. Woodger. “Revised Report on the Algorithm Language ALGOL 60”. In: *Commun. ACM* 6.1 (Jan. 1963). Ed. by P. Naur, pp. 1–17 (cit. on p. 2).

[Bad+06] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. “Grid Computing: Software Environments and Tools”. In: ed. by José C. Cunha and Omer F. Rana. London: Springer London, 2006. Chap. Programming, Composing, Deploying for the Grid, pp. 205–229 (cit. on pp. 29, 165).

[Bau+05] Françoise Baude, Denis Caromel, Christian Delbé, and Ludovic Henrio. “A Hybrid Message Logging-CIC Protocol for Constrained Checkpointability”. In: *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*. Euro-Par’05. Lisbon, Portugal: Springer-Verlag, 2005, pp. 644–653 (cit. on p. 152).

[Bau+09] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. “GCM: a grid extension to Fractal for autonomous distributed components”. In: *Annales des Télécommunications* 64.1-2 (2009), pp. 5–24 (cit. on p. 140).

[Bau+01] Françoise Baude, Alexandre Bergel, Denis Caromel, Fabrice Huet, Olivier Nano, and 1Julien Vayssière. “IC2D: Interactive Control and Debugging of Distribution”. In: *Large-Scale Scientific Computing: Third International Conference, LSSC 2001 Sozopol, Bulgaria, June 6–10, 2001 Revised Papers*. Ed. by Svetozar Margenov, Jerzy Waśniewski, and Plamen Yalamov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 193–200 (cit. on p. 165).

[BHR15] Françoise Baude, Ludovic Henrio, and Cristian Ruz. "Programming distributed and adaptable autonomous components—the GCM/ProActive framework". In: *Software: Practice and Experience* 45.9 (2015), pp. 1189–1227 (cit. on p. 30).

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-oriented Software: The KeY Approach.* Berlin, Heidelberg: Springer-Verlag, 2007 (cit. on p. 27).

[BB16] Nikolaos Bezirgiannis and Frank Boer. "SOFSEM 2016: Theory and Practice of Computer Science: 42nd International Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 23-28, 2016, Proceedings". In: ed. by Mārtiņš Rūsiņš Freivalds, Gregor Engels, and Barbara Catania. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. Chap. ABS: A High-Level Modeling Language for Cloud-Aware Programming, pp. 433–444 (cit. on pp. 27, 53, 137).

[BJ87] K. Birman and T. Joseph. "Exploiting Virtual Synchrony in Distributed Systems". In: *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles.* SOSP '87. Austin, Texas, USA: ACM, 1987, pp. 123–138 (cit. on p. 15).

[Bjø+13] Joakim Bjørk, Frank S. Boer, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. "User-defined Schedulers for Real-time Concurrent Objects". In: *Innov. Syst. Softw. Eng.* 9.1 (Mar. 2013), pp. 29–43 (cit. on p. 87).

[Blo70] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (July 1970), pp. 422–426 (cit. on p. 74).

[BCJ07] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. "A Complete Guide to the Future". In: *Proceedings of the 16th European Conference on Programming.* ESOP'07. Braga, Portugal: Springer-Verlag, 2007, pp. 316–330 (cit. on pp. 12, 14, 22).

[BH13] Francesco Bongiovanni and Ludovic Henrio. “A Mechanized Model for CAN Protocols”. In: *Fundamental Approaches to Software Engineering: 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings.* Ed. by Vittorio Cortellessa and Dániel Varró. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 266–281 (cit. on p. 172).

[Bra+15] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I. Pun, S. Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. “Formal Methods for Multicore Programming: 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures”. In: ed. by Marco Bernardo and Broch Einar Johnsen. Cham: Springer International Publishing, 2015. Chap. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore, pp. 1–56 (cit. on p. 33).

[Cap+05] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. “Grid’5000: a large scale and highly reconfigurable grid experimental testbed”. In: *Grid Computing, 2005. The 6th IEEE/ACM International Workshop on.* 2005, 8 pp.– (cit. on p. 107).

[CCD07] Denis Caromel, Alexandre di Costanzo, and Christian Delbé. “Peer-to-Peer and Fault-tolerance: Towards Deployment-based Technical Services”. In: *Future Gener. Comput. Syst.* 23.7 (Aug. 2007), pp. 879–887 (cit. on p. 140).

[CH05] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects.* 1st. Springer Publishing Company, Incorporated, 2005 (cit. on pp. 28, 165).

[CHS04] Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. “Asynchronous and Deterministic Objects”. In: *Proceedings of the 31st*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '04. Venice, Italy: ACM, 2004, pp. 123–134 (cit. on pp. 28, 52).

[CW16] Elias Castegren and Tobias Wrigstad. "Reference Capabilities for Concurrency Control". In: *Proceedings of 30th European Conference on Object-oriented Programming (ECOOP).* Springer, 2016 (cit. on p. 34).

[Cav+11] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. "Habanero-Java: The New Adventures of Old X10". In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java.* PPPJ '11. Kongens Lyngby, Denmark: ACM, 2011, pp. 51–61 (cit. on p. 38).

[Cha+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. "X10: An Object-oriented Approach to Non-uniform Cluster Computing". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications.* OOPSLA '05. San Diego, CA, USA: ACM, 2005, pp. 519–538 (cit. on p. 38).

[Cla+08] 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. Bangalore, India: Springer-Verlag, 2008, pp. 139–154 (cit. on p. 33).

[Con63] Melvin E. Conway. "Design of a Separable Transition-diagram Compiler". In: *Commun. ACM* 6.7 (July 1963), pp. 396–408 (cit. on p. 17).

[CVCGV13] Alfons Crespo, Juan Antonio Vila Carbó, and M Garcia-Valls. "Resource management for mobile operating systems based on the Active Object model". In: *Computer Systems Science and Engineering* 28.4 (2013), pp. 225–235 (cit. on p. 12).

[Cut+07] T. V. Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. D. Meuter. “AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks”. In: *Chilean Society of Computer Science, 2007. SCCC ’07. XXVI International Conference of the.* 2007, pp. 3–12 (cit. on pp. 31, 32).

[DBS84] A. Ciuffoletti D. Briatico and L. Simoncini. “A distributed domino-effect free recovery algorithm”. In: *Proceedings of the International Symposium on Reliability, Distributed Software and Databases.* 1984, pp. 207–215 (cit. on p. 153).

[DMN68] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *SIMULA 67 Common Base Language, (Norwegian Computing Center. Publication).* 1968 (cit. on p. 17).

[DN66] Ole-Johan Dahl and Kristen Nygaard. “SIMULA: An ALGOL-based Simulation Language”. In: *Commun. ACM* 9.9 (Sept. 1966), pp. 671–678 (cit. on p. 2).

[DP15] Mads Dam and Karl Palmskog. “Location-independent Routing in Process Network Overlays”. In: *Serv. Oriented Comput. Appl.* 9.3-4 (Sept. 2015), pp. 285–309 (cit. on p. 127).

[DKVCD12] Joeri De Koster, Tom Van Cutsem, and Theo D’Hondt. “Domains: Safe Sharing Among Actors”. In: *Proceedings of the 2Nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions.* AGERE! 2012. Tucson, Arizona, USA: ACM, 2012, pp. 11–22 (cit. on p. 15).

[DG08] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113 (cit. on p. 106).

[Ded+06] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D’Hondt, and Wolfgang De Meuter. “Ambient-Oriented Programming in Ambienttalk”. In: *Proceedings of the 20th European Conference on Object-Oriented Programming.* ECOOP’06. Nantes, France: Springer-Verlag, 2006, pp. 230–254 (cit. on p. 31).

[Del07] Christian Delbé. “Tolérance aux pannes pour objets actifs asynchrones : protocole, modèle et expérimentations”. PhD thesis. 2007, 1 vol. (xvi–196 p.) (Cit. on pp. 155, 159).

[DV14] Travis Desell and Carlos A. Varela. “Concurrent Objects and Beyond: Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday”. In: ed. by Gul Agha, Atsushi Igarashi, Naoki Kobayashi, Hidehiko Masuhara, Satoshi Matsuoka, Etsuya Shibayama, and Kenjiro Taura. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. Chap. SALSA Lite: A Hash-Based Actor Runtime for Efficient Local Concurrency, pp. 144–166 (cit. on p. 38).

[DBH15] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. “Automated Deduction - CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings”. In: ed. by P. Amy Felty and Aart Middeldorp. Cham: Springer International Publishing, 2015. Chap. KeY-ABS: A Deductive Verification Tool for the Concurrent Modelling Language ABS, pp. 517–526 (cit. on p. 27).

[Din+15] Crystal Chang Din, S. Lizeth Tapia Tarifa, Reiner Hähnle, and Einar Broch Johnsen. “Formal Methods and Software Engineering: 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings”. In: ed. by Michael Butler, Sylvain Conchon, and Fatiha Zaïdi. Cham: Springer International Publishing, 2015. Chap. History-Based Specification and Verification of Scalable Concurrent and Distributed Systems, pp. 217–233 (cit. on p. 27).

[EAH04] Sameh El-Ansary and Seif Haridi. “An overview of structured P2P overlay networks”. In: *Swedish Institute of Computer Science and Royal Institute of Technology* (2004) (cit. on p. 168).

[FRCS16] Kiko Fernandez-Reyes, Dave Clarke, and Daniel S.McCain. “ParT: an asynchronous parallel abstraction”. In: *Coordination Models and Languages, 18th International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on*

Distributed Computing Techniques, DisCoTec 2016, Crete, Greece, June 6-9, 2016. Proceedings. 2016 (cit. on p. 34).

[FF99] C. Flanagan and Mattias Felleisen. "The Semantics of Future and an Application". In: *Journal of Functional Programming* 9.1 (Jan. 1999), pp. 1–31 (cit. on p. 12).

[GLL15] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. "A framework for deadlock detection in core ABS". In: *Software & Systems Modeling* (2015), pp. 1–36 (cit. on pp. 27, 110, 111, 194).

[Gia+16] Elena Giachino, Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. "Actors may synchronize, safely!" In: *Principles and Practice of Declarative Programming. (to appear).* 2016 (cit. on p. 194).

[Gmb03] YourKit GmbH. *YourKit Java Profiler.* 2003. URL: `https://www.yourkit.com/java/profiler/features/` (visited on 2016) (cit. on p. 150).

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983 (cit. on p. 2).

[Gör14] A. Göransson. *Efficient Android Threading: Asynchronous Processing Techniques for Android Applications.* Programming / Android. Oreilly & Associates Incorporated, 2014 (cit. on p. 12).

[GB13] Olivier Gruber and Fabienne Boyer. "Ownership-Based Isolation for Concurrent Actors on Multi-core Machines". In: *Proceedings of the 27th European Conference on Object-Oriented Programming.* ECOOP'13. Montpellier, France: Springer-Verlag, 2013, pp. 281–301 (cit. on p. 37).

[Häh13] Reiner Hähnle. "Formal Methods for Components and Objects: 11th International Symposium, FMCO 2012, Bertinoro, Italy, September 24-28, 2012, Revised Lectures". In: ed. by Elena Giachino, Reiner Hähnle, Frank S. de Boer, and Marcello M. Bonsangue. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. Chap. The Abstract Behavioral Specification Language: A Tutorial Introduction, pp. 1–37 (cit. on pp. 105, 124).

[Hal12] Philipp Haller. "On the Integration of the Actor Model in Mainstream Technologies: The Scala Perspective". In: *Proceedings of the 2Nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions.* AGERE! 2012. Tucson, Arizona, USA: ACM, 2012, pp. 1–6 (cit. on p. 36).

[HO09] Philipp Haller and Martin Odersky. "Scala Actors: Unifying Thread-based and Event-based Programming". In: *Theor. Comput. Sci.* 410.2-3 (Feb. 2009), pp. 202–220 (cit. on p. 36).

[Hal85] Robert H. Halstead Jr. "MULTILISP: A Language for Concurrent Symbolic Computation". In: *ACM Trans. Program. Lang. Syst.* 7.4 (Oct. 1985), pp. 501–538 (cit. on p. 12).

[HL06] Max Haustein and Klaus-Peter Löhr. "JAC: declarative Java concurrency". In: *Concurrency and Computation: Practice and Experience* 18.5 (2006), pp. 519–546 (cit. on p. 87).

[HSF15] Yaroslav Hayduk, Anita Sobe, and Pascal Felber. "Distributed Applications and Interoperable Systems: 15th IFIP WG 6.1 International Conference, DAIS 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings". In: ed. by Alysson Bessani and Sara Bouchenak. Cham: Springer International Publishing, 2015. Chap. Dynamic Message Processing and Transactional Memory in the Actor Model, pp. 94–107 (cit. on p. 18).

[Hay+13] Yaroslav Hayduk, Anita Sobe, Derin Harmanci, Patrick Marlier, and Pascal Felber. "Speculative Concurrent Processing with Transactional Memory in the Actor Model". In: *Proceedings of the 17th International Conference on Principles of Distributed Systems - Vol-*

ume 8304. OPODIS 2013. Nice, France: Springer-Verlag New York, Inc., 2013, pp. 160–175 (cit. on p. 18).

[HHI12] Ludovic Henrio, Fabrice Huet, and Zsolt István. *A Language for Multi-threaded Active Objects*. Research Report RR-8021. INRIA, July 2012 (cit. on pp. 45, 75).

[HHI13] Ludovic Henrio, Fabrice Huet, and Zsolt István. "Coordination Models and Languages: 15th International Conference, COORDINATION 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings". In: ed. by Rocco Nicola and Christine Julien. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. Chap. Multi-threaded Active Objects, pp. 90–104 (cit. on pp. 39, 46, 75, 113).

[HK15] Ludovic Henrio and Florian Kammüller. *Multiactive objects: Formalisation of the semantics with MultiASP*. 2015. URL: `http://www-sop.inria.fr/members/Ludovic.Henrio/misc.html` (visited on 03/2015) (cit. on p. 46).

[HKM16] Ludovic Henrio, Oleksandra Kulankhina, and Eric Madelaine. "Integrated environment for verifying and running distributed components". In: *Proc. International Conference on Fundamental Approaches to Software Engineering (FASE 2016)*. LNCS. Springer, 2016 (cit. on pp. 30, 193).

[Hen+11] Ludovic Henrio, Muhammad Uzair Khan, Nadia Ranaldo, and Eugenio Zimeo. "First Class Futures: Specification and Implementation of Update Strategies". In: *Proceedings of the 2010 Conference on Parallel Processing*. Euro-Par 2010. Ischia, Italy: Springer-Verlag, 2011, pp. 295–303 (cit. on p. 28).

[Hen+14] Ludovic Henrio, Oleksandra Kulankhina, Dongqian Liu, and Eric Madelaine. "Verifying the correct composition of distributed components: Formalisation and Tool". In: *Proceedings 13th International Workshop on Foundations of Coordination Languages and*

Self-Adaptive Systems, FOCLASA 2014, Rome, Italy, 6th September 2014. 2014, pp. 69–85 (cit. on p. 141).

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. “Featherweight Java: A Minimal Core Calculus for Java and GJ”. In: *ACM Trans. Program. Lang. Syst.* 23.3 (May 2001), pp. 396–450 (cit. on p. 46).

[IS14] Shams Imam and Vivek Sarkar. “Habanero-Java Library: A Java 8 Framework for Multicore Programming”. In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools.* PPPJ ’14. Cracow, Poland: ACM, 2014, pp. 75–86 (cit. on p. 38).

[IS12] Shams M. Imam and Vivek Sarkar. “Integrating Task Parallelism with Actors”. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications.* OOPSLA ’12. Tucson, Arizona, USA: ACM, 2012, pp. 753–772 (cit. on p. 18).

[Inc12] Typesafe Inc. *Akka Documentation: Release 2.0.2.* 2012. URL: `http://doc.akka.io/docs/akka/2.0.2/Akka.pdf` (visited on 10/2012) (cit. on p. 36).

[JOY06] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. “Creol: A Type-safe Object-oriented Model for Distributed Concurrent Systems”. In: *Theor. Comput. Sci.* 365.1 (Nov. 2006), pp. 23–66 (cit. on p. 22).

[JST15] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. “Integrating deployment architectures and resource consumption in timed object-oriented models”. In: *Journal of Logical and Algebraic Methods in Programming* 84.1 (2015). Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011)Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011, pp. 67 –91 (cit. on p. 27).

[Joh+11] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. “ABS: A Core Language for Abstract Behavioral Specification”. In: *Proceedings of the 9th International Conference on Formal Methods for Components and Objects*. FMCO’10. Graz, Austria: Springer-Verlag, 2011, pp. 142–164 (cit. on pp. 25, 111).

[Jon87] James V. Jones. *Integrated Logistics Support Handbook*. Blue Ridge Summit, PA, USA: TAB Books, 1987 (cit. on p. 151).

[KSA09] Rajesh K. Karmani, Amin Shali, and Gul Agha. “Actor Frameworks for the JVM Platform: A Comparative Analysis”. In: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. PPPJ ’09. Calgary, Alberta, Canada: ACM, 2009, pp. 11–20 (cit. on p. 35).

[Kay93] Alan C. Kay. “The Early History of Smalltalk”. In: *SIGPLAN Not.* 28.3 (Mar. 1993), pp. 69–95 (cit. on p. 2).

[Kha+15] Ehsan Khamespanah, Marjan Sirjani, Zeynab Sabahi Kaviani, Ramtin Khosravi, and Mohammad-Javad Izadi. “Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system”. In: *Science of Computer Programming* 98, Part 2 (2015). Special Issue on Programming Based on Actors, Agents and Decentralized Control, pp. 184 –204 (cit. on p. 36).

[Khv15] Pavlo Khvorostov. “A viewer tool for multi-active object”. MA thesis. Universite Nice Sophia Antipolis, Aug. 2015 (cit. on p. 147).

[LS96] R. Greg Lavender and Douglas C. Schmidt. “Pattern Languages of Program Design 2”. In: ed. by John M. Vlissides, James O. Coplien, and Norman L. Kerth. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. Chap. Active Object: An Object Behavioral Pattern for Concurrent Programming, pp. 483–499 (cit. on pp. 10, 29).

[Lee06] Edward A. Lee. “The Problem with Threads”. In: *Computer* 39.5 (May 2006), pp. 33–42 (cit. on p. 2).

[LL13] Mohsen Lesani and Antonio Lain. “Semantics-preserving Sharing Actors”. In: *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control.* AGERE! 2013. Indianapolis, Indiana, USA: ACM, 2013, pp. 69–80 (cit. on p. 15).

[MTS05] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. “Concurrency Among Strangers: Programming in E As Plan Coordination”. In: *Proceedings of the 1st International Conference on Trustworthy Global Computing.* TGC’05. Edinburgh, UK: Springer-Verlag, 2005, pp. 195–229 (cit. on pp. 15, 31).

[Mor+13] Benjamin Morandi, Mischael Schill, Sebastian Nanz, and Bertrand Meyer. “Prototyping a Concurrency Model”. In: *Proceedings of the 2013 13th International Conference on Application of Concurrency to System Design.* ACSD ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 170–179 (cit. on p. 37).

[NSS06] J. Niehren, J. Schwinghammer, and G. Smolka. “A Concurrent Lambda Calculus with Futures”. In: *Theor. Comput. Sci.* 364.3 (Nov. 2006), pp. 338–356 (cit. on p. 12).

[NB14] Behrooz Nobakht and Frank S. Boer. “Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications: 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II”. In: ed. by Tiziana Margaria and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. Chap. Programming with Actors in Java 8, pp. 37–53 (cit. on p. 38).

[Nob+12] Behrooz Nobakht, Frank S. De Boer, Mohammad Mahdi Jaghoori, and Rudolf Schlatte. “Programming and Deployment of Active Objects with Application-level Scheduling”. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing.* SAC ’12. Trento, Italy: ACM, 2012, pp. 1883–1888 (cit. on p. 88).

[Pel14] Laurent Pellegrino. “Pushing dynamic and ubiquitous event-based interactions in the Internet of services : a middleware for event clouds”. Theses. Université Nice Sophia Antipolis, Apr. 2014 (cit. on pp. 77, 175).

[Per85] Radia Perlman. “An Algorithm for Distributed Computation of a Spanningtree in an Extended LAN”. In: *Proceedings of the Ninth Symposium on Data Communications.* SIGCOMM ’85. Whistler Moutain, British Columbia, Canada: ACM, 1985, pp. 44–53 (cit. on p. 171).

[Rat+01a] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. “A Scalable Content-addressable Network”. In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications.* SIGCOMM ’01. San Diego, California, USA: ACM, 2001, pp. 161–172 (cit. on pp. 168, 186).

[Rat+01b] Sylvia Ratnasamy, Mark Handley, Richard M. Karp, and Scott Shenker. “Application-Level Multicast Using Content-Addressable Networks”. In: *Proceedings of the Third International COST264 Workshop on Networked Group Communication.* NGC ’01. London, UK, UK: Springer-Verlag, 2001, pp. 14–29 (cit. on pp. 171, 186).

[SPH10] Jan Schäfer and Arnd Poetzsch-Heffter. “JCoBox: Generalizing Active Objects to Concurrent Components”. In: *Proceedings of the 24th European Conference on Object-oriented Programming.* ECOOP’10. Maribor, Slovenia: Springer-Verlag, 2010, pp. 275–299 (cit. on pp. 23, 24).

[SPM16] M. Schill, C. M. Poskitt, and B. Meyer. “An Interference-Free Programming Model for Network Objects”. In: *Proceedings of the 18th International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques.* DisCoTec 2016. Heraklion, Greece, 2016 (cit. on p. 37).

[STDM14] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. “Parallel Actor Monitors: Disentangling Task-level Parallelism from Data Partitioning in the Actor Model”. In: *Sciense of Computer Programming* 80 (Feb. 2014), pp. 52–64 (cit. on p. 18).

[SG10] B. Schroeder and G. Gibson. “A Large-Scale Study of Failures in High-Performance Computing Systems”. In: *IEEE Transactions on Dependable and Secure Computing* 7.4 (2010), pp. 337–350 (cit. on p. 151).

[Ser+16] V. Serbanescu, K. Azadbakht, F. de Boer, C. Nagarajagowda, and B. Nobakht. “A design pattern for optimizations in data intensive applications using ABS and JAVA 8”. In: *Concurrency and Computation: Practice and Experience* 28.2 (2016). cpe.3480, pp. 374–385 (cit. on pp. 27, 38, 53, 137).

[Ser+14] Vlad Serbanescu, Chetan Nagarajagowda, Keyvan Azadbakht, Frank Boer, and Behrooz Nobakht. “Adaptive Resource Management and Scheduling for Cloud Computing: First International Workshop, ARMS-CC 2014, held in Conjunction with ACM Symposium on Principles of Distributed Computing, PODC 2014, Paris, France, July 15, 2014, Revised Selected Papers”. In: ed. by Florin Pop and Maria Potop-Butucaru. Cham: Springer International Publishing, 2014. Chap. Towards Type-Based Optimizations in Distributed Applications Using ABS and JAVA 8, pp. 103–112 (cit. on pp. 38, 109).

[Sir+05] M. Sirjani, F. de Boer, A. Movaghar, and A. Shali. “Extended Rebeca: a component-based actor language with synchronous message passing”. In: *Fifth International Conference on Application of Concurrency to System Design (ACSD’05).* 2005, pp. 212–221 (cit. on p. 36).

[Sir+04] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. “Modeling and Verification of Reactive Systems Using Rebeca”. In: *Fundam. Inf.* 63.4 (June 2004), pp. 385–410 (cit. on p. 36).

[SM08] Sriram Srinivasan and Alan Mycroft. "Kilim: Isolation-Typed Actors for Java". In: *Proceedings of the 22Nd European Conference on Object-Oriented Programming.* ECOOP '08. Paphos, Cypress: Springer-Verlag, 2008, pp. 104–128 (cit. on p. 37).

[Sto+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications". In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications.* SIGCOMM '01. San Diego, California, USA: ACM, 2001, pp. 149–160 (cit. on p. 186).

[TDJ13] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. "Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?" In: *Proceedings of the 27th European Conference on Object-Oriented Programming.* ECOOP'13. Montpellier, France: Springer-Verlag, 2013, pp. 302–326 (cit. on p. 60).

[TMY94] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. "ABCL/f: A Future-Based Polymorphic Typed Concurrent Object-Oriented Language - Its Design and Implementation". In: *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms.* American Mathematical Society, 1994, pp. 275–292 (cit. on p. 12).

[VWW96] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG (2Nd Ed.)* Ed. by Joe Armstrong. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996 (cit. on p. 36).

[WNM15] Scott West, Sebastian Nanz, and Bertrand Meyer. "Efficient and Reasonable Object-oriented Concurrency". In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* PPoPP 2015. San Francisco, CA, USA: ACM, 2015, pp. 273–274 (cit. on p. 37).

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. "A Distributed Object Model for the javaTM System". In: *Proceedings of the 2Nd*

Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 2. COOTS'96. Toronto, Ontario, Canada: USENIX Association, 1996, pp. 17–17 (cit. on pp. 25, 29).

[WDS12] Peter Y. H. Wong, Nikolay Diakov, and Ina Schaefer. "Modelling Adaptable Distributed Object Oriented Systems Using the HATS Approach: A Fredhopper Case Study". In: *Proceedings of the 2011 International Conference on Formal Verification of Object-Oriented Software*. FoVeOOS'11. Turin, Italy: Springer-Verlag, 2012, pp. 49–66 (cit. on p. 54).

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. "Object-oriented Concurrent Programming ABCL/1". In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPLSA '86. Portland, Oregon, USA: ACM, 1986, pp. 258–268 (cit. on p. 12).

List of Acronyms

API	Application Programming Interface	191
CAN	Content-Addressable Network	7
CIC	Communication-Induced Checkpointing	152
DNS	Domain Name System	103
FIFO	First In First Out	11
HPC	High Performance Computing	28
IDE	Integrated Development Environment	27
IP	Internet Protocol	20
JVM	Java Virtual Machine	28
RMI	Remote Method Invocation	29
RPC	Remote Procedure Call	31
SIMD	Single Instruction Multiple Data	34
TTC	Time To Checkpoint	152
URL	Uniform Resource Locator	30