Formal Models for Programming and Composing Correct Distributed Systems

Ludovic Henrio

UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS

École Doctorale STIC

Sciences et Technologies de l'Information et de la Communication

Formal Models for Programming and Composing Correct Distributed Systems

Mémoire de Synthèse présenté à l'Université de Nice Sophia Antipolis pour l'obtention d'une

HABILITATION À DIRIGER LES RECHERCHES

Spécialité Informatique

par

Ludovic Henrio

soutenue le 19 Juillet 2012

Jury:

Président du Jury	Pr. Michel Riveill	Université de Nice-Sophia-Antipolis - I3S, France
Rapporteurs	Pr. Gordon Blair	Lancaster University, United Kingdom
	Dr. Pascal Poizat	Université d'Evry Val d'Essonne, France
	Pr. Davide Sangiorgi	University of Bologna, Italy
Examinateurs	Dr. Fabienne Boyer	Université Joseph Fourier - LIG, France
	Dr. Alan Schmitt	INRIA Rennes, France
Invité	Dr. Eric Madelaine	INRIA Sophia-Antipolis, France

Contents

List of Figures

Remerciements

Tout d'abord je tiens à m'excuser de ne pas nommer explicitement avec un petit mot de remerciement personnalisé chacune des personnes que je remercie ci-dessous, la liste serait surement trop longue bien qu'incomplète si je tentais de le faire. Sachez néamoins que je pense intensément à chacun d'entre vous au moment d'écrire ces quelques lignes.

Mes remerciement vont tout d'abord à mes collègues et de façon plus générale à tous les chercheurs et étudiants avec qui j'ai travaillé pendant ces 10 dernières années. Tout particulièrement je tiens à remercier les doctorants de l'équipe OASIS, qu'ils aient été officiellement ou non sous ma responsabilité, car c'est souvent eux qui m'ont apporté de nombreux sujets de recherche et de nouvelles applications possibles à mes équations qu'ils trouvaient pourtant incompréhensibles en arrivant dans l'équipe. Mon travail n'aurait souvent que bien peu d'originalité si avec eux nous n'avions aussi intimement lié théorie et pratique. Bien sûr, je tiens aussi à remercier mes collègues, permanents de l'équipe OASIS, qui ont accordé beaucoup de confiance au jeune chercheur que j'ai été, et avec qui travailler et surtout échanger des idées est un réel bonheur quotidien. Mais la recherche ne se situe pas que dans une petite équipe, et il serait injuste de ne pas mentionner mes collègues étranger avec qui j'ai pu travailler que ce soit au travers de projets, comme Marco Danelutto, Christian Pérez, et Vladimir Getov pour ne citer qu'eux, ou simplement par la volonté de travailler ensemble, comme Florian Kammüller. A tous les chercheurs mentionnés ci-dessus, avec lesquels j'espère poursuivre des collaborations, et spécialement à ceux qui sont devenus un peu plus que des collègues, je souhaite leur adresser un immense Merci!

Au moment des remerciements on pense aussi à ce qu'il y a eu avant, car ma thèse me semble encore proche, et je ne serais pas devenu le chercheur que je suis sans les conseils de mes deux directeurs de thèse, avec qui j'ai toujours des discussions fort intéressantes, bien qu'elles portent sur des sujets de plus en plus opposés. Une pensée émue me vient pour Isabelle Attali qui nous a quitté tragiquement en 2004 et a su créer cette équipe dans laquelle je me sens vraiment à ma place.

Pour revenir au présent, je remercie aussi très sincèrement mes rapporteurs et tous les membres de mon jury d Habilitation, qui ont accepté de passer du temps à lire, écouter et évaluer mes travaux.

Et puis la science n'est pas tout dans la vie, malgré la beauté de la recherche il est bon d'aller chercher un peu de notre bonheur ailleurs et pour cela je remercie énormément tous mes amis pour ces moments passés ensemble qui font de la vie un tout. Que ce soit à travers une soirée au cinéma, une soirée passée à jouer entre amis, ou un trail en montagne (qu'il fasse 20 ou 300km), vous m'avez énormément apporté! Aussi, mes parents et ma famille ont su m'apporter le soutien, la complicité et la tendresse pendant tant d'années, il me semblerait inconcevable de ne pas avoir une forte pensée pour eux au moment de ces remerciements. Enfin, mes pensées vont par dessus tout à Marine qui a su m'apporter son support et sa tendresse pendant ces dernières années, pendant lesquelles nous avons déjà partagé tant de choses.

Chapter 1

Introduction

My works aim at a common practical objective: *help the programmer to write distributed applications that run correctly*. My contribution to this objective is to provide theoretical models of languages, runtime platforms and algorithms for distributed computing.

Programming distributed applications is a difficult task. The distributed application developer has to face both concurrency issues and location-related issues. The programming paradigm I have studied the most is the active object programming model. The active object paradigm [LS96] [16] provides a solution for easing the programming of distributed applications by abstracting away the notions of concurrency and of object location. Active objects are similar to actors [AMST97, Agh86], but better integrated with the notion of objects. Active objects are isolated entities that are manipulated by a single thread; they communicate between themselves by asynchronous method invocations (called requests). Active objects act as the unit of distribution and of concurrency, naturally abstracting away communication between remote entities and local availability of data. One of the key feature of active object models is the notion of *futures* [Hal85]: a future is a placeholder for an object that is not yet available; this construct is very convenient for easily expressing concurrent or distributed programs. We chose to have transparent futures so that the programmer does not have to explicitly manipulate futures: if a future object is accessed while the object is not yet available, the program is stuck. The advantage of this approach is that while synchronisation between entities is simple, the programmer does not have to worry about which objects can be a future, and the result of a remote invocation is only waited at the moment when it is really needed. During my thesis, I proposed a calculus formalising such active objects, it is called ASP [16]. Of course, when synchronisation between remote entities gets more complex, deadlocks can appear, and in that case the programmer has to worry about which object is a future or not. In order to detect such deadlocks, but also to guarantee safety of program execution in general, we studied behavioural specification and verification of distributed applications.

We designed behavioural models for active object based distributed applications. Our objective is to provide tools so that the programmer can specify the behaviour of his programs and verify its correctness. For example he can verify the absence of deadlocks or the ability of a program to provide an answer to a given request. We designed first a generic formal specification language allowing hierarchical composition of parameterised labelled transition systems, it is called *pNets*. The specification of an active object or component system is obtained by composition of basic blocks expressing the behaviour of the individual methods of the application. The behaviour of those methods could be either computed by static analysis of the source code, or directly specified by the programmer. In contrast, in our design of a new specification language called JDC, we studied the possibility of generating the behavioural specification. Most of our work on behavioural specification consisted in automatically generating the behaviour of a complete distributed application from the specification of pieces of applicative code. We are able to generate the behaviour of active objects with futures, asynchronous method invocations, FIFO service of requests or specific service policies, ... The model we generate corresponds to the semantics of the active objects as specified by the ASP calculus. Then, from the complete behavioural specification of an application expressed as a pNet, we can generate a finite instance of this model in a language that can be model-checked by an existing model-checker.

Providing tools and theoretical foundations for programming and verifying active objects is thus one of the key achievements of my work. I think the properties of active objects help the distributed programmer with the writing of his applications and the verification of their correctness. However, object-oriented programming has its limits in term of code re-usability and of runtime adaptation. For this, software components have been designed to provide composition framework raising the level of abstraction

compared to objects. Components split the application programming into two phases: the writing of basic business code (in our case, we could write this code as active objects for example), and the composition phase consisting in plugging together the basic blocks programmed above. To scale better and ease the programming of large applications, we focused on a *hierarchical* component model, allowing each component to be the composition of other components. In the context of the CoreGrid European Network of Excellence, we designed a component model called GCM (Grid Component Model); GCM is an adaptation of Fractal [BCL+04] for large scale distributed computing.

By nature, using components to compose applications restricts a little the expressivity by constraining the programmer to identify the dependencies between components statically. While this limits a little the programming model expressivity, it eases the design of large models and in our case it provides a static view of the code dependencies. In our model, components also act as the unit of distribution and concurrency. This way, the static view of the component system is in fact a static view of the distributed entities in which binding between component interfaces are the only places where communications occur. Also as a component is the unit of distribution and concurrency, it can be safely migrated from a machine to another and the deployment process consists in stating which component is placed where. This static view is also crucial to perform behavioural specification as it becomes trivial to identify where remote invocations occur (only component interfaces need to be equipped with the behaviour of asynchronous method calls), when futures are created, etc. This is why most of our developments on behavioural specification focused on distributed components.

The static view enforced by the component models is quite often too restrictive, as application structure need to evolve dynamically, in particular when facing changes in the execution environment. Some component models, including the GCM, address these issues of *adaptiveness* as a separated concern relatively to the business logic: the business code of the component does not have to deal with the structural changes of the component application. Those structural changes, called reconfiguration of the component assembly, are programmed separately, and are considered as non-functional concerns. Both the management and the enactment of structural changes are triggered at the non-functional level. This separation of concern increases the re-usability of applicative code, but also eases the programming of large scale highly adaptive applications. Indeed, in this context, as adaptation code is separated from business code, it is possible to design much smarter and generic adaptation procedures. Those procedures can even become smart and generic enough to allow the component system to take reconfiguration decisions in an autonomous manner; that is why component programming is a good candidate for realising *autonomic* computing. In autonomic computing, each entity is able to adapt itself to changes in the runtime environment, or in the quality of service desired. In this context, we extended GCM with a precise specification and structural constructs for defining non-functional concerns, but we also worked at the design and implementation of correct and distributed reconfiguration procedures.

My contributions are quite theoretical, but I particularly took care that they are also directly applicable to real middlewares, libraries, languages, or runtime platforms. Indeed, the results of our theoretical study have often been applied to the design and implementation of the ProActive[1] library, and of the ProActive/GCM component framework. Most of the time, I rely on a classical programming language theory approach, specifying the runtime semantics of the considered paradigm and proving its properties. What is important here is that the proven results generally had a direct impact on the implementation of the middlewares and platforms implemented in the OASIS team, or in the design of our behavioural models. In my work, application and theory have a strong and constant interaction, making the theoretical parts sometimes less elegant in order to stick to the real implementation, but also making the implementation sometimes less efficient in order to guarantee crucial properties of the programming model. The big challenge addressed partially here is indeed the guarantee of the correct program behaviour. Such a guarantee is the composition of: specification of programming languages and of their runtime semantics ensuring the correctness of the runtime platform; and behavioural specification and verification of the program ensuring the correctness of the application itself.

In my work, I designed an active object calculus formalising the notions used in active object languages, this calculus is called ASP. The ProActive library can be considered as an implementation of this formal model, and ASP acted as a strong guide for the implementation of some crucial features of the ProActive library that is why I actively took part in the design of some crucial functionalities of this middleware. Then, after the definition of the GCM component model, ProActive was extended to provide the reference implementation of GCM, not only did I took part in the design of the model and on the way it is implemented, but we also provided a formal model of the reference implementation (GCM/ProActive) to be able to prove the properties of the implementation. The formal model mentioned above also contributed to the generation of behavioural models

[1] ProActive [CDdCL06] is a Java library implementing active object that was originally developed in the OASIS team

for GCM components. Indeed, we quite massively use the proven properties on the runtime model to choose the right behavioural model but also to optimise the size or complexity of this model which helps us reduce the state-explosion problem inherent to the model-checking approach.

As guarantee of correctness is a strong guiding line of my research, it was natural at some point to try theorem proving tools. Theorem provers enable the specification of formal models and the mechanical verification of the properties of those models. Paper proofs are faster and easier to write than mechanised ones because the reader is often easier to convince than a theorem prover and for this reason they are useful as soon as there is not sufficient time for doing the mechanised proof (or one considers the effort is not worth it). However, mechanised proofs are a strong guarantee for the reader of the proof or the user of the framework: there is no more need to understand the proof to be convinced that the theorem is true, it becomes sufficient to understand the theorem and trust the theorem prover. For these reasons, several aspects of my work have been formalised in a theorem prover in order to increase the confidence the reader and the user will have in the properties of our models. I use the theorem prover Isabelle/HOL but I am convinced that all the proofs presented here would have worked if I used another theorem prover with similar capacities, like Coq for example.

Organisation of the manuscript

This document is organised as follows. I tried to alternate overviews of some works with more technical aspects in order to provide both an overall view of what we achieved and some insight on the way we formalise our programming models, applications and execution environments. The reader should however refer to the papers I cite for further details on both technical and implementation aspects. Among my papers, four of them are included in the manuscript. They are not necessarily the best ranked journal and conferences of my publication list, but I chose them because they are representative of our work in the last years.

Chapter 2 deals with the active object programming model. It focuses on three points: an overview of active object languages in Section 2.2, my contribution to some of the features of the ProActive middleware in Section 2.3, and our efforts for the mechanised formalisation of a functional ASP calculus in Section 2.4.

Chapter 3 deals with the composition of distributed applications. It mainly focuses on the GCM component model presented in Section 3.1. Our efforts in the design of non-functional features and adaptation procedures are described in Sections 3.2 and 3.3. Section 3.4 presents the formalisation of GCM in Isabelle/HOL. Then a short section (Section 3.5) presents our work on behavioural skeletons, a quite different approach for the composition of distributed or concurrent applications. Finally this chapter finishes with a long section on the behavioural specification and verification of distributed applications, mainly targeted at GCM components (Section 3.6).

Chapter 4 presents some works in progress and perspectives. It focuses on two contributions that are advanced enough to obtain first results, but for which the expected outcome is much greater than the current achievements. The two main research directions presented in this chapter are:

- Broadcast for structured peer-to-peer networks: the objective of this work is twofold. First we aim at designing an efficient broadcast protocol for CAN overlay networks. Second and most importantly, our objective is that this protocol is *proven correct*, and for this we want to formalise its principles in a theorem prover. This work is the opportunity for us to bring to the distributed system community the correctness guarantees ensured by formal methods.
- Multi-active objects: this is also a promising research direction aimed at increasing the expressive power of active objects while retaining their easy programming model. Such a model extends active objects with local multi-threading expressed in a much intuitive manner. This programming model both allows the programmer to avoid most of the deadlock that can occur in an active object program, and also increases the efficiency of active objects on multi-core architectures.

Finally, this chapter contains a few less advanced research directions. This chapter also contains a very short overall conclusion to this document.

Note on citations

In all this document, citations of papers that I co-authored are of the form [number] (e.g., [1]) whereas citations of other papers are of the form [AuthorsinitialsYear] (e.g., [AT04]).

Chapter 2

The Active object Programming Model

2.1 Introduction

The active object model is derived from the Actors model [AT04, AMST97, Agh86]. Actors and active objects share a lot of concerns and advantages. A great part of the mechanisms designed for one programming paradigm can be applied, almost straightforwardly, to the other.

The principle of active objects is very simple: An object is said to be active if it can be deployed on a remote machine. As a consequence, every call to such an object will be a remote method invocation; we call such an invocation a *request*. An active object is thus an object that treats the requests it receives, it is an object together with a thread.

To decouple the invocation object from the invoked object, contrarily to a classical remote invocation, the invoker is not blocked waiting for the result instead a future object is created and represents the result of the remote invocation.

Futures, first introduced in Multilisp [Hal85] and ABCL/f [TMY94] are used as constructs for concurrency and data flow synchronisation. Futures are language constructs that improve concurrency in a natural and transparent way. A future represents a result that has not been computed yet. When the result is available it can be retrieved (automatically or manually), we then say that the future is *resolved*. Frameworks that make use of explicit constructs for creating futures include Multilisp [Hal85], λ-calculus [NSS06], Creol [JOY06], SafeFuture API [WJH05], and ABCL/f [TMY94]. In contrast, futures are created implicitly in frameworks like ASP [10], [16], [1], AmbientTalk [DCMM06], ProActive [CDdCL06]. In ASP_{fun} [12], [40] we also chose to provide implicit futures. Usually, in those object-oriented languages, implicit creation corresponds to asynchronous method invocation. A key benefit of the implicit creation is that no distinction is made between synchronous and asynchronous operations in the program. This way, when a method invocation is local, usual method invocation is performed, whether when the accessed object is remote, a future is immediately obtained.

Additionally, the futures can be accessed explicitly or implicitly. In case of explicit access, operations like *claim* and *get*, *touch* are used to access the future [JOY06, TMY94]. For implicit access, operations that need the real value of an object (*blocking* operations) automatically trigger synchronisation with the future update operation. We say that futures are *first class* if future references can be transmitted between remote entities without requiring the future to be resolved.

2.2 Overview of languages for active objects and futures

Several programming languages or models rely on some form of active objects, we review the main ones below. Especially focusing on languages which have a formalised semantics. But before focusing on active objects, let us first review the main works on formalisation of futures in the context of functional languages. Indeed, Futures were first introduced in Multilisp [Hal85], and thus lead to a lot of work on their formalisation in the context of functional languages. We will then review main active object languages and finish by a couple of languages that are not pure active objects but also strongly relate the notions of distribution and concurrency to the notion of objects.

2.2.1 Futures in functional languages

To our knowledge, the first work on formalisation by semantic rules of Futures appeared in [FF99, FF95] and was intended at program optimisation. This work focuses on the futures of MultiLisp, that are explicitly created. The authors "compile" a program with futures into a low-level program that does explicit touch operations for resolving

the future, and then optimise the number of necessary touch operations.

In a similar vein, $\lambda(fut)$ is a concurrent lambda calculus with futures. It features non determinism primitives (cells and handles). In [NSS06], the authors define a semantics for this calculus, and two type systems. They use futures with explicit creation point in the context of λ-calculus; much in the same spirit as in Multilisp. Alice [NSSSS07] is an ML-like language that can be considered as an implementation of $\lambda(fut)$.

2.2.2 Asynchronous Sequential Processes

The ASP calculus we have defined [1] is a distributed active object calculus with futures; the ProActive library [CDdCL06] can be considered as its reference implementation. The ASP calculus formalises the following characteristics of active objects:

- *asynchronous communications*, by a request-reply mechanism;
- *futures*; in ASP, futures are transparent objects: their creation and access is implicit, futures are also *first-class*: they can replace transparently any other objects and can be communicated as result or parameter of remote method invocations;
- *sequential execution* within each process, each object is manipulated by a single thread;
- *imperative objects*, i.e. each object has a state and there is an operation for updating it.

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

Our main result consists in a confluence property and its application to the identification of a set of programs behaving deterministically. This property can be summarised as follows:

- Future updates can occur at any time without any consequence on the result of the computation,
- Execution is only characterised by the order of requests, and even more precisely by the order of request senders: To characterise uniquely an execution, it is sufficient to consider, for each activity, the ordered list of identifiers of the activities that have sent a request to this activity.
- Consequently, programs communicating over trees are deterministic.

The ASP calculus has been first designed during my thesis [49]. It leaded to several other publications, [1], [10], [16].

The impact of this formalisation on the development of the ProActive library is a strong achievement of this work. For example, thanks to this work, "automatic continuation", i.e. first class futures à la ASP have been implemented and massively used.

The code snippet shown in Figure 2.1 gives a typical example of a simple ProActive piece of code. It creates a new active object of type `A` on the JVM identified by `Node1`. All calls to that remote object will be asynchronous, and the access to the result might be subject to *wait-by-necessity*.

The main advantage of ASP is that most code can be written without caring about distribution and concurrency. Futures are automatically and transparently created upon method invocation on an active object. Synchronisation is due to wait-by-necessity that occurs upon access to a future that has not been resolved yet. This synchronisation is performed transparently, i.e. there is no construct for explicitly waiting the result of a request. Wait-by-necessity is always performed at the last moment, i.e. when a value is really needed. Futures are also transparently sent between activities. This way simple programs can be written extremely easily and rapidly.

Unfortunately, as active object are purely monothreaded, ASP's active objects can easily deadlock: most recursive *asynchronous* method calls lead to a deadlock, which is difficult to avoid when two active objects are involved in a mutually recursive invocation.

2.2.3 AmbientTalk

In AmbientTalk [DCMM06], active objects behave similarly to ASP's active objects. However, there is one major difference between the two models: in AmbientTalk the future access is a non-blocking operation, it is an asynchronous call that returns another future. There is no wait-by-necessity upon a method call on a future, instead the method call will be performed when the future becomes available, in the meantime a future represents the result of this method invocation. This differs from the approach adopted in other

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

Figure 2.1: Typical ProActive code

frameworks where access to a future is blocking. This approach avoids the possibility of a deadlock as there is no synchronisation, but programming can become tricky as there is, according to the programming model specification, no way to synchronise two processes or to know whether a computation has finished.

2.2.4 Creol

Creol [JO06, JOY06] is an active object language that executes several requests at the same time, with only one active at a given time. This is some form of *collaborative multi-threading* based on an *await* operation that releases the active thread so that another request can continue its execution. Typically, one would do an *await* when accessing a future so that if the future is not yet available another thread can continue its execution. In Creol [JO06] future creation and access is explicit, in particular a specific syntax exists for asynchronous method invocation. Creol is a uniform active object model where each object is an active one able to receive remote method invocations. Creol also ensures the absence of data races, even if request execution can be interleaved, and the result of computation is less predictable than in ASP.

De Boer et al. [BCJ07] provided the semantics of an object-oriented language based on Creol; it features active objects, asynchronous method calls, and futures. This semantics extends Creol in the sense that it supports first-class futures, although the future access is still explicit (using *get* and *await*). In the same paper, the authors also provide a proof system for proving properties related to concurrency.

The Creol model has the advantage of having less deadlocks than ASP, because in ASP a request must be finished before addressing the next one. Indeed, when the result of a request is necessary in order to finish another one, the Creol programmer can release the service thread, which is impossible in ASP. While no data race condition is possible, interleaving of the different request services triggered by the different release points makes the behaviour more difficult to predict (in particular the determinism properties of ASP cannot be proven in Creol).

Overall, explicit future access, explicit release points, and explicit asynchronous calls make the Creol programming model richer than ASP, but also more difficult to program. Finding a good compromise between expressiveness and safe program execution is a crucial aspect in the design of programming languages; we will provide in Section 4.2 another extension of the active object model providing a different tradeoff between expressiveness, efficiency, and ease of programming.

2.2.5 JCoBox

JCoBox [SPH10] is an active object programming model implemented in a language based on Java. It integrates asynchronous and synchronous communications as different operators, and partitions the object space into "coboxes", corresponding to ASP's activities. Each cobox is responsible for controlling the local concurrency and maintaining its invariants. Similarly to Creol, in each cobox a single thread is active at a time, but this thread can be released and coboxes support collaborative multi-threading.

In Creol [JO06] all objects are active, whereas ASP and JCoBox are non-uniform active object models: some objects are active and are invoked by asynchronous remote requests, other objects are passive and are accessible by a single activity (or cobox), they are transmitted by value. References from a cobox to an active object of another cobox are called "far references". Far references can only be used to perform asynchronous calls (`reference!method()`), which return futures. Futures are explicitly created and explicitly accessed, just as in Creol. `await` performs a cooperative wait on the future, whereas `get` blocks until the value of the future is received. In JCoBox, contrary to ASP, a cobox may contain multiple active objects.

Cooperative multi-threading is similar and leads to the same advantages and drawbacks as in Creol.

Figure 2.2 shows explicit future creation, and explicit future accesses in JCoBox. When inside an active object a single thread is active at a time, accessing futures can lead to deadlock in case of re-entrant requests. The solution proposed by ASP and ProActive is "first-class futures": since futures are implicitly created and transparently transmitted as method parameters and results, the

deadlock only occurs if the future is really needed. Alternatively, Creol and JCoBox provide explicit futures and allow the active thread to suspend itself until a result is returned. Consider the method `foo` of Figure 2.2, if one replaces the `await` statement by a `get`, the active object would deadlock waiting for `bar` to be executed. It might seem a safe programming guideline to systematically perform an `await` instead of a `get`. However, this might lead to unexpected non-determinism or unexpected results. In JCoBox, ready threads (i.e. threads that can execute but are suspended) are dequeued in a FIFO order. In the example, the `bar` and the second `foo` request will then be executed in an unpredictable order. Depending on when the second `foo` is received, the final result returned by the first `foo` may be '2'. This happens because 'x' can be modified by the second `foo` request before the return statement of the first.

For the moment, no programming model for distribution and concurrency that would be easy to program in all conditions, and each programming model has its own drawbacks. The great advantage of Creol and JCobox is that these programming models prevent race-conditions. They allow interleaving inside an active object so that some deadlocks can be avoided, but this interleaving is difficult to control. The concurrency model is powerful enough and easier to handle than basic threads and locking mechanisms. It is more complicated than ASP, but also more powerful. We will provide one alternative programming model featuring multi-threading for active object in Section 4.2) featuring another kind of tradeoff between expressiveness, deadlock prevention, race-condition prevention, and ease of programming.

2.2.6 JAC

JAC [HL06] is an extension of Java that introduces a higher level of concurrency and separates thread synchronisation from application logic in a declarative fashion. JAC relies on a custom pre-compiler and declarative annotations, in the form of Javadoc comments placed before method headers or inside the code. Objects are annotated as `controlled` when their threads are managed and synchronised according to JAC's annotations. JAC relies on compatibility annotations stating whether two methods can be executed at the same time; two methods should be compatible if they do not access the same variables (or if the access to those variables has been protected by some locking mechanism). For example, the following code states that `isEmpty` can be safely executed concurrently with `lookup` and with itself:

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

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

  int bar() {
    return 0;
  }

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

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

Figure 2.2: Thread interleaving in JCoBox may lead to unexpected outcomes

This way methods that can safely be run concurrently will automatically be. Additional annotations are given for finer grain or easier control of concurrency and synchronisation (e.g. to wait for a guard to be verified before executing a method). Exhaustive case study of annotation effect, in particular in relation with inheritance is also described in [HL06].

JAC's *async* annotation provides some form of active object behaviour: an asynchronous method is executed independently of others in a separate thread. The main difference with classical active objects is that classical active objects act as a unit of concurrency: they are manipulated with a single thread and enforce the absence of shared memory between active objects. Stating that all methods of a class are asynchronous and mutually exclusive would create some form of active objects, but without the absence of shared memory enforcement. For example, in ASP and JCoBox, non-active objects are called passive and are deeply copied when passed between activities in order to guarantee the absence of sharing.

We think JAC is a well designed model for declaring simply powerful concurrency rules, but unfortunately it is not

particularly adapted to a distributed environment. Classical active objects on the contrary provide a better encapsulation of data and concurrency but do not provide concurrency abstractions as powerful as JAC annotations. We thus think ASP's programming model is simpler to program, has stronger properties and is more adapted to distribution. We will see in Section 4.2 that we designed also a multi-active object programming model with annotations similar but simpler than the ones featured by JAC.

2.2.7 X10

X10 [CGS+05] is a programming language that adopts a fairly new model, called partitioned global address space (PGAS). In this model, computations are performed in multiple places (possibly on various computational units) simultaneously. Data in one place is accessible remotely, and is not movable once created. Computations inside places are locally synchronous, but inter-place activities are asynchronous. This decouples places and ensures global parallelism. While this model seems fundamentally different from active objects, both can be used to express the same kind of applications, but more importantly in X10 like in ProActive, a special care has been put to offer to the programmer a wide set of so called *technical services*. Technical services are non-functional features that are crucial to deploy and run large-scale applications, they typically include fault-tolerance, security, code migration, deployment on a wide set of architectures, support for several communication protocols, etc. Next section will focus on some of the technical services featured by ProActive (the ones in which formal background play a major role). Also note that X10 places can host multiple activities, resulting in a similar service to what multi-active objects offer (see Section 4.2).

2.3 A middleware for active objects

Most of my work on the active object model contributes to the formalisation of this model, but also to the design of additional mechanisms specific to this programming model. Indeed, from the original ASP calculus we designed during my PhD thesis, several works have been derived. The works presented in this section concern technical services, i.e. functionalities that are not directly part of the programming model but play a major role in the support for programming and running real applications. Even if technical services are not part of the programming model, they can be tightly coupled to it as we will show below. All the practical developments made in the OASIS team were done in the context of the ProActive Middleware. ProActive is a Java middleware implementing the active object paradigm. I consider it as a very useful implementation platform: the theoretical formalisms that we developed are applicable in a more general context than the ProActive middleware, but ProActive gave us the opportunity to show that our algorithms and ideas were working in a real large-scale runtime environment.

With members of the OASIS team, we designed a couple of crucial mechanisms specialised to active objects, those mechanisms are of course inspired by protocols and algorithms existing in the literature, but the reader is referred to the paper describing each of the mechanism for a comparison with related works.

2.3.1 Garbage collecting active objects

In [31] we designed a garbage collection mechanism for Active objects. We say that an active object is *idle* if it does not serve any request (and has no request to serve in its request queue). More exactly, an active object can be considered *useless*, and thus can be garbage collected, if it is not serving any request and cannot serve any request in the future, i.e. if it is only accessible by active objects that are idle. To garbage collect useless active objects, we build the reference graph between activities without modifying the local garbage collector. We identify cycles of idle activities as cyclic garbage instead of the more common unreachable strongly connected component. For this the idle activities reach a consensus stating that they are useless: consider a connected set of activities, if all of them are idle and agree on this point, then they can be garbage collected because they will always be idle and will never receive any request to serve. This Garbage collector has been implemented in ProActive by Guillaume Chazarain.

2.3.2 Fault-tolerance for active objects

The results proved formally on the ASP calculus can be of particular interest when designing protocols for recovering from faults [EAWJ02]. Indeed, the minimal characterisation of the execution provided by the properties of ASP allows the optimisation of the events that have to be stored to replay the original execution: *to enforce a second execution to occur similarly to a first one, it is sufficient to remember, for each active object, the ordered list of active objects that have sent a request to this active object.* This is particularly useful for designing a CIC (communication induced checkpointing) protocol when checkpoints cannot be taken at any time. Typically, ProActive is a Java middleware, and in Java it is impossible to stop a thread and

store its current status; thus the only moment when a checkpoint can be taken is between two requests. Indeed, at this moment the applicative state of the object is in general sufficient to restore its execution: execution can be restored by starting serving the next request.

In a few words, in order to recover after a fault, each process stores from time to time its state (when programming with active objects, the active objects abstracts away the processes). Such a saved process state is a *checkpoint*. The idea in CIC protocols [BCS84] is to force the checkpoints to happen at some precise point (relatively to communications), if checkpoints are not conveniently placed, they cannot be used at recovery because the recovery line would be inconsistent. In our protocol, we deal with constraints on the moment when a checkpoint can be taken. When a checkpoint should be taken for consistency reason (forced checkpoint), we delay it and place it as soon as possible. Between the time the checkpoint should be forced, and the time the checkpoint is really taken we remember the history of events on all the processes. In our case, this is realistic because this history is minimal (it is restrained to the list of request senders for each request queue). This way we managed to keep a low number of checkpoints while ensuring coherence of the execution upon recovery.

CIC protocols require all the processes to recover upon a failure. They are convenient for ensuring at low-cost the fault-tolerance of relatively small systems, but when systems get bigger message logging protocols are preferred even if they induce a bigger overhead on non-faulty executions, because only the failed processes have to restart. We also proposed a mixed protocol relying on groups of machines, each group is handled by our CIC protocol, but inter-group fault-tolerance relies on message logging. This way only the group that contains a failure restarts, and a good balance between overhead and recovery time can be found.

To summarise, we used ASP results on confluence, and a formalisation specific to fault-tolerance for active objects [29] to design and formalise a CIC protocol for active objects. We then designed a mixed protocol between the previous one and message logging mechanism to better adapt to large-scale infrastructures, like Grids. This work has been realised during the PhD thesis of Christian Delbé [Del07], and all those protocols have been implemented in the ProActive Middleware and heavily tested in a large-scale environment [22], [7].

Determinism and characterisation of execution

One of the interesting point of this work is that it relates determinism and execution characterisation. On this particular aspect it bridges the gap between two research areas: language theory and distributed systems. Recent work published in the domain of distributed systems [LS11] focuses on the characterisation of deterministic program execution from the distributed systems point of view. Unfortunately, the work of Lu and Scott failed to link the notion of determinism in execution characterisation with the one used in programming languages theory. Somehow our work on fault tolerance gives a first idea on how to link those two worlds: determinism property proved on the calculus could be used to minimally characterise an execution. Somehow both [LS11] and our work are just early preliminary results that could be reused to provide general results unifying the notion of deterministic languages and deterministic distributed execution. A first step in this direction would be to use the semantics of the concurrent programming language to provide a formal and minimal characterisation of nondeterministic events logged by rollback-recovery protocols [EAWJ02].

Also, to study a few security aspects related to the active object programming model, during the thesis of Felipe Luna del Aguila, we proposed a security extension to the ASP calculus [18].

2.4 Functional active objects and their mechanised formalisation

Reasoning on complex calculi is a difficult task, especially when the language encodes several complementary constructs. Indeed, in ASP the coexistence of active objects, futures, and local objects induce several semantic rules, and reasoning on them, while not particularly difficult can be error prone because of the numerous cases to consider. This is also the case for most of the calculi presented in Section 2.2, indeed for those calculi that have a semantics formally defined (e.g. Creol, JCoBox), the semantics is defined by a quite large number of cases for handling each of the runtime construct of the language. This complexity comes most probably from the coexistence of concurrency, distribution, and object-related concerns. In practice paper proofs are most often valid but still contain numerous small mistakes. Thus even if most paper proofs are quite convincing, a mechanical proof verified by a theorem prover is necessarily more reliable. Indeed, as the prover verifies the steps of the proof, to be convinced by a mechanical proof, it is sufficient to check the formalisation of the hypothesis and of the conclusion and to run the theorem prover to check all the steps, instead of verifying them by hand. This great progress in the confidence that can be put on proofs comes at a price: mechanical formalisation is much more costly in

time than paper formalisation. Considering the difficulty to check proofs and how important it is to have a strong confidence in the generic proofs ensuring correct behaviour of a language or a middleware, I think that in quite a lot of cases it is worth spending the additional time required for the mechanical formalisation.

Considering the complexity of the task and the time it should take, we first formalised a simpler version of the ASP calculus, a functional ASP called ASP_{fun} in the Isabelle/HOL theorem prover. This work also allowed us to prove new results on the functional variant of the calculus, to show the calculus is confluent, and to study typing.

This contribution follows several other works encoding in a theorem prover calculi and languages. As promoted by the POPLmark challenge [ABF+08], we are close to a point where calculi and languages can be reasonably formalised inside theorem provers. The closest work to ours is probably [CLM07] that formalises in Coq a sequential version of the imperative ς-calculus. On the concurrent side, several works focus on the π-calculus [RH03, BP07] but no work studied the coexistence of objects and distribution. That is why we consider the formalisation in Isabelle/HOL of ASP_{fun} as a valuable contribution in the domain of mechanised formalisation of languages and calculi.

Up to now, the mechanised formalisation of ASP_{fun} gave us both the opportunity to study a functional active object calculus and its properties, and to experiment with binder techniques in this context. The rest of this section summaries our main contributions related to ASP_{fun}. Then a publication further detailing those points is included.

2.4.1 A functional dead-lock free active object calculus

The first contribution of this work is to provide and formalise a new distributed active object calculus. Our formalisation encodes:

- A functional active object calculus with futures.
- A type system for active objects.
- The proof that typed objects never dead-lock; more precisely, we proved both subject-reduction and progress for well-typed ASP_{fun} terms.

We think this formalisation will allow further investigations on futures, typing, and active objects paradigms.

ASP_{fun} calculus is an extension of ς-calculus with only the minimal concepts for defining active objects and futures. At the root of this extension is the notion of active object, which is an object to which is attached a request queue and a thread for treating the requests. We call the set consisting of an active object, its request queue, its service thread, and all the non-active objects referenced by the active object, an *activity*. Syntactically, the extension only requires a single new primitive: *Active*. This primitive encodes the active object creation, when invoked, a new active object is spawned and will be able then to treat (remote) invocations sent to it. An active object encodes quite well the notion of distribution as each active object is well-separated from the other active objects: each active object is independent in terms of data access, and of synchronisation: communications and synchronisation with an activity is limited to request sending and reply sending. In ASP_{fun}, field update can also be performed on an active object; in fact it should be viewed as an activity creation primitive. ASP_{fun} is distributed in the same sense as ASP: it enables parallel evaluation of activities while being oblivious about the concrete locations in which the execution of these activities takes place. The actual deployment is not part of the programming language, it is the task of an application deployer rather than of the application programmer.

The absence of side-effects and the guarantee of progress make the program easy to reason about and easy to parallelise. Compared to previous works on ASP, ASP_{fun} has the following main originalities:

- The functional nature of the calculus gives us the opportunity to give a semantics to the update of an active object's field: here a new activity is created with the updated object, this semantics is coherent with the semantics of update in the functional ς-calculus.
- Also due to the functional nature of ASP_{fun} it is safe to reply partially evaluated futures, making the semantics more general. A partial reply is here a request partially evaluated, i.e. an ASP term that is not completely reduced and can continue its evaluation on the caller side.
- We studied typing and proved properties on well-typed terms.
- We also proved several properties on the calculus, like the fact that there is no way to create cycle of activities or futures in ASP_{fun}.

The paper included in this section details the semantics of ASP_{fun}, and the properties we proved. All the Isabelle files for the ASP_{fun} formalisation can be downloaded at `gforge.inria.fr/scm/viewvc.php/ASPfun/?root=tods-isabelle` or at `www-sop.inria.fr/oasis/Ludovic.Henrio/misc.html`.

2.4.2 Binder techniques

One crucial question raised by the formalisation of calculi and semantics is the representation of binders and variables.

Indeed, binder representation plays a major role in the formalisation of calculi as shown in the current solutions proposed to the POPLmark challenge for example. Binders play a major role in the formalisation of calculi because the notion of α-conversion (i.e. renaming of bound variables) makes the representation of terms difficult: one wants at the same time to represent terms in a way that is concise and easy to manipulate, but also to equate terms that only differ by the name chosen for the bound variables. Intuitively, a language that has local scopes and parameters – for example functions $\lambda x.fx$ – needs to refer to the formal parameters – here x – when they occur inside these scopes. The natural, human understandable way is to use variables, like x, to define and denote formal parameters by name, but variables are not well suited for mechanisations. Variable capture may occur: a free variable x in a term t may accidentally be "captured" when substituting t inside a scope where x is bound. To avoid this, we use a consistent renaming, α-equivalence (renaming of bound variables). However, α-equivalence creates equivalence classes making equality and proofs of theorems harder to handle. Several ways of representing binders and variables have been designed.

De Bruijn indices

The solution proposed by N. G. de Bruijn, is to replace each occurrence of a variable by an integer equal to the number of binders that have to be crossed to reach the binder for the considered variable: a variable is replaced by the distance from its binding scope. For example, the λ-term $\lambda x.x(\lambda y.x\,y)$ becomes $\lambda(0(\lambda 1\,0))$. Unfortunately, substitution becomes technical because of the "lifting" of indices when entering a binder, or replacing a term inside binders.

Locally nameless

The de Bruijn method can be refined in order to avoid manipulation of explicit indices. For this, the principle of locally nameless representation is to use indices to represent bound variables, and classical named variables to represent free (unbound) variables. Open and close operations translate between those representations [ACP$^+$08]. This technique is attractive as it combines unique representation, with human understandable expression of specification.

The *open* operation, written t^u, substitutes a term u for the outermost bound variable, in the term t. For example

$$(bvar0\,\lambda((bvar1)(bvar0)))^n = (n\,\lambda(n\,(bvar0)))$$

The opposite operation *closes* a term: given a name, the closing replaces the occurrence of variables of this name with an index for a variable bound at the outermost level, for example:

$$^{\backslash n}\,(n\,\lambda(n\,(bvar0))) = (bvar0\,\lambda((bvar1)(bvar0)))$$

In the locally nameless approach we must use only well-formed terms, where *bound* variables are represented by indices. The notion of locally closed terms ensures this e.g. $\lambda(bvar\ 2)$ is not locally closed. Local closure of terms is a necessary requirement for most theorems. Another problem arises when reducing a term under a binder. Useful properties should be valid when closing a term under *any fresh variable*. We need: $\forall x \in FV(t).t^x \to (t')^x \implies \lambda(t) \to \lambda(t')$. The drawback of this proposition is that it is sensitive to the set of free variables, that may vary in an unexpected way. In other words, the property written above is too strong and cannot be verified because $FV(t)$ varies during reduction. The approach of *cofinite quantification* [ACP$^+$08] should be used: we abstract over the set of free variables $FV(t)$, and let fresh variables range over the complementary of an *existentially quantified finite* set L:$\exists L$ finite.$\forall x \notin L \ldots$. This set can then be instantiated appropriately, when handling proofs.

Nominal techniques

Another approach, proposed by Urban based on Pitts' work on nominal logic [Pit03], is called nominal technique [Uea06]. Here, terms are identified as a set bijective to all terms factorised by α-equivalence. The classical hypothesis, "there is a fresh variable" for a term t is replaced by "there is a finite *support* for x": the set of atoms used in t is finite, and infinitely many "fresh" atoms are available. Unfortunately, we cannot use the Isabelle/HOL package for nominal techniques as it is, because our terms contain finite maps, and while it is trivial that finite maps guarantee finite support, such a reasoning is not yet supported by the Isabelle/HOL package for nominal techniques.

Higher order abstract syntax

In HOAS binders are directly represented by binders of the meta-level [RH03]. The encoding is more direct than in the other approaches, but HOAS is restricted when it comes to meta-level reasoning [HMS01].

Binder techniques and ASP$_{\mathbf{fun}}$

As shown in the paper included in this section we experimented both with de Bruijn indices and with locally nameless notations. A precise comparison can be found in the

included paper. To summarise, locally nameless allowed us to reason at a higher level of abstraction, with more precise distinction between bound and free variables, and relieved us from the burden of having to prove a lot of technical small lemmas involving indexes. Unfortunately, additional lemmas are also required on the locally nameless side, involving reasoning on cofinite quantification, on the link between bound and free variables, fresh variables, and renaming. Those new lemmas bring more precision to the theory, but the gain in time for the mechanised formalisation is questionable, especially in our case, where the formalisation of binders is not at all a major concern of the calculus. However, as stated earlier the objective of the mechanised formalisation is to my mind a much greater confidence in the properties proved, and locally nameless techniques induce a more natural notation for writing the theorems and the semantics of the calculus; consequently, locally nameless increase the readability of the proved properties and thus the confidence an external reader will have of those properties. This is to my mind the main reason why switching from de Bruijn notations to locally nameless techniques enriched our formalisation.

2.4.3 Paper from Science of Computer Programming, Jan 2011

This paper details our works on ASP_{fun}, it defines the calculus, its semantics, its typing. It also presents our formalisation in Isabelle/HOL including two different binder representations, and their comparison in the context of mechanised formalisation.

Science of Computer Programming 77 (2012) 823–847

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

ASP_{fun}: A typed functional active object calculus

Ludovic Henrio [a], Florian Kammüller [b,c,*], Bianca Lutz [c]

[a] CNRS – I3S – Univ. Nice Sophia-Antipolis – INRIA, Sophia-Antipolis, France

[b] Middlesex University, London, UK

[c] Technische Universität Berlin, Germany

ARTICLE INFO

Article history:
Received 10 November 2009
Received in revised form 31 October 2010
Accepted 28 December 2010
Available online 12 January 2011

Keywords:
Theorem proving
Object calculus
Futures
Distribution
Typing
Binders

ABSTRACT

This paper provides a sound foundation for autonomous objects communicating by remote method invocations and futures. As a distributed extension of ς-calculus we define ASP_{fun}, a calculus of functional objects, behaving autonomously and communicating by a request-reply mechanism: requests are method calls handled asynchronously and futures represent awaited results for requests. This results in an object language enabling a concise representation of a set of active objects interacting by asynchronous method invocations. This paper first presents the ASP_{fun} calculus and its semantics. Then, we provide a type system for ASP_{fun} which guarantees the "progress" property. Most importantly, ASP_{fun} has been formalised; its properties have been formalised and proved using the Isabelle theorem prover and we consider this as an important step in the formalization of distributed languages. This work was also an opportunity to study different binder representations and experiment with two of them in the Isabelle/HOL theorem prover.

1. Introduction

This paper presents a functional active object language featuring asynchronous method calls and futures; it has been formalised in the Isabelle/HOL theorem prover. ASP_{fun} (asynchronous sequential processes) is an extension of the ς-calculus [1] where objects are distributed into several activities, and activities are the units of distribution. Communications toward activities are asynchronous (remote) method calls; and futures are identifiers for the result of such asynchronous invocations. A future represents an evaluation-in-progress in a remote activity. Futures can be transmitted between activities as any object: several activities may refer to the same future. The calculus is said to be functional because method update is realised on a copy of the object: there is no side-effect. The paper also studies a type system for active objects. Typing is a well-studied technique [2]; we prove here a classical typing property, progress, in unusual settings, distributed active objects.

We mechanically proved properties about ASP_{fun} and, since the calculus is abstract, our semantics and mechanisation can be a basis for the analysis of related languages. Distributed active objects represent an abstract notion of concurrently computing and communicating activities. Clearly, finding a combination of objects and concurrency is not new as a notion — related notions are summarized in the following paragraph — but providing a fully formalized and mechanized calculus including typing for this combination is. Mechanical proofs, though more difficult to perform, are more reliable because they should contain no errors. This article shows that theorem proving techniques can handle distributed features of programming languages. Our work is an important step toward the mechanisation of calculi for distributed computing. The calculus is a model for distributed frameworks relying on active objects or on actors as explained below.

* Corresponding author at: Middlesex University, London, UK.
E-mail addresses: Ludovic.Henrio@inria.fr (L. Henrio), f.kammueller@mdx.ac.uk, flokam@cs.tu-berlin.de (F. Kammüller), sowilo@cs.tu-berlin.de (B. Lutz).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.12.008

Object and distribution: the active object model

The underlying principle for distribution considered in this paper originates from Actors [3,4]. Our calculus provides a model of computations that are distributed in the same way as the actor or the active object paradigm. In these paradigms, distributed computation relies on absence of sharing between processes allowing them to be placed on different machines. Those models feature asynchronous RMI-like communications. We detail below some characteristic distributed languages adhering to these principles.

Principles of actors are the following. Each actor is an independent functional process, i.e., an object together with its own thread. Actors interact by asynchronous message passing. They receive messages in their inbox and process them asynchronously. Instead of having an internal state, actors can change their behaviour, i.e., their reaction to received messages. Actors are some form of active objects. Our approach is to take distribution and parallelism notions similar to actors but fit them into a calculus of classical objects. This article introduces a formalisation, both on paper and in a theorem prover, of actor paradigms in the context of ς-calculus.

From the original actor paradigm [5,6,4], several languages have been designed. Some languages directly feature actors, distributed active objects (like the ProActive [7] library), or other derived paradigms. The calculus ASP_{fun} provides a simple model for such languages.

The ASP calculus [8,9] provides understanding and proofs of confluence for asynchronous distributed systems; it is a formalisation of the active object model. In ASP, active objects communicate in an actor-like manner. Additionally, ASP uses *future* objects, i.e., objects for which the real value is being calculated. Syntactically, the ASP calculus is an extension of the **imp**ς-calculus [1,10] with two primitives (*Serve* and *Active*) to deal with distributed objects.

An active object is similar to an actor in the sense that it has a request queue (corresponding to the actor's mailbox), it does not share memory with other active objects, and active objects communicate by messages. For active objects, communications take the form of a remote method invocation that will be treated asynchronously. We call *activity* the set consisting of an active object, its request queue, the set of normal (also called *passive*) objects known by the active objects, and the set of results the active object has computed. Each active object has a *single* thread; only this thread is allowed to access the active object and the passive ones.

Proactive [7] is a Java middleware for distributed computing. It is based on the notion of active objects and is considered as an implementation of the ASP calculus. It is particularly designed for large scale distributed computations (clusters, Grids, or cloud computing). Deployment is based on the notion of virtual nodes and deployment descriptors: when an activity is created, it is associated with a virtual node, and a deployment descriptor file associates virtual nodes to real machines. As active objects do not share memory they provide a good abstraction of location. Finally, an active object is uniquely associated to a location and an application thread (even if several active objects can be placed on the same machine in practise). Active objects act as the unit of both concurrency and distribution. In ProActive, the programmer only cares about splitting its computation into independent active objects that will run in parallel; then the localisation aspect is delegated to a different role: the deployer. It is a key feature of the programming language and the middleware to guarantee that the program behaves the same whatever physical locations are chosen to deploy the active objects.

Also, the Creol [11] language features futures with (multi)-active objects; distribution principles in Creol are quite similar to ASP_{fun} except that Creol is an imperative language with a more complex semantics. Johnsen et al. [11] also advocate the active object paradigm as a model of distributed computation: "The Creol model targets distributed objects by a looser coupling of method calls and synchronization." The mechanised formalisation of an active object language is a major contribution of this paper. Such a formalisation will increase the confidence in the properties of this programming model and our understanding of distributed computation.

Contribution

We define in this paper ASP_{fun}, a calculus of functional active objects with futures. It formalises the notion of active objects presented in the previous paragraph. For example, the behaviour of ProActive active objects follows quite faithfully the semantics of ASP_{fun}, and thus properties proved here can be transferred to this context. Compared to imperative ASP, ASP_{fun} investigates the typing of active objects and ensures progress properties in a functional context.

The language, its type system, and all properties have been completely formalised (http://gforge.inria.fr/scm/viewvc.php/ASPfun/?root=tods-isabelle) and proved in Isabelle/HOL [12]. This formalisation is approximatively 14 000 lines, only 10% dealing with the language definition, and the rest dealing with the proof of ASP_{fun} properties. We also believe that the formalisation of a calculus like ASP_{fun} in a theorem prover will be helpful in the future design of distributed languages and can provide a reliable basis for proofs using paradigms such as distributed objects, futures, remote method invocations, actors, or active objects. Our main contributions are:

- A functional active object calculus with futures and its properties. We illustrate the expressiveness of the calculus on a couple of examples.
- A type system for active object languages.
- An investigation on how to provide a type-safe calculus featuring active objects and futures, where typing ensures progress.

Table 1
ASP_{fun} syntax.

$s, t ::= \underline{x}$	Variable
$\mid \underline{[l_j = \varsigma(x_j, y_j)t_j]^{j\in 1\dots n}}$	$(\forall j,\ x_j \neq y_j)$ object definition
$\mid \underline{s.l_i(t)}$	$(i \in 1\dots n)$ method call
$\mid \underline{s.l_i := \varsigma(x, y)t}$	$(i \in 1\dots n, x \neq y)$ update
$\mid \underline{Active(s)}$	Active object creation
$\mid \alpha$	Active object reference
$\mid f_i$	Future reference

- A formalisation of those features in a theorem prover, that will allow further investigations on futures, typing, and active objects paradigms.
- A comparison of different techniques for representing binders together with two implementations of our framework using two different techniques.

ASP_{fun} is the first calculus to our knowledge to feature those characteristics, however each of those characteristics exists in some distributed programming language, and sometimes in other calculi. In this context, the main contribution of this paper is the formalisation of these features as a single calculus, but mainly the mechanised formalisation of this calculus in a theorem prover. This paper will provide a complete description of our formalisations, an analysis of the technical decisions that we have taken to represent distributed objects in Isabelle/HOL, and an overall conclusion on the techniques we used and the tools we provide.

This article is organised as follows. Section 2 presents ASP_{fun} and its semantics. Two examples illustrate the calculus in Section 3. Section 4 gives first properties of the calculus focusing on well-formed configurations and on the impossibility to create cyclic dependencies. Section 5 provides a type system for ASP_{fun} ensuring both subject-reduction and progress. Some details on the formalisation in Isabelle/HOL and on the major proofs are given in Section 6; this section particularly details binder representation. Section 7 discusses alternative semantics we could have chosen. Finally, Section 8 details our position relatively to existing distributed languages and calculi and Section 9 concludes by a summary of our achievements and a discussion of the properties of ASP_{fun} as presented in this paper.

2. Syntax and semantics

This section presents the ASP_{fun} calculus. We first define its syntax and explain its principles. Then, we give a small-step operational semantics for the calculus.

2.1. Syntax

We use three sets of identifiers: the labels of ς-calculus methods (l_i), the activities ($\alpha, \beta, \dots$), and the futures (f_i). Like in ς-calculus in ASP_{fun} every term is an object either given by its definition or returned by a term evaluation. The syntax of ASP_{fun} includes *object definition*, *method invocation*, and *method override* inherited from ς-calculus. An object consists of a set of labelled methods. A method is a function with two formal parameters: one represents *self*, i.e., the object in which the method is contained, the other, which is new in ASP_{fun}, is an actual parameter given at invocation time. Object fields are considered as degenerate methods not using the parameters. A method call is addressed to an object and receives an object as parameter. A method update acts on an object providing a new value for one method possibly defining it. ς-calculus terms are identified modulo *renaming of bound variables*.

One of the basic principles of ASP_{fun} is to perform a minimal extension of the syntax of ς-calculus. ASP_{fun} programs only use one additional primitive, *Active*, for creating an active object. The syntax of ASP_{fun} is shown in Table 1; the static syntax (the programs) consists of only underlined constructs; future and active object references are created at runtime.

While the syntactic extension of ς-calculus is minimal, the semantics, that we will define in the following, is (almost) entirely new. For example, in Table 2, only the two first rules are an adaptation of ς-calculus' semantics; all the others are specific to ASP_{fun}.

2.2. Informal semantics of ASP_{fun}

The semantics of the local object calculus is similar to the one of Abadi and Cardelli [1]. A method invocation reduces to the method body where formal parameters are replaced by actual ones: $[l = \varsigma(x, y)a].l(b)$ reduces to a where x is replaced by $[l = \varsigma(x, y)a]$ and y is replaced by b. A method update returns a new object replacing the original method by the one on the right side of ':='. We focus now on the distributed features of ASP_{fun}.

A *configuration* is a set of activities. Each activity possesses a single active object, which is a ς-calculus term. Activating an object, $Active(s)$, means creating a new activity with the object s to be activated becoming an *active object*. It is immutable. The activity is the unit of distribution. A request sent to an activity is an invocation to the active object; it is processed by

Table 2
ASP_{fun} semantics.

CALL

$$\frac{l_i \in \{l_j\}^{j\in 1\ldots n}}{E\left[[l_j = \varsigma(x_j, y_j)s_j]^{j\in 1\ldots n}.l_i(t)\right] \to_\varsigma E\left[s_i\{x_i \leftarrow [l_j = \varsigma(x_j, y_j)s_j]^{j\in 1\ldots n}, y_i \leftarrow t\}\right]}$$

UPDATE

$$\frac{l_i \in \{l_j\}^{j\in 1\ldots n}}{E\left[[l_j = \varsigma(x_j, y_j)s_j]^{j\in 1\ldots n}.l_i := \varsigma(x, y)t\right] \to_\varsigma E\left[[l_i = \varsigma(x, y)t, l_j = \varsigma(x_j, y_j)s_j^{j\in(1\ldots n)-\{i\}}]\right]}$$

LOCAL

$$\frac{s \to_\varsigma s'}{\alpha[f_i \mapsto s :: Q, t] :: C \to_\| \alpha[f_i \mapsto s' :: Q, t] :: C}$$

ACTIVE

$$\frac{\gamma \notin (\mathrm{dom}(C) \cup \{\alpha\}) \qquad noFV(s)}{\alpha[f_i \mapsto E[Active(s)] :: Q, t] :: C \to_\| \alpha[f_i \mapsto E[\gamma] :: Q, t] :: \gamma[\varnothing, s] :: C}$$

REQUEST

$$\frac{f_k \text{ fresh} \qquad noFV(s) \qquad \alpha \neq \beta}{\alpha\,[f_i \mapsto E[\beta.l(s)] :: Q, t] :: \beta[R, t'] :: C \to_\| \alpha\,[f_i \mapsto E[f_k] :: Q, t] :: \beta\left[f_k \mapsto t'.l(s) :: R, t'\right] :: C}$$

SELF-REQUEST

$$\frac{f_k \text{ fresh} \qquad noFV(s)}{\alpha\,[f_i \mapsto E[\alpha.l(s)] :: Q, t] :: C \to_\| \alpha\,[f_k \mapsto t.l(s) :: f_i \mapsto E[f_k] :: Q, t] :: C}$$

REPLY

$$\frac{\beta[f_k \mapsto s :: R, t'] \in \alpha[f_i \mapsto E[f_k] :: Q, t] :: C}{\alpha[f_i \mapsto E[f_k] :: Q, t] :: C \to_\| \alpha[f_i \mapsto E[s] :: Q, t] :: C}$$

UPDATE-AO

$$\frac{\gamma \notin \mathrm{dom}(C) \cup \{\alpha\} \qquad noFV(\varsigma(x, y)s) \qquad \beta[R, t'] \in \alpha[f_i \mapsto E[\beta.l := \varsigma(x, y)s] :: Q, t] :: C}{\alpha[f_i \mapsto E[\beta.l := \varsigma(x, y)s] :: Q, t] :: C \to_\| \alpha[f_i \mapsto E[\gamma] :: Q, t] :: \gamma[\varnothing, t'.l := \varsigma(x, y)s] :: C}$$

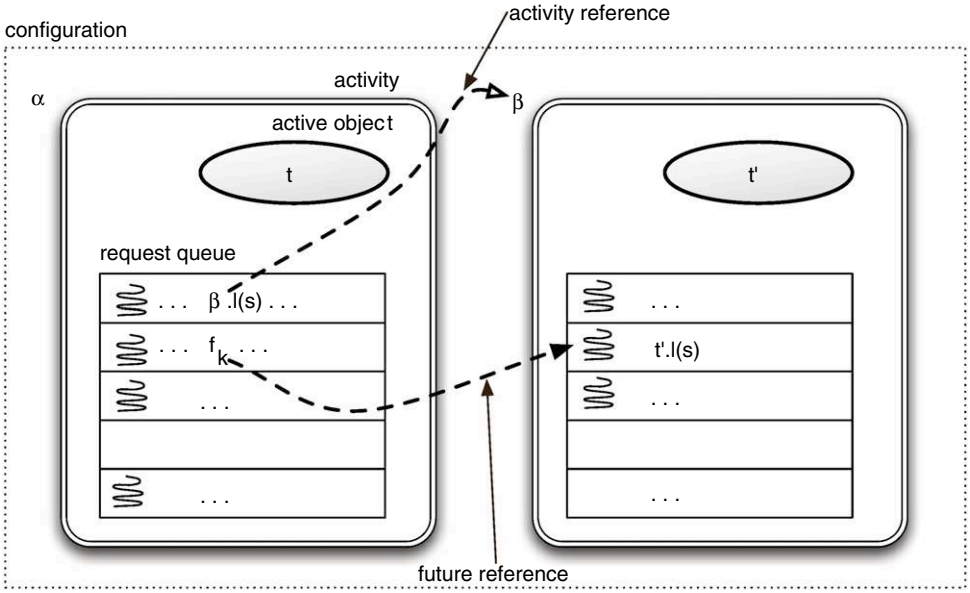


Fig. 1. Example configuration in ASP_{fun} with two activities.

the activity. The set of requests processed by an activity is called *request queue* by similarity with the active object model but, here, as the calculus is functional, requests can be treated in an unordered fashion. Indeed, as we do not have any side effects, the order of execution of request has no influence on the result.

Fig. 1 illustrates the basic concepts of ASP_{fun}. It shows a configuration consisting of two activities. In each activity an ellipse represents the active object, and each rectangle is a request (i.e., maps a future identifier to a term being evaluated). In ASP_{fun}, all the requests can be evaluated in parallel.

Every message sent toward an activity is a method call to the activate object. Such a *remote method invocation* (also called *request*) is asynchronous: the effect of this method call is — both — to create a new request in the request queue

of the destination and to replace the original method invocation by a reference to the result of the created request. A reference to a (promised) result is called a *future*. In ASP_{fun}, futures are entities that can be passed to other activities, e.g., as arguments or results of requests; several activities may use the same future. Trying to access the result referenced by a future (e.g., invoking a method on it) is not possible until the future has been received. The current term of any request (even partially evaluated) can be returned at any moment: the current term for the request replaces the corresponding future. This operation is called a *reply*. We chose to allow replies with a partially evaluated term because it fits well with the functional nature of the calculus; but we will see in Section 7 that a more classical semantics returning only requests entirely evaluated also guarantees progress. Future values must be stored forever because future references can spread over the activities and, without a mechanism for counting the future references, it is impossible to know if a future reference still exists in the system. A garbage collection mechanism for future would detect whether a future is still referenced; garbage collection of futures is not studied in this paper.

Fig. 1 can also be considered as an illustration of a method call: the configuration consisting of the first line of the request queues is transformed into the configuration consisting of the second lines of the request queues, the reference to the activity β is lost, and the reference to the future f_k is created together with a request computing f_k in β.

Reduction can occur in any request of any activity. The only restriction is that an object cannot be sent to another activity (e.g., as a request parameter) if this object has free variables, otherwise such variables would escape their scope and the moved object would be meaningless. To better understand this restriction, suppose one tries to evaluate the sub-term $\varsigma(x, y)remoteObject.send(y)$, which is the body of a method. Sending y first would be meaningless as this variable is bound by $\varsigma(x, y)$ and it would mean nothing in the remote object. We force to perform evaluation steps until the sent terms have no more free variables; in the example we would wait until the method is invoked with a parameter. Fortunately, the type system ensures that a term typed in an empty environment has no free variable, which is sufficient to guarantee that remote method invocations can be performed at some point of the reduction.

It is difficult to give a natural semantics to the update of an active object. Indeed the usual field update that directly modifies the value of an object field would create an additional way of communicating with an active object by changing its status without performing a method invocation. The functional nature of the calculus (updating an object creates a copy) oriented us toward the following solution: a method update on an active object creates a new activity with the method updated.

Proving confluence for ASP_{fun} would lead to numerous technical difficulties and is out of the scope of this paper. Informally, depending on the execution, the set of created activities and the number of requests may vary, but the result of the computation is always the same. For example, depending on the order of execution an activity creation may precede or succeed a term duplication thus creating one or two activities. But if two activities are created, they are equivalent, and, as no side effect exists in ASP_{fun}, the two activities will always behave the same. A similar reasoning can be applied to the possibly duplicated requests. This explains why the calculus is confluent in the sense that it always produces equivalent results.

As a tiny example of the semantics $Active([l = \varsigma(x, y)[]]).l([])$ first creates an activity with the object $[l = \varsigma(x, y)[]]$, then performs a remote invocation on the method l of this activity (which creates a future), and finally replies replacing the future by the result of the invocation, []. More formally, assuming $Active([l = \varsigma(x, y)[]]).l([])$ is being evaluating inside activity α for calculating the value for a future f_0 (notations will be detailed in the next section):

$$\begin{aligned}\alpha\,[(f_0 \mapsto Active([l = \varsigma(x, y)[]]).l([])), \ldots] &\to_{\|} \alpha\,[(f_0 \mapsto \beta.l([]))\ldots] \;\|\; \beta\,[\varnothing, [l = \varsigma(x, y)[]]] \\ &\to_{\|} \alpha\,[(f_0 \mapsto f_1)\ldots] \;\|\; \beta\,[(f_1 \mapsto []), [l = \varsigma(x, y)[]]] \\ &\to_{\|} \alpha\,[(f_0 \mapsto [])\ldots] \;\|\; \beta\,[(f_1 \mapsto []), [l = \varsigma(x, y)[]]]\end{aligned}$$

We consider this work as a reliable basis for further studies on stateless objects, giving a semantics for autonomous services, which in case they are stateless can be implemented such that they never dead-lock (i.e., they always progress). ASP_{fun} can also represent component-like distributed systems interacting by invocation of services: an active object exposes its methods to the external world and holds references to required external services provided by other active object.

2.3. Small-step operational semantics

The semantics of ASP_{fun} necessitates the definition of some structures that are used for the dynamic reduction. First, we define a configuration C as an unordered set of activities: a configuration is a mapping from activity identifiers to activities. Each activity is composed of a request queue (mapping from future identifiers to terms) and an active object (term). Configurations are identified modulo reordering of activities and of requests inside an activity.

$$C ::= \alpha_i[(f_j \mapsto s_j)^{j\in I_i},\ t_i]^{i\in 1\ldots p} \quad \text{where } \{I_i\} \text{ are disjoint subsets of } \mathbb{N}$$

As futures are referenced from anywhere, two requests must correspond to two different futures; uniqueness is ensured in this paper by indexing futures over disjoint families. We use the term *local semantics* to refer to the semantics expressing the execution local to each *activity*, where an activity is the unit of distribution. Abadi and Cardelli [1] present various ς-calculi that only consider objects and their manipulation as primitive; local semantics of ASP_{fun} (two first rules of Table 2) is just an adaptation of this work. More precisely, local semantics of ASP_{fun} extends ς-calculus with a second parameter for methods.

Classically we define contexts as expressions with a single hole ($\bullet$). $E[s]$ denotes the term obtained by replacing the single hole by s.

$$E ::= \bullet \mid [l_i = \varsigma(x,y)E, l_j = \varsigma(x_j,y_j)t_j^{j\in(1\ldots n)-\{i\}}] \mid E.l_i(t) \mid s.l_i(E) \mid E.l_i := \varsigma(x,y)s \mid s.l_i := \varsigma(x,y)E \mid Active(E)$$

For a better integration with the distributed calculus, we choose a small-step semantics ($\rightarrow_\varsigma$) for the ς-calculus. It is composed of the two first rules of Table 2; one invokes a method (using the invoked object as first parameter), the other updates a method, i.e., it creates a new object where one method is replaced by a new one.

To simplify the reduction rules, we let $Q, R ::= (f_{ij} \mapsto s_{ij})^{j\in 1\ldots n_p}$ range over request queues and identify mappings modulo reordering: $\alpha[f_i \mapsto s_i :: Q, b] :: C$ is a configuration containing the activity α which contains a request $f_i \mapsto s_i$, where C is the remainder of the configuration that cannot contain an activity α. Now, $\alpha[Q, s] \in C$ means: α is an activity of C with request queue Q and active object s: $\alpha[Q, s] \in C \Leftrightarrow \exists C'. C = \alpha[Q, s] :: C'$. Similarly, $(f_i \mapsto s) \in Q$ stands for: a request of Q associates s to the future f_i. The empty mapping is $\varnothing$; the domain of a mapping is dom; e.g., $\mathrm{dom}(C)$ is the set of activities defined by C. Predicate $noFV(s)$ is true if s has no free variables (the only binder being ς this definition is classical). The parallel reduction $\rightarrow_\parallel$ on configurations is defined in Table 2.

Classically, the substitution $s\{x \leftarrow t\}$ is capture avoiding (renaming is performed to avoid free variables in t to be captured by binders in s), whereas the replacement of $\bullet$ by a term in a context is not.

- LOCAL performs a local reduction inside an activity: one step of the reduction $\rightarrow_\varsigma$ is performed on one request.
- ACTIVE creates an activity; the term s passed as argument to the Active primitive becomes the active object. The newly created activity receives a fresh activity identifier γ. Initially, the new activity has an empty request queue, and γ replaces the activation instruction $Active(s)$ thus allowing future invocations to this activity.
- REQUEST sends a request from the activity α to the activity β with $\alpha \neq \beta$. A new request is created at the destination invoking the method l on the active object (t'); a fresh future f_k is associated to this request, and replaces the invocation on the sender side. Freshness is defined classically: f_k is fresh in C if $\forall \alpha[Q, t] \in C$, $f_k \notin \mathrm{dom}(Q)$.
- SELF-REQUEST is the REQUEST rule when the destination is the sender, $\alpha = \beta$. The semantics of this rule is similar to the preceding one but, as the request queue is modified on both the sender's and the receiver's side, it would be difficult to express a single simple rule for the two cases.
- REPLY updates a future: it picks the request calculating a value for the future f_k and sends the current value of this request (s) to an activity that refers to the future. The request may be only partially evaluated meaning a reply to a request is enabled as soon as the method invocation is performed. Returning partial replies can have the effect to duplicate computation and will be further discussed in Section 7. Necessarily, $noFV(s)$ holds because, as an active object and a transmitted parameter have no free variables, a request value never has free variables. This time, the structure of the rule avoids introducing a separate rule for $\alpha = \beta$.
- UPDATE-AO updates a method of an activity $\beta[R, t']$. It creates a new activity whose active object performs a (local) update on t': $t'.l := \varsigma(x,y)s$. It requires that the new method definition for l has no free variable.

The requirement $noFV(s)$ for the communicated terms is necessary. Indeed, communicating a term with free variables would cause variables to escape the scope of their binder as explained in Section 2.2. In Section 7, we will discuss the choices that have been made in the $\mathrm{ASP}_{\mathrm{fun}}$ semantics and the alternative possibilities.

2.4. Basic ς-calculus datatypes

For reasons of completeness of this paper, we introduce here the definitions of standard datatypes in the ς-calculus [1] that are used in this paper. They give a good illustration of encoding of basic datatypes in ς-calculus.

Booleans and conditional

$$\mathrm{true} = [if = \varsigma(x,y)x.then(y), then = \varsigma(x,y)[], else = \varsigma(x,y)[]]$$
$$\mathrm{false} = [if = \varsigma(x,y)x.else(y), then = \varsigma(x,y)[], else = \varsigma(x,y)[]]$$
$$\text{if } b \text{ then } c \text{ else } d = ((b.then := \varsigma(x,y)c).else := \varsigma(x,y)d).if([])$$

In the third line above, $x, y \notin FV(c) \cup FV(d)$; [] denotes the empty object. The definition shows how — similar to λ-calculus — the functionality of the constructor is encoded in the elements of the datatype: when b is true its method *if* delegates to the method *then*, filled with term c, when false, *if* delegates to *else*, executing term d.

Lists

$$c :: l = [hd = \varsigma(x,y)c, tl = \varsigma(x,y)l, mty = \mathrm{false}]$$
$$hd\ l = l.hd$$
$$tl\ l = l.tl$$
$$\langle\rangle_{\mathrm{list}} = [hd = \varsigma(x,y)[], tl = \varsigma(x,y)[], mty = \mathrm{true}]$$
$$l = \langle\rangle_{\mathrm{list}} = l.mty$$

In the first line above, $x, y \notin FV(c) \cup FV(l)$; [] denotes again the empty object. Lists are encoded as accumulation of first elements in the head field *hd*; the predicate judging emptiness of a list is an abbreviation for the third field *mty* that always tracks whether a list is empty or not.

3. Examples

This section illustrates the ASP_{fun} calculus with two examples, one focusing on futures, and the other showing a few less conventional features of the calculus.

3.1. A broker

The following example illustrates some of the advantages of futures for the implementation of services. The three activities hotel α, broker β, and customer γ are composed by $\|$ into a configuration (to improve readability, $\|$ separates the different activities in the examples). Here, the customer γ wants to make a hotel reservation in hotel α. He uses a broker β for this service by calling a method *book* provided in the active object of the broker. We omit the actual search of the broker β in his database and instead hardwire the solution to always contact some hotel α. That is, the method *book* is implemented as a call $\varsigma(x, date)\alpha.room(date)$ to a function *room* in the hotel α. Also the internal administration of hotel α is omitted; its method *room* just returns a constant bookingreference. Initially, only the future list of the customer γ contains a request for a booking to broker β; the future lists of α and β are empty. The following steps of the semantic reduction relation $\rightarrow_{\|}$ illustrate how iterated application of reduction rules evaluates the program.

$$\begin{array}{l}\gamma[f_0 \mapsto \beta.book(date), t] \\ \| \ \beta[\varnothing, [book = \varsigma(x, date)\alpha.room(date), \ldots]] \\ \| \ \alpha[\varnothing, [room = \varsigma(x,date)\text{bookingreference}, \ldots]]\end{array}$$

The following step of the semantic reduction relation $\rightarrow_{\|}^{*}$ creates the new future f_1 in β by rule REQUEST, this call is reduced according to LOCAL, and the original call in the client γ is replaced by f_1.

$$\begin{array}{l}\gamma[f_0 \mapsto f_1, t] \\ \| \ \beta[f_1 \mapsto \alpha.room(date), \ldots] \\ \| \ \alpha[\varnothing, [room = \varsigma(x,date)\text{bookingreference}, \ldots]]\end{array}$$

The parameter x representing the *self* is not used but the call to α's method *room* with parameter *date* creates again by rule REQUEST a new future in the request queue of the hotel activity α that is immediately reduced due to LOCAL to bookingreference.

$$\begin{array}{l}\gamma[f_0 \mapsto f_1, t]] \\ \| \ \beta[f_1 \mapsto f_2, \ldots] \\ \| \ \alpha[f_2 \mapsto \text{bookingreference}, \ldots]\end{array}$$

Finally, the result bookingreference is returned to the client by two REPLY-steps: first the future f_2 is returned from the broker to the client γ and then this client receives the bookingreference via f_2 directly from the hotel α.

$$\begin{array}{l}\gamma[f_0 \mapsto \text{bookingreference}, t] \\ \| \ \beta[f_1 \mapsto f_2, \ldots] \\ \| \ \alpha[f_2 \mapsto \text{bookingreference}, \ldots]\end{array}$$

The example is intentionally simplified to focus on the flow of control given by the requests, replies, and the passing on of the futures: the booking reference can flow directly to the customer γ possibly without passing by the broker β. This shows that futures allow the implementation of efficient communication flows. The example further illustrates how futures can be employed to provide some confidentiality. The broker β does not need to give away his data base of hotel references: he can instead return just a reference to the result of his negotiations; the booking reference.

3.2. A service provider

We illustrate how ASP_{fun} can be used to implement a (generic) service detailing on the control structure and the service administration while abstracting the actual service content. Eventually, we use the informal description "some function on *client_data*" to denote the final function representing the service. What we are interested in is the global service architecture. We want to show how the active object update — generating a new object on update — can be employed efficiently to support creation and delegation of service objects. The service scenario uses three active objects: a client, a server, and a service.

A client object can be any object having some *data* that is passed to the service by a request. Furthermore, each client can be started by supplying a corresponding server *s*. The method *start* generates a service request with the client's *data* on its request queue. The server object is defined below. Note that the method invocation $s.serve(x.data)$ accepts as parameter $x.data$ due to our extension of the parameter-less ς-calculus.[1]

$$\text{client} \equiv [data = \text{"some data"}, start = \varsigma(x, s)(s.serve(x.data)), \ldots]$$

[1] In the ς-calculus the parameter has to be simulated by updating a separate field in the object.

On the other side, the service is an object for which a new instance will be generated for the client's use. Such a new instance is created by server objects below by updating the field *client_data* of a service object which automatically creates a new active object representing the service for the client.

$$\text{service} \equiv [\mathit{client\_data} = \text{“some data”}, \mathit{actual\_service} = \text{“some function on } \mathit{client\_data}\text{”}, \ldots]$$

A server object generates an individual service personalised by the client's data by instantiating an active object representing the basic service. Initially, the field *base_service* contains the empty object but during initialisation this will be updated.

$$\text{server} \equiv [\mathit{base\_service} = [], \mathit{serve} = \varsigma(x, d)\ (x.\text{base\_service}).\mathit{client\_data} := d]$$

3.2.1. Initialisation

We first describe how the service is initialised. The ASP_{fun} program initialising the system is a base object that has a method *init*. The initial configuration will be defined in Section 4.3. It contains a single activity with a unique request. In our case, this request is the activation of the init object and the corresponding call $\mathit{Active}([\mathit{init} = \ldots]).\mathit{init}$.

Now, *init* needs to start clients that know this server. We can use the following ASP_{fun} object to start one client,

$$\mathit{Active}(\text{client}).\mathit{start}(\mathit{Active}(\text{server}.\mathit{base\_service} := \mathit{Active}(\text{service})))$$

where “client”, “server”, and “service” are the abbreviations given before. For several clients being started in the *init* method, we need some iterator construct. We define a *map* function for a method name f as follows. This function applies the method f on each object of a list of objects l while using s as a second parameter to all these calls. It returns a list of objects (which is itself an object). The operator :: is the list constructor, $\langle\rangle_{\text{list}}$ the empty list, *hd* and *tl* give first element and rest of a list, and $l = \langle\rangle_{\text{list}}$ is the empty-list predicate. We, furthermore, use the let and if-then-else construct presented in Section 2.4.

$$\begin{aligned} \mathit{map}_f \ = \ & \varsigma(x, (s, l))\ \text{if } l = \langle\rangle_{\text{list}} \text{ then } \langle\rangle_{\text{list}} \\ & \qquad \text{else } (\mathit{hd}\ l).f(s) :: (x.\mathit{map}_f(s, \mathit{tl}\ l))\ \text{end} \end{aligned}$$

Now, we use the following list of $n + 1$ occurrences of client activations,

$$\Lambda = \langle \mathit{Active}(\text{client}.\mathit{data} := d_0), \ldots, \mathit{Active}(\text{client}.\mathit{data} := d_n)\rangle$$

where the d_i denote the different data items of the clients. Summarising, the definition of the user program is as follows.

$$\begin{aligned} & \mathit{Active}([\ \mathit{init} \ = \ \varsigma(x, y) \\ & \qquad\qquad \text{let } \Lambda = \ldots \\ & \qquad\qquad\quad S = \mathit{Active}(\text{server}.\mathit{base\_service} := \mathit{Active}(\text{service})) \\ & \qquad\qquad \text{in } x.\mathit{map}_{\mathit{start}}((S, \Lambda)), \\ & \mathit{map}_{\mathit{start}} = \ldots\ ]).\mathit{init} \end{aligned}$$

This user program sent as the only request in the initial activity α sets the server into action.

3.2.2. Server in action

In this section, we show how the server works. Let us first show the configuration after initialisation; the *Active* commands have all been evaluated; the evaluation of the activation list Λ has created client instances γ_i, $i \in \{0, \ldots, n\}$; the evaluation of S in *init* has

- created a service object σ by evaluating $\mathit{Active}(\text{service})$,
- created a server object Σ by evaluation of $\mathit{Active}(\text{server}.\mathit{base\_service} := \sigma)$,
- sent *start* requests to all client objects γ_i by evaluating the $\mathit{map}_{\mathit{start}}$ invocation putting $\Sigma.\mathit{serve}(\gamma_i.\mathit{data})$ on their request queues.

Note that we choose to evaluate first the innermost service activation, then S itself, before we pass it to $\mathit{map}_{\mathit{start}}$. This leads to the particular service architecture we have in mind; a different order creates several servers ultimately producing the same results (see remark on confluence in Section 2.2) but it would be less economical. The obtained configuration additionally contains an activity ι for the initialiser object with a single served request (*init*) and another α for the initial configuration.

$$\begin{aligned} \Big[\ \ & \gamma_0[f_0 \mapsto \Sigma.\mathit{serve}(\gamma_0.\mathit{data}), [\ldots]], \\ & \ldots \\ & \gamma_n[f_n \mapsto \Sigma.\mathit{serve}(\gamma_n.\mathit{data}), [\ldots]], \\ & \Sigma[\varnothing, [\mathit{base\_service} = \sigma, \\ & \qquad \mathit{serve} = \varsigma(x, d)\ (x.\text{base\_service}).\mathit{client\_data} := d]], \\ & \sigma[\varnothing, \text{service}], \\ & \iota[f' \mapsto [\langle f_0, \ldots, f_n\rangle], [\mathit{init} = \ldots, \mathit{map}_{\mathit{start}} = \ldots]], \\ & \alpha[f \mapsto f', []] \qquad \Big] \end{aligned}$$

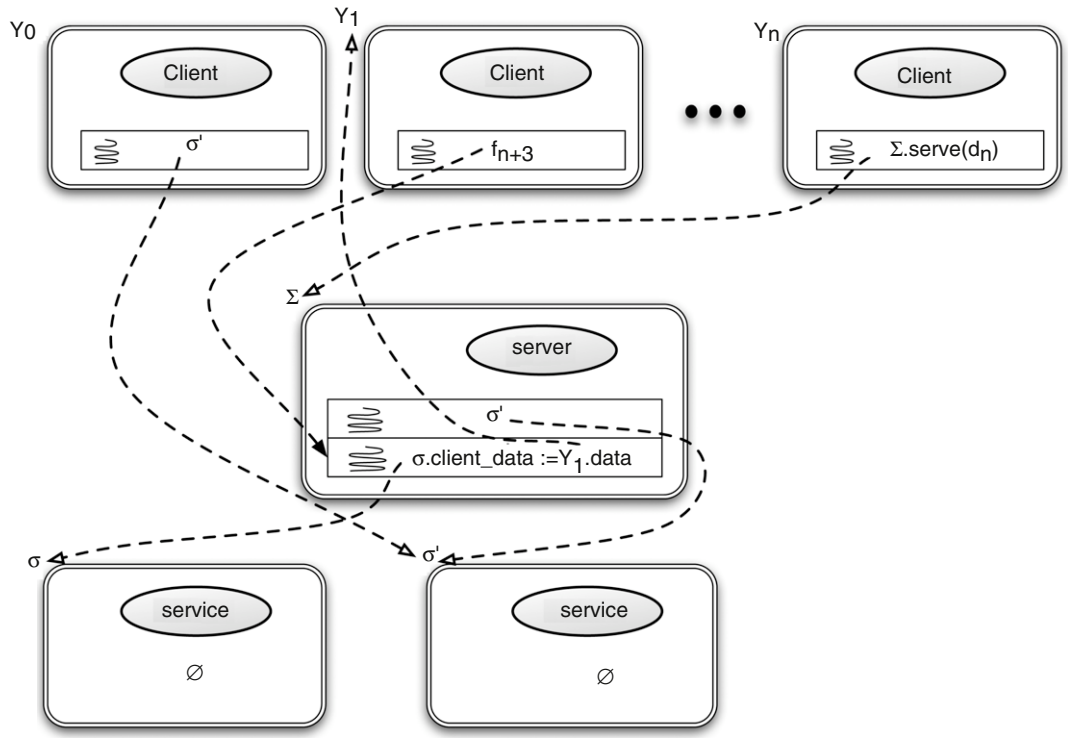


Fig. 2. Server in action.

Evaluating the request of the first client γ_0, the above configuration reduces to one in which the server Σ holds one request for creating a service for γ_0.

$$\begin{bmatrix} \gamma_0[f_0 \mapsto f_{n+1}, [\ldots]], \\ \ldots \\ \Sigma[f_{n+1} \mapsto (\sigma.\mathit{client\_data}) := \gamma_0.\mathit{data}, [\ldots]], \\ \sigma[\varnothing, \mathsf{service}], \ldots \end{bmatrix}$$

Next, evaluation of future f_{n+1} creates a new service object σ' for this service call with the first client's data $\gamma_0.\mathit{data}$ injected as *client_data* in σ' while Σ's request queue now holds the activity reference σ' in future f_{n+1}.

$$\begin{bmatrix} \gamma_0[f_0 \mapsto f_{n+1}, [\ldots]], \\ \ldots \\ \Sigma[f_{n+1} \mapsto \sigma', [\ldots], \\ \ldots \\ \sigma'[\varnothing, [\mathit{client\_data} = \gamma_0.\mathit{data}, \\ \quad \mathit{actual\_service} = \text{“some function on } \mathit{client\_data}\text{”}]], \ldots \end{bmatrix}$$

The rule REPLY returns the activity reference σ' as a result to the client γ_0 by future f_{n+1}.

$$\begin{bmatrix} \gamma_0[f_0 \mapsto \sigma', [\ldots]], \\ \ldots \\ \Sigma[f_{n+1} \mapsto \sigma', [\ldots], \\ \ldots \\ \sigma'[\varnothing, [\mathit{client\_data} = \gamma_0.\mathit{data}, \\ \quad \mathit{actual\_service} = \text{“some function on } \mathit{client\_data}\text{”}]], \ldots \end{bmatrix}$$

Now, the client γ_0 has access to its service σ' in a personal instantiation. The client may call at leisure the services of σ'—using this reference to his "personalised" service. The following calls of the other clients $\gamma_2, \ldots, \gamma_n$ all have a similar effect. For each of them a new instance of the first service object σ is automatically created by the semantics of update. All clients finally receive an activity reference and all have been served by the same server Σ. The server in action is illustrated in Fig. 2 depicting the moment just when the second client's service call is launched to the server.

As a further extension to this example, we could consider programming a central registry for the service objects. To this end, we just change the *init* method of the base object to make a final update to a local field registry to store the result of

Table 3
The *nocycle* property.

$$\frac{\alpha[Q, E[\beta]] \in C}{\alpha \; knows_C \; \beta} \qquad \frac{\alpha[f_i \mapsto E[\beta] :: Q, t] \in C}{f_i \; knows_C \; \beta} \qquad \frac{\alpha[Q, E[f_k]] \in C}{\alpha \; knows_C \; f_k} \qquad \frac{\alpha[f_i \mapsto E[f_k] :: Q, t] \in C}{f_i \; knows_C \; f_k}$$

$$nocycle(c) \Leftrightarrow \exists r.\, r \; knows_C^+ \; r$$

the activation map to all clients.

$$\begin{aligned}
&[init \;=\; \varsigma(x, y)\\
&\qquad \text{let } \Lambda = \ldots\\
&\qquad\quad S = Active(\mathsf{server}.base_service := Active(\mathsf{service}))\\
&\qquad \text{in } x.registry := (x.map_{start}(S, \Lambda)),\\
&registry = \langle\rangle_{\text{list}},\\
&map_{start} = \ldots]
\end{aligned}$$

With this changed base object, the call to *init* has exactly the same effect as before. Only as a final step, is the base object updated to keep the results list of all the created client objects (and thereby also of the services).

4. Properties of ASP_{fun}

This section presents two major properties of ASP_{fun}: the semantics is well-formed; and reduction does not create cycles of futures and activity references.

4.1. Well-formed configuration

To show correctness of the semantics, we define a *well-formed configuration* as referencing only existing activities and futures; then we prove that reduction preserves well-formedness.

Definition 4.1 (*Well-Formed Configuration*)**.** A configuration C is well-formed, denoted $wf(C)$, if and only if for all α, f_i, s, Q, and t each of the following holds:

$$\alpha[Q, E[\beta]] \in C \;\vee\; \alpha[f_i \mapsto E[\beta] :: Q, t] \in C \;\Rightarrow\; \beta \in \mathrm{dom}(C)$$
$$\alpha[Q, E[f_k]] \in C \;\vee\; \alpha[f_i \mapsto E[f_k] :: Q, t] \in C \;\Rightarrow\; \exists \gamma,\, R,\, t'.\, \gamma[R, t'] \in C \wedge f_k \in \mathrm{dom}(R)$$

We have shown that, starting from a well-formed configuration, the reduction shown in Table 2 always reaches a well-formed configuration.

Property 1 (*Reduction Preserves Well-Formedness*)**.**

$$(s \rightarrow_{\parallel} t \;\wedge\; wf(s)) \;\Rightarrow\; wf(t)$$

This can be considered as a correctness property for the semantics of ASP_{fun}: no ill-formed configuration can be created by the reduction.

4.2. Absence of cycles

Informally, ASP_{fun} avoids blocking method invocations because a not fully evaluated future can be returned to the caller at any time. The natural question arises whether there is the possibility for live-locks: a cycle of communications (here, a cycle of replies in fact) in which no real progress is made apart from the actual exchange of communication. However, we can show that, given a configuration with no cycle, any possible configuration that may be derived from there has no cycle either. The cycles we consider are formed of activity references and futures.

We say that an activity or a future knows another one if it holds a reference to it. An activity holds a reference if it has this reference inside its active object. A future holds a reference if the request computing this future contains this reference. Table 3 shows the rules defining the $knows_C$ relationship for a configuration C together with the *nocycle* property where $knows_C^+$ is the transitive closure of $knows_C$ ($r \; knows_C^+ \; r' \Leftrightarrow r \; knows_C \; r' \vee \exists r''.(r \; knows_C \; r' \wedge r'' \; knows_C^+ \; r')$). It is necessary to interleave references to futures and activities in the definition of $knows_C$ because, for example, a reference from an active object becomes a reference from a future when a REQUEST rule is evaluated.

We proved that the reduction defined in Table 2 maintains the absence of cycles for a well-formed configuration.

Theorem 1. *Reduction does not create cycles:*

$$nocycle(C) \;\wedge\; wf(C) \;\wedge\; C \rightarrow_{\parallel} C' \;\Rightarrow\; nocycle(C')$$

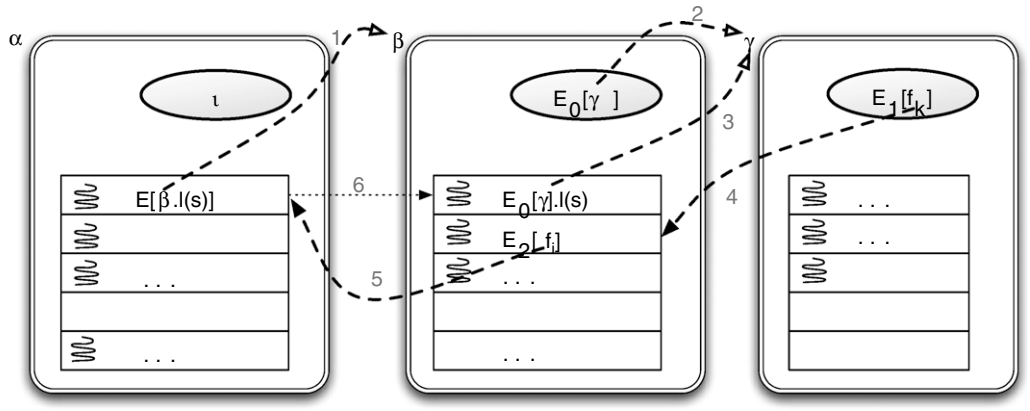


Fig. 3. A cycle of future and activity references.

The theorem relies on the fact that domains of request queues are disjoint, which is enforced by the definition of a configuration in ASP_{fun}. Absence of cycles ensures that there are no live-locks related to the distributed aspects of ASP_{fun}, i.e., no infinite cycle of replies. Live-locks that can exist in ASP_{fun} are inherited from ς-calculus: they are either infinite loops inside a ς-calculus term or infinite sequences of method calls (distributed or not).

Fig. 3 shows cycles of futures and activity references. We have two cycles, one consisting of the arrows numbered {1, 2, 4, 5}, another one is formed by the arrows {3, 4, 5, 6}.

Absence of cycle limits the expressiveness of the language (no cross-references), but this restriction is inherited from the functional nature of the language. Indeed, functional languages have no references, whereas active objects and futures create some kind of references; preventing cycles and modification is necessary to keep the functional nature of ASP_{fun}.

4.3. Initial configuration

This section shows how a reasonable initial configuration can be built from a program. In a usual programming language, a programmer does not write configurations but usual programs invoking some distribution or concurrency primitives (in ASP_{fun} *Active* is the only such primitive). This is reflected by the ASP_{fun} syntax given in Section 2.1. A "program" is a term s_0 given by this static syntax (it has no future or active object reference and no free variable). In order to be evaluated, this program must be placed in an initial configuration. The initial configuration has a single activity with a single request consisting of the user program:

$$\mathit{initConf}(s_0) = \alpha[f_0 \mapsto s_0, []]$$

This configuration is well-formed, and the activity α will never be accessible. Consequently, any reachable configuration is well-formed. We also see that the initial configuration has no cycles, and Theorem 1 ensures that any reachable configuration has no cycles.

Property 2. *Any configuration reachable from an initial configuration is well-formed and has no cycles ($\rightarrow_{\parallel}^{*}$ is the reflexive transitive closure of $\rightarrow_{\parallel}$).*

$$\mathit{initConf}(s_0) \rightarrow_{\parallel}^{*} C \;\Rightarrow\; \mathit{wf}(C) \wedge \mathit{nocycle}(C)$$

5. Typing active objects

This section provides a type system for ASP_{fun}. Starting from a ς-calculus basic type system, we first define typing for the *Active* primitive; then we define type-checking rules for an ASP_{fun} configuration. After the classical subject-reduction property, we show that the type system ensures *type uniqueness*, *well-formedness* of configurations, and more importantly *progress*. We will see that typing ensures that no method can be invoked on a term that is unable to handle it; the semantics ensures that no invocation or update on a future or an activity can be indefinitely blocked.

5.1. A local type system

We first adapt the simple type system that Abadi and Cardelli [1] devised as $\mathbf{Ob_1}$. Object types are of the form $[l_i : B_i \rightarrow D_i]^{i \in 1 \ldots n}$. The syntax of ASP_{fun} is extended by adding type information on both variables under the binder ($\varsigma(x, y)$ becomes $\varsigma(x{:}A, y{:}B)$). As highlighted in [1], adding type information on the binders ensures type uniqueness.

Table 4 defines the typing of local ASP_{fun} terms as presented in 2.1. It is an adaptation of the typing of $\mathbf{Ob_1}$ in [1]. A, B, and D range over types. The variable T represents a *type environment* containing type assumptions for variables and is identified

Table 4
Typing the local calculus.

$$\textsc{Val}\ x \quad x{:}A :: T \vdash x{:}A$$

$$\textsc{Type Object}\quad \frac{A = [l_i : B_i{\to}D_i]^{i\in 1\ldots n} \qquad \forall i \in 1\ldots n,\ x_i{:}A :: y_i{:}B_i :: T \vdash t_i : D_i}{T \vdash [l_i = \varsigma(x_i : A, y_i : B_i)t_i]^{i\in 1\ldots n} : A}$$

$$\textsc{Type Call}\quad \frac{T \vdash s : [l_i : B_i{\to}D_i]^{\,i\in 1\ldots n} \quad j \in 1\ldots n \qquad T \vdash t : B_j}{T \vdash s.l_j(t) : D_j}$$

$$\textsc{Type Update}\quad \frac{A = [l_i : B_i{\to}D_i]^{i\in 1\ldots n} \qquad T \vdash s : A \quad j \in 1\ldots n \qquad x{:}A :: y{:}B :: T \vdash t : D_j}{T \vdash s.l_j := \varsigma(x : A, y : B)t : A}$$

Table 5
Typing configurations.

$$\textsc{Type Active}\quad \frac{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash a : A}{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash \mathit{Active}(a) : A}$$

$$\textsc{Type Activity Reference}\quad \frac{\beta \in \mathrm{dom}(\Gamma_{act})}{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash \beta : \Gamma_{act}(\beta)}$$

$$\textsc{Type Future Reference}\quad \frac{f_k \in \mathrm{dom}(\Gamma_{fut})}{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash f_k : \Gamma_{fut}(f_k)}$$

$$\textsc{Type Configuration}\quad \frac{\mathrm{dom}(\Gamma_{act}) = \mathrm{dom}(C) \qquad \mathrm{dom}(\Gamma_{fut}) = \bigcup\{\mathrm{dom}(Q) \mid \exists\, \alpha, a.\ \alpha[Q, a] \in C\} \qquad \forall\, \alpha[Q, a] \in C.\ \begin{cases} \langle \Gamma_{act}, \Gamma_{fut}\rangle, \varnothing \vdash a : \Gamma_{act}(\alpha) & \wedge \\ \forall f_i \in \mathrm{dom}(Q).\ \langle \Gamma_{act}, \Gamma_{fut}\rangle, \varnothing \vdash Q(f_i) : \Gamma_{fut}(f_i) \end{cases}}{\vdash C : \langle \Gamma_{act}, \Gamma_{fut}\rangle}$$

modulo reordering. Its extension by a new assumption stating that the variable x has type A is denoted by $x{:}A :: T$. We now authorise :: to update a mapping entry: $(x : A) :: T$ associates the type A to x even if an entry for x existed in T. The first rule of Table 4 accesses the type environment. Type Object describes how an object's type is checked from its constituents: an object of type $[l_i : B_i{\to}D_i]^{i\in i\ldots n}$ is formed from bodies t_i of types B_i using self parameter x_i of type A and additional parameter y_i of type B_i. When a method l_j is invoked on an object s of type $[l_i : B_i{\to}D_i]^{i\in i\ldots n}$ the result $s.l_j(b)$ has type D_j provided s has type B_j (Type Call). A method update requires that the updated object has the same type as self in the new method (Type Update).

In [1], additional rules ensure that the typing environment is well-formed. We simplified it here by defining environment as a mapping. Also, a rule for correct formation of object types is introduced in [1] mainly ensuring that there is no infinitely nested object type. This last assumption has been omitted here as it did not seem necessary and, indeed, the properties shown below have been mechanically proved without any additional assumptions on type formation.

5.2. A type system for ASP_{fun}

The type system for ASP_{fun} is based on an inductive typing relation on ASP_{fun} terms; it is defined in Table 5. From local typing (Table 4), in addition to types of variables, we need to refer to types for futures and activities. Thus, we add a pair of parameters $\langle \Gamma_{act}, \Gamma_{fut}\rangle$ in the assumptions of a typing statement: we write $\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash x : A$ instead of $T \vdash x : A$. These parameters consist of a mapping Γ_{act} from activities to the type of their active object and another one Γ_{fut} from future identifiers to the type of the corresponding request value. Thus, we first adorn each rule of Table 4 with those two additional parameters.

Then, we add to these rules the three first rules of Table 5 that define the local typing of ASP_{fun}. These rules allow the typing of references to activities and futures and define typing of the *Active* primitive: the type of an activated object is the type of the object.

The last rule of Table 5 incorporates into a configuration the local typing assertions. This rule states that a configuration C has the configuration type $\langle \Gamma_{act}, \Gamma_{fut}\rangle$ if the following conditions hold.

- The same activity names are defined in C and in Γ_{act};
- the same future references are defined in the activities of C and in Γ_{fut};
- for each activity of C, its active object has the type defined in Γ_{act};
- and each request has the type defined in Γ_{fut} for the corresponding future.

Similarities can be found between typing of activity or future references and reference types [2]. A closer work seems to be the typing rules for futures [13].

5.3. Basic properties of the type system

Let us start by a couple of simple properties of the typing system. First, type-uniqueness existing for $\mathbf{Ob_1}$ is also verified by our type system.

Property 3 (*Unique Type*)**.** *Each expression in ASP_{fun} has a unique type.*

$$\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash a : A \wedge \langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash a : A' \Longrightarrow A = A'$$

Well-typed configurations are well-formed. Indeed, if an activity or a future is referenced in the configuration, it must have a type and thus be defined in Γ_{act} or Γ_{fut}, and also in the configuration.

Property 4 (*Typing Ensures Well-Formedness*)**.** $\vdash C : \langle \Gamma_{act}, \Gamma_{fut}\rangle \Rightarrow wf(C)$.

5.4. Subject reduction

Subject reduction ensures that reduction preserves the typing relation. Therefore, it is often also called *preservation*. We prove subject reduction of ASP_{fun} with respect to the type system given in the previous section.

We prove first the subject reduction property for the local reduction:

Property 5 (*Local Subject Reduction*)**.**

$$\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash t : A \ \wedge \ t \rightarrow_{\varsigma} t' \Rightarrow \langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash t' : A$$

Then, we prove subject reduction for the full typing relation of configurations.

Theorem 2 (*Subject Reduction*)**.**

$$\vdash C : \langle \Gamma_{act}, \Gamma_{fut}\rangle \ \wedge \ C \rightarrow_{\parallel} C' \Rightarrow \exists\, \Gamma'_{act}, \Gamma'_{fut}.\ \vdash C' : \langle \Gamma'_{act}, \Gamma'_{fut}\rangle$$

where $\Gamma_{act} \subseteq \Gamma'_{act}$, and $\Gamma_{fut} \subseteq \Gamma'_{fut}$.

Note that activities and futures may be created by the reduction and thus the typing environment may have to be extended.

5.5. Progress and absence of dead-locks

Finally, we can prove progress for well-typed configurations. Progress states that any expression of the language is either a result or can be reduced. In ASP_{fun}, we prove progress for each request of a configuration. A term is a result, i.e., a totally evaluated term, if it is either an object (like in [1]) or an activity reference.

$$isresult(s) \Leftrightarrow \exists l_i,\ t_i,\ A.\, s = [l_i = \varsigma(x_i : A, y_i : B)t_i]^{i\in 1\ldots n} \vee \exists \alpha,\ s = \alpha$$

The type system is useful for ensuring that every accessed method exists on the invoked object. In fact, local typing ensures progress of local reduction. Typing for configurations extends the typing relation to distributed objects ensuring for example that a method invocation on a future will be possible once the result is returned. Absence of dead-locks for the distributed semantics is only ensured by the functional nature of ASP_{fun}, by the absence of loops, and by the particular semantics of the calculus. A first notion of progress can be proved: for a correctly typed configuration, either all requests are reduced to a future, or the configuration can be reduced.

Property 6. $\vdash C : \langle \Gamma_{act}, \Gamma_{fut}\rangle \ \wedge \ \alpha[f_i \mapsto s :: Q, t] \in C \Rightarrow isresult(s) \vee \exists C'.\, C \rightarrow_{\parallel} C'$.

More precisely, we can prove that the request that is not yet reduced to a result, i.e., the term s in the theorem above, can be reduced. Unfortunately, as already shown in [1], ς-calculus does not ensure that a reduced term is different from the source one, but this is an issue related to the local reduction which is not the concern of this paper. We proved that, on the distributed side, the term really always progresses and that no reduction loop is induced by the distributed features of ASP_{fun}. We can reformulate the preceding theorem:

Theorem 3 (*Progress*)**.**

$$nocycle(C) \wedge \vdash C : \langle \Gamma_{act}, \Gamma_{fut}\rangle \ \wedge \ \alpha[f_i \mapsto s :: Q, t] \in C \ \Rightarrow isresult(s) \vee \exists\, C'.\, C \rightarrow_{\parallel} C'$$

where C' can be chosen to verify: $\alpha[f_i \mapsto s' :: Q, t] \in C' \wedge (s' \neq s \vee s \rightarrow_{\varsigma} s)$.

By proving progress, we also show that ASP_{fun} is dead-lock free: as any term that is not already a result must progress, this ensures the absence of dead-lock.

As configurations reachable from the initial configurations have no cycle, a variant of the progress theorem can be stated by replacing the *nocycle* hypothesis by the reachability from a well-typed initial configuration:

Property 7 (*Progress from Initial Configuration*)**.** *Let s_0 be a* static term*; if it is correctly typed (in an empty environment), then each request of any configuration C obtained from s_0 is either reduced to a value or can be further reduced; more formally:*

$$initConf(s_0) \rightarrow_{\parallel}^{*} C \ \wedge \ \langle \varnothing, \varnothing\rangle, \varnothing \vdash s_0 : A \ \wedge \ \alpha[f_i \mapsto s :: Q, t] \in C \ \Rightarrow \ isresult(s) \ \vee \ \exists C'.\, C \rightarrow_{\parallel} C'$$

where C' can be chosen to verify: $\alpha[f_i \mapsto s' :: Q, t] \in C' \wedge (s' \neq s \vee s \rightarrow_{\varsigma} s)$.

6. Formalisation in Isabelle/HOL

The interactive theorem prover Isabelle/HOL [12] offers a classical higher order logic (HOL) as a basis for the modelling of application logics. Inductive definitions and datatype definitions can be written in a way close to programming language syntax and semantics. Semantic properties over datatypes can be expressed in a clear manner using primitive recursion which is supported by powerful proof automation using rewriting techniques. Nevertheless — unlike model checking or other fully automated proof techniques — the expressivity of HOL comes at a price: the user has to find the gist of proofs concerning his application logics himself even if simple simplification steps are handled automatically.

In this section we will give an outline of the mechanisation of ASP_{fun}, its syntax, semantics, type system, and proofs in Isabelle/HOL. To this end, we begin Section 6.1 by introducing finite maps, a useful extension of Isabelle/HOL we created for representing objects. We also discuss in some detail different techniques for representing *binders* when formalising language meta-theory—necessary for the subsequent experience report. We then describe in Section 6.2 important aspects of our proofs in a manner independent of the actual Isabelle/HOL representation. We give details on the Isabelle/HOL formalisation using de Bruijn indices in Section 6.3. For defining the operational semantics of the local object calculus, we adapted the semantics for the ς-calculus defined in [14] in order to use reduction contexts. We also proved in Isabelle/HOL that both models are equivalent or, more precisely, that both small step semantics express exactly the same reduction.

In a constant attempt to improve the Isabelle/HOL mechanisation, we have updated the ASP_{fun} mechanization with a different binder technique: we replaced the classical de Bruijn indices by a locally nameless representation that provides a more natural representation of formulae by variable names [15]. The experience of having thus performed the entire formalisation of ASP_{fun} twice enables us to provide a profound comparison of the two representation techniques in Section 6.4.

6.1. Tools for programming languages and semantics

6.1.1. Finite maps: deep versus shallow

The embedding of the language ASP_{fun} into Isabelle/HOL needs to be deep enough to reason about the language and its semantics while also being shallow enough, i.e., using enough basic concepts of HOL to facilitate reasoning and simulation of examples. Finite maps are a primitive feature we needed to formalise; this feature is defined closely to the HOL type system to reduce the depth of our embedding.

An object in the ς-calculus is a finite unordered list of named elements that is recursive in its self-parameter: objects *are* finite maps. To enable primitive recursive definitions of functions on terms we need a recursive datatype for objects. The inbred recursion of objects forces us to use a primitive function type to represent these object maps. Thus, we use HOL's primitive map type to define finite maps $\alpha \Rightarrow_f \beta$ by coercing their domain α to be in the type class `finite` of all finite types.

We derive the following induction scheme from the induction rule for finite sets using a domain isomorphism between finite maps and finite functional relations. If a property P is valid for the empty finite map and it is, furthermore, preserved when an element is added to the finite map by updating the map, then the property is true for all finite maps. Note, that for the general function type $\Rightarrow$ such an induction does not hold.

```
⟦ P empty;
  ⋀ x (F:: α ⇒f β) y . ⟦  P F; x ∉ dom F ⟧ ⟹ P (F(x ↦ y))
⟧   ⟹ P F
```

The brackets ⟦. . . ⟧ indicate the conjunction of meta-level hypotheses of a rule. The additional type judgement $\alpha \Rightarrow_f \beta$ coerces F to be an `fmap`.

6.1.2. Binder representation

The formalisation of programming languages in rigorous frameworks, like theorem provers, has revealed some crucial issues summarised in the POPL-mark challenge [16] a set of benchmarks for the mechanisation of language meta-theory. The problem of the representation of binders is there identified as a central problem to the challenges. We discuss in this section the main techniques for representing variable binders laying the ground for the following formalisations.

Problem statement. The representation of binders has already been recognised by Bruijn [17] in the Automath project as a major problem when mechanising languages. Intuitively, a language that has local scopes and parameterisation — for example functions $\lambda x.fx$ — need to refer to the formal parameters — here x — when they occur inside these scopes — here x occurs in the context f. The natural, human understandable way is to use variables, like x, to define and denote formal parameters by name but variables are neither well-suited for mechanisations nor proofs. For example, variable capture may occur, that is, a variable occurring free in a term t may accidentally be "captured" when substituting t inside a scope where x is the name of a bound variable. For example, in $(\lambda x.xy)[\lambda z.x/y]$, the free variable x in $\lambda z.x$ could be captured by the substitution. To avoid this, we use a consistent renaming. Formally, α-equivalence justifies such renamings. However, α-equivalence creates classes of equivalent terms with equal denotation which complicates the semantics. In particular, when fresh variables are a prerequisite inside semantic rules, the choice of α-conversions inside a term predisposes the choice of fresh variables creating an interference that obstructs compositional reasoning.

De Bruijn indices. The classical solution, proposed by N.G. de Bruijn, is to replace each occurrence of a variable by an integer equal to the number of binders that have to be crossed to reach the binder for the considered variable. In other words, a variable is replaced by the distance from its binding scope. Note, the same "variable" may be represented by different integers. For example, the lambda term $\lambda x.x(\lambda y.x\,y)$ in de Bruijn notation is $\lambda(0(\lambda\,1\,0))$; x is once represented as 0, once as 1. The "nominality" of terms is abstracted—semantic denotation becomes unique but substitution becomes very technical because of the "lifting" of indices when entering a binder or replacing a term under binders. Then a term that has to be substituted at nesting depth n into another term needs to add n to all its indices representing free variables. To this end, one first defines a "lift" operation that performs this addition and the substitution then uses lift.

Locally nameless representation. Already at the time of first devising his concept of indices, de Bruijn [17] suggested an alternative where indices represent bound variables (written *bvar i*) and classical named variables represent free (unbound) variables (written *fvar x*); *open* and *close* operations translate between those representations. This technique, known as *locally nameless* representation, has since recently attracted wide attention [15]. It seems very attractive as it combines unique representation provided by de Bruijn indices with human understandable expression of specification of theorems using names—avoiding manipulation of explicit indices, in terms, semantics, and lemmata.

The *open* operation, written t^u, substitutes a term u for the outermost bound variable in the term t. For example $\lambda(bvar\,0\,\lambda((bvar\,1)(bvar\,0)))^n$ is equal to $n\,\lambda(n\,(bvar\,0))$. The opposite operation *closes* a term: given a name, the closing replaces the occurrence of variables of this name with an index for a bound variable, such that the variable is bound at the outermost level of the term.

A drawback of the locally nameless approach is that we need to take explicit care that we do include only well-formed terms, i.e., only *bound* variables are represented by indices. The notion of locally closed terms ensures this. E.g., $\lambda(bvar\,2)$ is not locally closed. Ensuring that we manipulate only locally closed terms will have to be added as prerequisite to our propositions when dealing with locally nameless representation. Another problem arises when reducing a term under a binder. Here, we should close the term under a *fresh variable* (to keep the term locally closed). Formally, we need: $\forall x \notin FV(t).t^x \to (t')^x \implies \lambda(t) \to \lambda(t')$. The drawback of this approach is that it is sensitive to the set of free variables, that may vary in an unexpected way. Here, the approach of *cofinite quantification* [15] is an important step forward. The basic idea is to abstract over the set of free variables $FV(t)$ and to let a fresh variable be taken among the complementary of an *existentially quantified* finite set L, the proposition above becomes: $\exists L$ finite.$\forall x \notin L.t^x \to (t')^x \implies \lambda(t) \to \lambda(t')$. This set L can then be instantiated appropriately when handling proofs.

Nominal techniques. Another approach, proposed by Urban and et al. [18] based on work on nominal logic by Pitts [19], is called nominal techniques. Here, terms are identified as a set bijective to all terms factorised by α-equivalence. Instead of using substitution, nominal logic uses permutations of atomic names. Permutations are built from elementary *name swaps*: e.g., $(a, b) \cdot t$ replaces all occurrences of a by b and *vice versa* in t. Permutations are only applicable if there are fresh atoms available. This is expressed by keeping track of the support set (fresh atoms). The classical hypothesis, "there is a fresh variable" for a term t is replaced by, "there is a finite *support* for x", i.e., the set of atoms used in t is finite, and infinitely many "fresh" atoms are available. Unfortunately, the Isabelle/HOL package implementing nominal techniques cannot be used as it is — in our case — because we use finite maps in our implementation; consecutively the recursive datatype defining ASP_{fun} syntax is a bit more complex than the usual simple recursive construction. While it is trivial that a finite map containing terms of finite support has a finite support, such a reasoning is not yet supported by Urban's package.

Higher order abstract syntax. Another technique for formalising binders is Higher Order Abstract Syntax (HOAS) in which binders of applications are directly represented by binders of the meta-level, e.g., [20,21]. Therefore, by contrast to the above sketched approaches, HOAS is often also called the *direct* encoding. For example, in Isabelle/HOL, we would use the HOL λ-abstraction to encode object-level binders. This approach has advantages in terms of mechanisations: reductions are usually performed automatically but it is restricted when it comes to meta-level reasoning. Sometimes, "meta-theoretic properties involving substitution and freshness of names inside proofs and processes, cannot be proved inside the framework and instead have to be postulated" [22].

6.2. Crucial aspects of the proofs

This section details some of the parts of the formalisation that seem the most important to us, it gives proof sketches, and is not much coupled with Isabelle/HOL.

6.2.1. Finiteness

When considering language semantics we often implicitly assume finiteness of programs and configurations. In fact, the implicit assumption is worth mentioning: for programs it grants induction over the recursive datatype of ς-terms, and for configurations, it permits the assumption that there are always fresh activity and future names available. Our formalisation relies on this assumption. We particularly highlight the fact that it becomes necessary to show progress. For example, to create a new activity one must find a fresh identifier. We have shown that initial configurations and configurations reduced from them are all finite: they have a finite number of activities and futures.

6.2.2. Absence of cycles

Proving the absence of cycles (Theorem 1) required us several steps. We first defined a datatype for future or activity reference and then specified the $knows_C$ and $knows_C^+$ relations defined in Section 4.2. In order to handle the proofs, we refine the $knows_C^+$ relation by remembering the list of intermediate activities: $r\ knows_C^{+\,(L)}\ r'$ iff $r\ knows_C^+\ r'$ passing by the references in L.

We first prove lemmata relating cycles, $knows_C^+$, and paths. E.g., if $r\ knows_C^{+\,(L)}\ r'$ and C' is obtained from C by just modifying the request corresponding to f_k, then $r\ knows_{C'}^{+\,(L)}\ r'$ provided $f_k \notin L$ and $f_k \neq r$. A similar lemma exists for activity references. Consequently, it is sufficient to prove that no cycle is created by the activities and requests modified by the considered reduction. We also show that, when $r\ knows_C^{+\,(L)}\ r'$, L can be chosen to include neither r nor r'.

The main proof of absence of cycles is a long case analysis on the reduction rules that uses lemmata presented above, well-formedness of the initial configuration, and shows that if there is a cycle in the obtained configuration, there was necessarily one in the original configuration. As an example, we detail the main argument for the REQUEST rule referring to the rule of Table 2 with C_1 being the source configuration and C_2 the obtained one. One can first note that, as the source configuration is well-formed by hypothesis, only f_i may refer to f_k in C_2. Second, if $f_k\ knows_{C_2}\ r$ then either $\beta\ knows_{C_1}\ r$ or $f_i\ knows_{C_1}\ r$. We only have to show that $\exists L.f_k\ knows_{C_2}^{+\,(L)}\ f_k$. By contradiction and induction on the length of L, length 0 is impossible because β or f_i would know f_k in C_1 which would not be well-formed. For greater lengths, $L = L'\#r$, and necessarily $r = f_i$ as shown above. Thus, $f_k\ knows_{C_2}^{+\,(L')}\ f_i$, where $f_k \notin L$ and $f_i \notin L$. Consequently, $f_i\ knows_{C_2}^{+\,(L')}\ f_i$ or $\beta\ knows_{C_2}^{+\,(L')}\ f_i$ as the request for f_k is only built from the request for f_i and the active object of β ($t'.l(s)$ in Table 2). Since $f_k \notin L$ and $f_i \notin L$, and only f_i and f_k have been modified between C_1 and C_2: $f_i\ knows_{C_1}^{+\,(L')}\ f_i$ or $\beta\ knows_{C_1}^{+\,(L')}\ f_i$. As $f_i\ knows_{C_1}\ \beta$, in either case there would be a cycle from f_i in C_1, which is contradictory.

Fig. 3, page 833 illustrates this case of the proof. It considers the case of a REQUEST from α to β creating the future f_l and the reference depicted by the arrow 6. Additionally, suppose a cycle is created: arrows 3, 4, 5, 6 in the figure, we consider the sub-case where this cycle was created because of a reference in the active object in β. We decompose the cycle into $f_l\ knows_{C_2}^{+\,(L')}\ f_i$, with L' consisting of the arrows 3, 4, 5 on the figure, plus arrow 6 ($f_i knows_{C_2} f_l$). Then, necessarily, before the reduction f_i was involved in a cycle passing by β and by the path consisting of the arrows 1, 2, 4, 5, which shows the contradiction.

6.2.3. Typing and subject reduction

Subject reduction is handled in two phases, each proved by case analysis: one for local and one for distributed reduction. We detail below a few useful lemmata. The first lemma states that any term that has a type in an empty environment has no free variable:

$$\langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash a : A \ \Rightarrow\ noFV(a)$$

Conversely, a term without free variable can be typed in an empty environment (in fact, below we could prove $A = A'$ but this was not useful):

$$\langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash a : A \ \wedge\ noFV(a) \ \Rightarrow\ \exists A'.\langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash a : A'$$

Both preceding properties are necessary to show that for an activated object or a new request a type can be found.

$$\langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash E[a] : A \ \wedge\ \langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash a : B \ \wedge\ \langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash b : B \Rightarrow \langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash E[b] : A$$

This lemma is both crucial and interesting because it relates contexts and typing. As our reduction relies on the use of contexts, this lemma is decisive for the proof of subject reduction, Theorem 2.

6.2.4. Proving progress

Proving progress relies on a long case analysis on the reduction rules. We focus first on one crucial argument: how can the absence of free variable be ensured in order to communicate an object between two activities. Each request can be typed in an empty environment (for variables); thus it does not have any free variable, and thus each sub-term of a request that is not under a binder has no free variable. We prove that one can reduce at least the part of the request under the evaluation context F, where $F ::= \bullet \mid F.l_i(t) \mid F.l_i := \varsigma(x, y)\, s \mid Active(F)$. If one replaces E by F in the semantics, this prevents reduction to occur inside the binders. Indeed, in F the term in the position of the hole has no free variables: $\langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash F[a] : A \ \Rightarrow\ noFV(a)$.

Considering the other arguments of the proof, the absence of cycles ensures that an application of a REPLY rule cannot return a future value which is the future itself, in which case the configuration would be reducible but to itself. This ensures that no live-lock exists in the distributed semantics even if the local one can create live-locks. Of course, the proof also massively uses the fact that well-typed configurations are well-formed.

6.3. The formal model in Isabelle/HOL with de Bruijn indices

This section presents a first version of the formalisation of ASP_{fun}, its syntax, and a few theorems in Isabelle/HOL; this version relies on de Bruijn indices. The main objective of this section is to give a real feel for the Isabelle/HOL formalisation

and outline the main steps of the formalisation process. We use here the de Bruijn representation for the syntax of ASP_{fun} but the major part of the formalisation process is similar for the locally nameless representation presented in the subsequent section.

6.3.1. Syntax

The formalisation of functional ASP is constructed as an extension of the base Isabelle/HOL theory for the ς-calculus [14]. The term type of the ς-calculus is represented by an Isabelle/HOL datatype definition called dB. In this datatype definition, objects are represented as finite maps `Obj (Label ⇒f dB) type`. We formalised finite maps in the first argument of `Obj` using the abstract concept of axiomatic type classes. As discussed in Section 6.1.1, it is crucial to have finite maps as a basic Isabelle/HOL type to be able to employ the recursive datatype construction here. The second argument of the constructor `Obj` is a type annotation. The resulting datatype for basic terms of ASP_{fun} is then as follows. Variables are represented by de Bruijn indices. A given index has two entries: one for self, and the other for the parameter as defined by the datatype `Variable`.

```
datatype Variable = Self nat | Param nat

datatype dB =                (*The typed ASPfun datatype*)
    Var Variable              (*Variable - deBruijn index*)
  | Obj "Label ⇒f dB" type(*Objects map labels to terms, and have a type*)
  | Call dB Label dB          (*Call a l b calls meth l on a with param b*)
  | Upd dB Label dB           (*Upd a l b updates meth l of a  with body b*)
  | Active dB                 (*Creates an active object*)
  | ActRef ActivityRef        (*References an active object - dynamic syntax*)
  | FutRef FutureRef          (*References a future - dynamic syntax*)
```

The type of configurations relies on partial functions expressed by the constructor ⇸.

```
futmap = FutureRef ⇸ dB
configuration = ActivityRef ⇸ (futmap × dB)
```

6.3.2. Reduction contexts in Isabelle

In our model we developed a simple mechanisation of a reduction context using again the datatype construct as follows:

```
datatype general_context = (*a general context is a term with a hole*)
    cHole
  | cObj FmapLabel type general_context
  | cCallL general_context Label dB
  | cCallR dB Label general_context
  | cUpdL general_context Label dB
  | cUpdR dB Label general_context
  | cActive general_context;
```

Isabelle/HOL internally generates rules for a datatype specification most notably induction rules for recursive types and injectivity rules for the constructors. Pattern matching facilitates case analysis proofs crucial for reasoning with complex languages.

This representation of contexts by a specific datatype constructor exploits the power of the efficient datatype feature of Isabelle while at the same time finding a first class representation of the syntactical concept of "context". For the use of contexts we define an operator to "fill" the "hole" enabling a fairly natural notation of `E↑t` for $E[t]$ (remember this substitution is not "capture avoiding" contrary to the variable substitution).

```
consts Fill :: [general_context, dB]  ⇒  dB ("↑")
```

We use this simple function to illustrate the definition of functions in Isabelle/HOL. Functions over datatypes may be defined in a particularly efficient way in Isabelle/HOL using primitive recursion. Efficient means, in this context, that proofs involving these operators may be mostly solved automatically using automatic rewriting techniques provided in Isabelle. The semantics of the `Fill` operator is described by the following set of equations — again, this substitution is, unlike variable substitution, not "capture avoiding".

```
primrec
  Fill cHole x = x
| Fill  (cObj f T E ) x =  Obj ((FLmap f)((FLlabel f) ↦ (Fill E x))) T
| Fill  (cCall E l) x =  Call (Fill E x) l
| Fill (cUpdL E l (y::dB)) x = Upd (Fill E x) l y
| Fill (cUpdR (y::dB) l E) x = Upd y l (Fill E x)
| Fill (cActive E) x = Active (Fill E x)
```

The rest of this section intensively use this operator and thus illustrates its usefulness.

6.3.3. Semantics

The parallel semantics of ASP_{fun} is given as an inductive relation over this type of configurations encoding the reduction relation $\rightarrow_{\parallel}$ (see Table 2). To give some flavor of the expression of the semantic, we depict only the rule REQUEST; this rule is a crucial one for the calculus, and it gives a representative idea of the other semantic rules. This rule is part of an inductive definition for the reduction relation $\rightarrow_{\parallel}$ on configurations. An inductive definition in Isabelle/HOL defines a set, here the relation $\rightarrow_{\parallel}$, by a set of simple rules. The set defined by an inductive definition is the least set that is closed under those rules.

```
request:
 ⟦ ∀ D ∈ dom C. fk ∉ dom(fst(the (C  D))); C A = Some(m',a');
    m'(fi) = Some(E↑(Call(ActRef B) l s)); C B = Some(mb, t'); noFV s; A≠ B ⟧
⟹ C →∥ C(A ↦ (m'(fi ↦ E↑(FutRef(fk))), a'))(B ↦ (mb(fk ↦ (Call t' l s)), t'))
```

Assumptions are enclosed in Isabelle/HOL's meta-logic brackets ⟦⟧, and conclusion is placed after ⟹. Additionally, a partial function admits a `dom` operator defining the domain of the function, and a partial function returns either `None`, if the function is not defined for this value, or `Some(x)` if the function is defined and returns `x`. `C(A ↦ x)` represents the partial function `C` where `A` is now given the value `x`.

The above code for the `request` rule in Isabelle clearly corresponds to the following rule of the semantics of ASP_{fun}. As one can notice, the main differences in Isabelle are that, first the definition "fresh" has been directly encoded in the rule, and second a few assumptions were used to decompose the source configuration, e.g., `C A' = Some(m',a')` states that the activity `A` of configuration `C` is defined by the couple `m` (the request queue) and `a` (the active object). Even with these minor differences, it is easy to see that both rules express the same behaviour.

REQUEST

$$\frac{f_k \text{ fresh} \qquad noFV(s) \qquad \alpha \neq \beta}{\alpha\,[f_i \mapsto E[\beta.l(s)] :: Q, t] :: \beta[R, t'] :: C \rightarrow_{\parallel} \alpha\,[f_i \mapsto E[f_k] :: Q, t] :: \beta\left[f_k \mapsto t'.l(s) :: R, t'\right] :: C}$$

6.3.4. Typing and progress

We skip the description of the proofs related to well-formedness and decide to focus on typing. We first define the following datatypes for object type and configuration type and a constant `typing` for typing judgements. The syntactic sugar: `CT, E ⊢ a : A` abbreviates `(CT, E, a, A) ∈ typing`.

```
datatype type = Object (Label ⇒f (type × type))
datatype Ctype =  TConfig (ActivityRef ⇸ type)(FutureRef ⇸ type)
typing :: [Ctype, ((type× type) list), dB, type] ⇒ bool
```

The most remarkable point in the signature above is the use of `(type×type) list` instead of finite maps from variables to types (cf. Section 6.4). A list is sufficient because of the use of de Bruijn indices: the depth in the list represents the de Bruijn index; and a couple of types is necessary because one represents the type of self, and the other represents the parameter type.

Then this relation `typing` is defined using an inductive definition. The rules of the inductive definition are exactly the typing rules for ASP_{fun} introduced in Section 5. For comparison we show just the rule TYPE CALL.

```
⟦Tconf, env ⊢ a : A; l ∈ dom A; A!l=(B,T);Tconf, env ⊢ b : B⟧
⟹ Tconf, env ⊢ (Call a l b) :  T
```

The operator `!` selects a type field `l` in an object type `A`. Typing for configurations is also defined as presented in Section 5. We completely proved in Isabelle/HOL all the theorems presented in this paper. Theorems are expressed similarly in Isabelle as in the paper version. Below is the subject reduction theorem (Theorem 2). Note that, as ⟹ can only be used at the top-level, ⟶ is used to denote implication inside formulae:

```
theorem Csubject_reduction:    ⊢C: CT ⟹ (∀C'. C →∥ C' ⟶ ∃CT'. ⊢C': CT')
```

The theorem `progress_ASP_init_conf` below is a particular instance of the progress theorem employing the previous results that all reachable configurations are finite and have no cycles; it corresponds to Property 7.

```
theorem progress_ASP_init_conf:
 ⟦init_config a →∥ C; TConfig empty empty, [] ⊢ a : T; A∈ dom C; fi∈C.RA⟧
⟹ (isresult  C.FA<fi>) ∨
   (∃C'. (C →∥ C') ∧(C'.FA<fi>≠C.FA<fi>∨ C.FA<fi> →ς C.FA<fi>))
```

6.4. Locally nameless representation

The main advantage of the de Bruijn representation is also its biggest handicap: indices instead of variables get rid of α-conversion problems but are very technical. An unwelcome effect of the lifting and substitution functions, necessary for index handling, is that there are many lemmata that are hard to find and difficult to prove. Their difficulty is not

their theoretical depth but that they merely shuffle indices—a facility easy for a machine and hard for a human mind. An illustrative example is the following lemma `subst_subst` proving how two substitutions can be swapped.

```
i < j + 1 ⟹
t[lift v i,lift s' i / Suc j][u[v,s'/j],s[v,s'/j]/i] = t[u,s/i][v,s'/j]
```

The locally nameless representation, on the other hand, is closer to paper style notation due to the use of named free variables in addition to indices. The price to pay for the gained understandability are additional concepts. Consequently, new hypotheses in rules and theorems arise. We believe that both representations have their merits and their weaknesses as we will point out in the following exposition of the locally nameless representation of ASP_{fun}.

6.4.1. Basic constructs

The only difference of the locally nameless representation to the de Bruijn representation concerning the terms is the addition of named free variables. This new type `fVariables` is conveniently chosen to be the type `string`. The datatype of terms stays the same (it is named `term` now instead of `dB`)—only the constructor `Var` is replaced by two new constructors `Bvar` and `Fvar`, the former taking an index and the latter a free variable. Also at the level of configurations there is not much difference: the type of configurations actually stays the same. In the parallel semantics, the only difference is in the rule LOCAL where local terms need to be *locally closed* in order to be reduced according to the local semantics. The main differences in the locally nameless semantic definition is in the reduction relation for the evaluation of the objects. Here, the new concept of named variables is supported by operations for opening and closing of terms.

Opening and closing

Opening is a form of substitution; it corresponds to an instantiation of a bound variable with a given subterm. While the following definition's core part is the first clause, the others just pass the recursion into the term structure. This first clause replaces a bound variable if `n` matches the index of the parameter. Due to the two parameter types of our terms, we always open with a pair of terms and replace depending on whether the bound is `Self` or `Param`, by the first or second element of the pair, respectively.

```
primrec
open :: [nat, term, term, term] ⇒ term ("{_ → [_,_]} _")
and
open_option :: [nat, term, term, term option] ⇒ term option
where
  open_Bvar: {k→[s,p]}(Bvar b) =
      (case b of (Self i) ⇒ (if (k = i) then s else (Bvar b))
                 | (Param i) ⇒ (if (k = i) then p else (Bvar b)))
| open_Fvar: {k→[s,p]}(Fvar x) = Fvar x
| open_Call: {k→[s,p]}(Call t l a) = Call({k→[s,p]}t) l ({k→[s,p]}a)
| open_Upd : {k→[s,p]}(Upd t l u) = Upd({k→[s,p]}t) l ({(Suc k)→[s,p]}u)
| open_Obj : {k→[s,p]}(Obj f T) = Obj(λl.open_option(Suc k) s p (f l)) T
| open_Act : {k→[s,p]}(Active a) = Active ({k→[s,p]} a)
| open_ARef: {k→[s,p]}(ActRef g) = ActRef g
| open_FRef: {k→[s,p]}(FutRef f) = FutRef f
| open_None: open_option k s p None = None
| open_Some: open_option k s p (Some t) = Some ({k→[s,p]}t)
```

Let us only describe the most characteristic of the other clauses: `open_Obj`. Recursive opening inside the object is defined by mapping a function (`λl. ...`) on all its methods (most of them being undefined, `None`). This explains why we use two mutually recursive functions `open` and `open_option`, one of them accepting `Some term` or `None`. The function applied to each member method is the recursive application of open but with `Suc k` as index, because we entered a binder (similarly to what we would do for de Bruijn method).

Open is usually used to replace the outermost binder, i.e., `{0 →[s,p]} t` abbreviated by $\texttt{t}^{\texttt{[s,p]}}$. For example, one crucial rule of our semantics of objects is to evaluate calls to an object's method $[l_j \mapsto \varsigma(x, y)t, \ldots].l_j(p)$ to the body with substituted parameters: $t[o/x, p/y]$, where $o = [l_j \mapsto \varsigma(x, y)t, \ldots]$. In locally nameless representation, it is expressed by $\texttt{t}^{\texttt{[o,p]}}$.

To abstract a variable, `close` is defined as a primitive recursive function of type `[nat, fVariable, fVariable, term] ⇒ term`. As close corresponds to a method abstraction we chose the syntax `{_ ←[_,_]} _`. Its definition uses identical patterns with open; we thus only show the decisive case for `Fvar`.

```
close_Fvar: {k ← [s,p]}(Fvar x) = (if x = s then (Bvar (Self k))
                    else (if x = p then (Bvar (Param k)) else (Fvar x)))
```

Similarly to open, most of the time we will close the variable indexed by 0; we thus abbreviate `{0 ←[s,p]} t` by σ`[s,p] t`.

Opening and closing efficiently convert between free and bound variables. Remember, however, that the coexistence of free and bound variables necessitates to restrict propositions to terms without "unbound bound variables": preconditions generally restrict propositions to *locally closed terms*.

The predicate `lc` formalises local closure:

```
inductive lc :: term ⇒ bool
where
  lc_Fvar: lc (Fvar x)
| lc_Call: ⟦ lc t; lc a ⟧ ⟹ lc (Call t l a)
| lc_Upd: ⟦ lc t; finite L; ∀s p. s ∉ L ∧ p ∉ L ∧ s ≠ p ⟶ lc (u[Fvar s,Fvar p])⟧
                                ⟹ lc (Upd t l u)
| lc_Obj: ⟦ finite L; ∀l∈dom f. ∀s p. s ∉ L ∧ p ∉ L ∧ s ≠ p
                                ⟶ lc (the(f l)[Fvar s,Fvar p]) ⟧
                                ⟹ lc (Obj f T)
| lc_Act: lc a ⟶ lc (Active a)
| lc_ARef: lc (ActRef g)
| lc_FRef: lc (FutRef f)
```

An explicit substitution operator with syntax `[x →s] t` replaces a free variable `x` by a term `s` in a term `t`. The structure of its primitive recursive definition is similar to open and close but the decisive `Fvar` case is as follows.

```
subst_Fvar: [z → u](Fvar x) = (if (z = x) then u else (Fvar x))
```

Although we use open for a "substitution" in the semantics, the substitution above is better suited for free variables for example in renaming lemmata.

6.4.2. Cofinite quantification

One problem when changing from a bound to a free variable is the need for fresh variables. Whenever we have a rule which uses a newly introduced variable name, we need to find a fresh name. For example, suppose that `t` is a subterm under a binder. To make it locally closed, we need to instantiate the top-level bound variable of `t`: $t^{[s,p]}$, but to keep the original term `t` (and close the term later with `s` and `p`), we need `s` and `p` fresh. Technically, we can use a function `FV` collecting the free variables of a term and add the additional premise $x \notin FV(t)$ whenever a fresh variable name x is required. This way of formalising can be described as the "exists-fresh" approach [15]. Unfortunately, the "exists-fresh" approach leads to very clumsy proofs: intuitively, we need to prove statements for a set of free variables differing from the ones given as hypotheses. In recent work by Aydemir et al. [15], a more sophisticated technique called cofinite quantification is introduced that eases the proofs involving such rules. The basic idea (cf. Section 6.1.2) is to abstract from sets of free variables $FV(t)$, but instead consider some arbitrary finite set L, i.e., assuming a "cofinite set" of variable names. Since L is arbitrary, it can be chosen later as a convenient set bigger than the set of free variables. Any naïve way using simply locally nameless representation *without* using cofinite induction in the semantic definition would lead to unsolvable proof obligations for some theorems. Thus the semantics of our calculus in the locally nameless representation is expressed by rules of the form:

COFINITE-UPDATE-LN

$$\frac{\text{finite } L \qquad \forall x\, y.\, x{\neq}y \wedge x, y \notin L \longrightarrow \exists t''.t^{[x,\ y]} = t'' \ \wedge\ t' = \varsigma[x,y]t'' \qquad \texttt{lc}\ o}{o.l := t \rightarrow_\varsigma o.l := t'}$$

6.4.3. Semantics and proofs

When comparing the techniques, two criteria must be considered: how easy it is to write the formalisation, and how easy and convincing it is to read it. Locally nameless terms are definitely easier to read as they use named variables instead of de Bruijn indices. However, in the specification of the syntax and semantics we often encounter some technical overhead due to the new constructors for named free variables. Moreover, we need to establish the well-formedness of terms by adding predicates *lc* to the premises of the reduction rules. Fortunately, the additional *lc* condition mainly states that substituted terms correspond to correct ς-calculus terms. We have seen an example in the previous section when considering the semantic rule COFINITE-UPDATE-LN for the local update on objects.

Let us focus on the reduction inside binders. Specifying that any field can be reduced in de Bruijn notation leads to the rule:

```
Obj: ⟦s →ς t; l ∈ dom f⟧⟹ Obj (f (l ↦ s)) T→ςObj (f (l ↦ t)) T
```

This is very similar to the paper version. The locally nameless version is less straightforward: we need cofinite quantification:

```
Obj: ⟦ l ∈ dom f; finite L; lc (Obj f T);
      ∀s p. s∉L ∧ p∉L ∧ s≠p⟶ ∃t''. t[Fvar s,Fvar p] →ς t'' ∧ t' = σ[s,p] t'') ⟧
      ⟹  Obj (f(l ↦ t)) T →ς Obj (f(l ↦ t')) T
```

Additional requirements refine what is meant by "reduce under the binder"; in fact the difficulty is to make the sub-term under the binder locally closed before reducing it, which somehow refines the intuitive notion of (correct) reduction under binders.

The essential relations of the calculus, reduction and typing, are not more readable in the locally nameless versions compared to the de Bruijn incarnations. In both formalisations, the introduction of syntactic sugar can bring some rules very close to a paper version. However, the more restrictive reduction relation for locally nameless variables is closer to the version found on paper, as it does not apply to terms with dangling indices.

Concerning proofs, the notable benefit comes from the explicit distinction between the variable types, which can improve readability and ease reasoning for many lemmata, especially the basic lemmata and confluence proofs, more cases being proved automatically.

Concerning typing, the locally nameless formalisation improves the understandability of proofs but at the price of rather technical lemmata for renaming. We are not able to observe a major improvement in the complexity of the major proofs but, for the most part, there is no notable burden either. The proof principles are similar for either variable representation.

6.4.4. Overall comparison with the de Bruijn approach

The clear advantage of the locally nameless formalisation is the handling of free variables. The de Bruijn version did not allow reasoning about free variables for a very simple reason: it is not possible to express free variables. More precisely, unbound de Bruijn indices could sometimes simulate free variables, but such a solution is unsatisfactory because the intent of a free variable is different from a dangling index. Moreover, the explicit distinction between bound and free variables eases the handling of either kind of variable and enhances the readability of proofs and formalisations.

Cofinite quantification, freshness, and renaming are the major reasons for additional and technical proofs in the locally nameless representation, and all of these items are required for the reasoning about named free variables. The locally nameless rules are more complex than their de Bruijn counterparts because the locally nameless representation introduces new concepts and is precise about well-formedness and closure. This initial formal overhead is paid back by a natural notation in theorems and by improvement for interactive proofs. Overall, the locally nameless technique allows a more precise formalisation, avoids proving obscure lemmata on substitution and lifting, and leads to a more natural notation for terms but at the price of additional non-trivial requirements in semantic and typing rules, and additional non-trivial concepts.

6.5. An experience in the formalisation of calculi and semantics

The entire development takes around 14 000 lines of code for each of the two representations. Among those lines less than 10% are necessary for the formalisation of the languages and the properties, and most of the development concerns the proof of the properties and the intermediate lemmata. The development time is difficult to evaluate but is above one man-year for the two formalisations.

The most difficult and crucial step is certainly the definition of the right model for the calculus, its semantics, but also for the additional constructs used in intermediate lemmata. Of course, the structure and difficulties of the proofs are highly dependent on the basic structures on which the formalisation relies.

Even if the length and form of the proof is not optimal, the development for formalising such a theory is really consequent; and it becomes difficult to keep a proof minimal and well-structured when it grows to several thousands of lines in length. Handling simplification steps in such a complex and rich theory becomes tricky. Additionally, making modular proofs for subject reduction and equivalent properties is difficult in a theorem prover because useful lemmata are tightly coupled with the numerous and complex hypotheses involved by the case analysis; for example it is difficult to specify a lemma that will be used in case the REQUEST rule has been applied, because such a lemma would have numerous and complex hypotheses.

However, globally, we consider that the formalisation of ASP_{fun} is of reasonable size, and provides a set of constructs relatively easy to use. We think this formalisation can be used efficiently to prove new properties on distributed object languages.

7. Discussion and alternative semantics

Reduction contexts. There are different ways of specifying at which point(s) of a term a reduction can occur. A convenient and classical technique for this is to use reduction context (a term with a hole). Reduction occurs at the position of the hole, and the definition of the contexts define the possible reduction points. The most operational semantics generally reduce innermost terms and implement a call by value for method parameters. The most general semantics, like the classical semantics of λ-calculus, allows reduction to occur at any point in the term.

Because it gives the most general results, we chose the general semantics where any part of the terms can be reduced. In particular, we allow the reduction to occur inside binders. This is similar to the general semantics of ς-calculus, as in Definition 6.2.1 of [1] or even example page 62 showing a reduction inside binders. Then for their "operational semantics", in Section 6.2.4 of [1], Abadi and Cardelli [1] use *reduction contexts* that do not allow reduction inside binders: $F ::= \bullet \mid F.l_i(t) \mid F.l_i := \varsigma(x, y)\, s \mid Active(F)$. In ASP_{fun}, these reduction contexts would avoid using the *noFV* requirement in the reductions. We chose to specify the most general semantics—allowing reduction inside binders.

Properties and proofs presented in this paper are also valid for reduction contexts (replacing E by F), and reformulating our results for reduction contexts would be trivial. Indeed all the properties are trivially easier to verify for the reduction with F except progress (all of them are more general for E than for F). But, progress was proved using exactly this reduction context. Consequently progress is also verified by the reduction context semantics.

Communicating non-closed terms. In our semantics we prevented terms with free variables to be communicated in order to avoid variables to escape their binders. Technically, all communication rules require the communicated term s to verify "$noFV(s)$". To avoid this requirement in the semantics an alternative semantics could be provided to communicate free variables without entailing shared memory; but this is out of the scope of this paper, see [23] for example.

Optimising parallel evaluation. A few drawbacks could be found in the semantics given in this paper if a real programming language was to be implemented exactly as specified in Table 2. Indeed a straightforward implementation of our semantics could allow some inefficient execution paths especially because too many communications or computations could occur if no optimisation is done.

First, it seems unreasonable to create in practice as many threads as there are requests in an active object: using a thread pool seems a much better implementation choice.

Additionally, the most critical inefficient point is the possibility to return a future partially evaluated, i.e., the result for a request partially computed. This can result in the computation being done twice which is, in general, not efficient. However, the properties proved here allow enough variation on the semantics to make it usable in practise. In our critical example, it is possible to restrict the REPLY rule to only return completely evaluated futures. Then, if one picks a request, there is no more any guarantee that it can evolve, but the absence of cycle ensures that *some request in the configuration can always be reduced*. Some intermediate reductions have to be added to guarantee the progress property: we first reduce the request(s) calculating the future value before returning the future and progressing. Finally, *returning only completely evaluated futures leads to a more efficient semantics, and still ensures a (weaker) form of progress*.

8. Related works and positioning

Distributed languages: Actors and objects

Actors [5] is a widely used paradigm for programming distributed autonomous entities and their interactions by messages. They are rather functional entities but their behaviour can be changed dynamically giving them a state.

Agents and Artifacts with simpA, concentrating on the higher level of modelling concurrent agent based systems, also feature a calculus [24]. Although the formalisation is based on Featherweight Java, the agent concept of Agents and Artifacts resembles ASP_{fun}'s activities but the calculus has no type system and proofs. ASP_{fun} framework may be used to provide formal support to this work.

Obliq [25] is based on the ς-calculus; it expresses both parallelism and mobility. It relies on threads communicating with a shared memory. Like in ASP_{fun}, calling a method on a remote object leads to a remote execution but this execution is performed by the original thread. Øjeblik, e.g., [26], a subset of Obliq, equally differs from ASP_{fun} by thread execution. The authors investigate safety of *surrogation* meaning that objects should behave the same independent of migration.

The distributed object calculus by Jeffrey [27] is based on a concurrent object calculus by Gordon et al. [28] extended with explicit locations. The main objective is to avoid configurations where one object at one location is being accessed by another. A type system enforces these restrictions. Because migrating objects can carry remote calls, in order to ensure subject-reduction, Jeffrey introduces serialisable objects, which are non-imperative. Compared to our calculus the most decisive difference is that *activities abstract away the notion of location* and are remotely accessible thanks to a request queue. The concept of futures somehow explicitly supports mobility and serialisation.

Futures

Futures have been studied several times in the programming languages' literature originally appearing in Multilisp [29] and ABCL [30].

$\lambda(fut)$ is a concurrent lambda calculus with futures. It features non-determinism primitives (cells and handles). Niehren et al. [13] define a semantics for this calculus and two type systems. They use futures with explicit creation point in the context of λ-calculus; much in the same spirit as in Multilisp. Alice [31] is an ML-like language that can be considered as an implementation of $\lambda(fut)$.

In [32], the authors provide a language with futures that features "uniform multi-active objects": roughly each method invocation is asynchronous because each object is active. Thus, compared to ASP_{fun}, the calculus has no *Active* primitive. Each object has several current threads, but only one is active at each moment. Each object holding a future may block waiting for the future, or it may use the `await` construct to release the current thread and activate a new one. In this framework, futures are also explicit: a `get` operation retrieves their value. The authors also provide an invariant specification framework for proving properties. This work also formalises the Creol language [11]. Indeed, Creol has exactly the same notion of uniform multi-active objects, and of a single thread active at a time. Johnsen et al. [11] also provide a type system specifying behavioural interfaces, and a semantics for Creol in Maude. Also note that [33] provide a model of Creol's multi-active objects with futures but they focus on the definition of interfaces and on a safety property on *promises* (a generalisation of futures).

To summarize, the main difference between Creol and ASP_{fun} are that future creation and access is explicit in Creol, all Creol objects are active, and the functional nature of ASP_{fun}.

ASP [9] is an imperative distributed object calculus; it is based on the $\varsigma_{\mathbf{imp}}$-calculus [1]. It features asynchronous method calls and transparent futures. No instruction deals directly with futures. Activities in ASP are mono-threaded: one request is served at each moment, and a primitive can be used to select the request to serve. Some confluence properties for ASP have been studied in [9,8]. ProActive [7] is an implementation of the ASP calculus.

Dedecker et al. [34] suggest a communication model, called *AmbientTalk*, based on an actor-like language and adapted to loosely coupled small devices communicating over an ad-hoc network. The communication model is quite similar to the ASP calculus but with queues for message sending, handlers invoked asynchronously, and automatic asynchronous calls on futures. The resulting programming model is slightly different from ASP and ASP_{fun} because there is no blocking synchronisation in AmbientTalk. In AmbientTalk, the flow of control might be difficult to understand for complex applications, because one can never ensure that a future has been returned at a precise point of the program. AmbientTalk should be dead-lock free but, unfortunately, as no formalisation of the language has been proposed to our knowledge, this has not been formally proved. Our framework could be relatively easily adapted to prove the absence of dead-locks in AmbientTalk by transferring our progress property.

Concerning analysis of programs with futures, Cansado et al. [35] proposed an automatic way to generate a model of a component application with futures in order to verify its correct behaviour. Note that the objective of our paper is quite different because we aim here at proving generic properties of languages that handle futures whereas [35] aim at proving properties of a specific application. However, generic properties proved in ASP_{fun} for the programming model are directly used in the verification approach to know that the specified model fits the reality but also to optimise verification procedures by using generic properties of the language.

Mechanical proofs for calculi

One of the greatest contributions of this work is the formalisation of the ASP_{fun} language, its semantics, and type system plus the proof of safety and progress in an interactive theorem prover. We believe that in the discipline of language development the application of mechanical verification is particularly relevant even if it comes at the price of intensive and partly cumbersome work. Related works from the viewpoint of mechanised language verification is the formalisation of the imperative ς-calculus in the theorem prover Coq most prominently using a co-inductive definition and higher order abstract syntax by Ciaffaglione et al. [21]. However, they do not consider concurrency or distribution. With respect to concurrency, the formalisation of the π-calculus in Isabelle/HOL by Roeckl and Hirschkoff [20] is related. There, higher order abstract syntax is employed. More recent work by Bengtson and Parrow [36] uses nominal techniques in Isabelle/HOL for the formalisation of the π-calculus. The authors prove many standard results concerning bisimulation and congruence of the calculus. In recent work, they formalised their own generalisation of the π-calculus, the Psi-calculus [37]. Concerning mechanisation of calculi, their solution to model *binding sequences* for nominal datatypes in Isabelle/HOL is interesting because it also shows that generalisations of the nominal package in Isabelle/HOL are necessary and possible (see Section 6.1.2). Unfortunately the design of the π-calculus is too far from ASP_{fun} for this formalisation to be directly useful in our case. Moreover, no objects are introduced neither in the π-calculus nor in its extensions. Ridge [38] works on a formalisation of concurrent OCaml in Isabelle/HOL. However, he concentrates on concurrency using abstraction techniques to improve automation of concrete algorithm proofs and has not formalised objects at all. The originality of our approach lies in the formalisation of distribution concerns and futures.

Positioning

Futures have been formalised in several settings generally functional-based [13,32,39]; those developments rely on explicit creation of futures by thread creation primitives in a concurrent setting. They are getting more and more used in real life languages; for example, explicitly created futures are also featured by the `java.util.concurrency` library. ASP's [8, 9] particularities are: distribution, *absence of shared memory*, and *transparent futures*, i.e., futures created *transparently* upon a remote method invocation.

This paper presented a *distributed evaluation* of the *functional* ς-calculus using *transparent* futures and active objects. It can also be seen as a study of the functional fragment of ASP. That is why we consider this calculus as complementary to the preceding ones. Futures can be passed around between different locations in a much transparent way; thanks to its functional nature and its type-system, this calculus ensures progress. Progress for active objects means that evaluation cannot lead to dead-locks. ASP_{fun} is called "functional" because objects are immutable. In ASP_{fun}, activities are organised in an actor-like manner. That is why we consider our language as a form of "functional actors" or "functional active objects". The main novelty of ASP_{fun} is that it is simple enough to allow for a mechanised specification and mechanised proofs of typing properties.

In comparison to the first presentation of the ASP_{fun}-calculus at the FOCLASA-workshop [40], the current paper better illustrates the semantics and further demonstrates the use of the functional update to personalise services (see Section 3). Moreover, this paper gives a precise description of the Isabelle/HOL formalisation comparing the two different approaches we have implemented for binders (see Section 6). In particular, the second implementation using the concept of locally nameless representation with its recent concept of cofinite induction is an independent contribution. We consider that the

major contribution of this paper is the mechanical formalisation, and the precise definition of formalisation tools that can be re-used to mechanically formalise other properties or languages.

Beyond the scope of this paper is a recent prototypical implementation of the ASP_{fun}-calculus in the concurrent language Erlang [41] intended for the practical exploration of privacy concerns in distributed systems. In a second conceptual paper we show that the functional update of ASP_{fun} can be used to implement a hiding mechanism for private data enabling the enforcement of an information flow property [42].

9. Conclusion

We presented a functional calculus for communicating objects and its type system. This work can be seen both as a distributed version of ς-calculus and as an investigation on the functional fragment of ASP. The particular impact of this work relies on the fact that it has been entirely formalised and proved in the Isabelle theorem prover. The functional nature of ASP_{fun} should make it influence directly stateless distributed systems like skeleton programming [43]. Our approach could be extended to study frameworks where most of the services are stateless, and the state-full operations can be isolated (access to a database), e.g., workflows and SOA. Our formalisation in a theorem prover should also impact other developments in the domain of semantics for distributed languages.

A calculus of communicating objects

The calculus is an extension of ς-calculus with only the minimal concepts for defining active objects and futures. Syntactically, the extension only requires one new primitive: *Active* creates a new activity from a term. The absence of side-effects and the guarantee of progress make the program easy to reason about and easy to parallelise. ASP_{fun} is distributed in the same sense as ASP: it enables parallel evaluation of activities while being oblivious about the concrete locations in which the execution of these activities takes place. The actual deployment is not part of the programming language and should be provided by an application deployer rather than by the application programmer.

Well-formed terms and absence of cycle

We proved that ASP_{fun} semantics is correct: no reference to non-existing activities or futures can be created by the reduction. Also, no cycle of future or activity references can be created. Thus, starting from an initial configuration, we always reach a well-formed configuration without cycle.

A type system for functional active objects

We extended the simple type system for ς-calculus: *Active* returns an object of the same type as its parameter; activities are typed like their active objects; and futures are typed like the request calculating their value. The type system ensures progress and preservation. Preservation states that the types are not changed during execution. Progress states that a program does not get stuck. In ASP_{fun}, this is due to the following facts:

- The type system plus the subject reduction property ensure that all method calls will access an existing method.
- Well-formedness ensures that all accessed activities and futures exist.
- Absence of cycles prevents cycles of mutually waiting synchronisations and infinite loops of replies.
- As partially evaluated futures can be replied, any chosen request can be reduced.
- All operations are defined for both local and active objects avoiding "syntactical" dead-locks like updating a method of an activity.
- Terms under *evaluation* contexts can be safely communicated between activities.

A formalisation in Isabelle/HOL. The formalisation adds the necessary quality assurance to a language development where rules and properties are intricate while the need for verification is as worthwhile as imperative. The formalisation is relatively long. It involves the definition of several constructs commonly encountered in the semantics for distributed languages (reduction contexts, references, typing, futures, . . .) that we think can be re-used in other developments, at least in the domain of semantics for distributed languages.

In practice we provided two formalisations: one uses de Bruijn indices, and the other uses locally nameless representation for representing variables. Those two approaches have been precisely compared.

The overall framework provides, to our mind, a good basis for the formal study of distributed object languages with futures.

Can we find a better progress property?

Let us analyse the limitations of the progress property.

First, though a reduction is possible, the reduced term can sometimes be identical to the original one. The absence of cycle ensures that such a situation can only occur in the local semantics. This is inherent to the ς-calculus and is out of the scope of this paper.

Second, the reduction can occur in any chosen request but not at any chosen place. Indeed, we can only ensure that points specified in restricted reduction contexts can be reduced. (See the definition of F in Section 6.2). This is a consequence of the fact that objects can only be sent between activities if they do not have free variables that otherwise would escape their binder. This restriction seems both natural and safe.

Future works. Additional properties could be proved on ASP_{fun}. First of all a proof of confluence for ASP_{fun} could be a good follow-up to this work. ASP_{fun} is also a good basis to study security or fault-tolerance concerns. More generally, we think that our mechanised formalisation is a good tool to prove properties on communication optimisations and protocols in the context of languages for distributed systems. We also aim at providing a formalisation of an imperative distributed object calculus, like ASP, and further mixing functional and imperative activities.

References

[1] M. Abadi, L. Cardelli, A Theory of Objects, Springer-Verlag, New York, 1996.
[2] B.C. Pierce, Types and Programming Languages, MIT Press, 2002.
[3] G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, MA, USA, 1986.
[4] G. Agha, I.A. Mason, S.F. Smith, C.L. Talcott, A foundation for actor computation, Journal of Functional Programming 7 (1997) 1–72.
[5] C. Hewitt, P. Bishop, R. Steiger, A universal modular actor formalism for artificial intelligence, in: IJCAI'73: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1973, pp. 235–245.
[6] G. Agha, An overview of actor languages, ACM SIGPLAN Notices 21 (1986) 58–67.
[7] D. Caromel, C. Delbé, A. di Costanzo, M. Leyton, ProActive: an integrated platform for programming and running applications on grids and P2P systems, Computational Methods in Science and Technology 12 (2006) 69–77.
[8] D. Caromel, L. Henrio, B. Serpette, Asynchronous and deterministic objects, in: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 2004, pp. 123–134.
[9] D. Caromel, L. Henrio, A Theory of Distributed Objects, Springer-Verlag, 2005.
[10] A.D. Gordon, P.D. Hankin, S.r.B. Lassen, Compilation and equivalence of imperative objects, in: Proceedings FST+TCS'97, in: LNCS, Springer-Verlag, 1997.
[11] E.B. Johnsen, O. Owe, I.C. Yu, Creol: a type-safe object-oriented model for distributed concurrent systems, Theoretical Computer Science 365 (2006) 23–66.
[12] T. Nipkow, L.C. Paulson, M. Wenzel, Isabelle/HOL—A Proof Assistant for Higher-Order Logic, in: LNCS, vol. 2283, Springer-Verlag, 2002.
[13] J. Niehren, J. Schwinghammer, G. Smolka, A concurrent lambda calculus with futures, Theoretical Computer Science 364 (2006) 338–356.
[14] L. Henrio, F. Kammüller, A mechanized model of the theory of objects, in: 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS, in: LNCS, Springer, 2007.
[15] B. Aydemir, A. Charguéraud, B.C. Pierce, R. Pollack, S. Weirich, Engineering formal metatheory, in: POPL'08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, 2008, pp. 3–15.
[16] B.E. Aydemir, A. Bohannon, N. Foster, B. Pierce, J. Vaughan, D. Vytiniotis, G. Washburn, S. Weirich, S. Zdancewic, M. Fairbairn, P. Sewell, The poplmark challenge, Web-site, 2008.
[17] N.G.D. Bruijn, Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem, Indagationes Mathematicae 34 (1972) 381–392.
[18] C. Urban, et al. Nominal methods group, Project funded by the German Research Foundation (DFG) within the Emmy-Noether Programme, 2006.
[19] A.M. Pitts, Nominal logic, a first order theory of names and binding, Information and Computation 186 (2003) 165–193.
[20] C. Roeckl, D. Hirschkoff, A fully adequate shallow embedding of the π-calculus in isabelle/hol with mechanized syntax analysis, Journal of Functional Programming 13 (2003) 415–451.
[21] A. Ciaffaglione, L. Liquori, M. Miculan, Reasoning about object-based calculi in (co)inductive type theory and the theory of contexts, JAR, Journal of Automated Reasoning 39 (2007) 1–47.
[22] F. Honsell, M. Miculan, I. Scagnetto, pi-calculus in (co)inductive-type theory, Theoretical Computer Science 253 (2001) 239–285.
[23] A. Schmitt, Safe Dynamic Binding in the Join Calculus, in: R. Baeza-Yates, U. Montanari, N. Santoro (Eds.), Proceedings of IFIP TCS 2002, in: IFIP, vol. 96, Kluwer, Montreal, Canada, 2002, pp. 563–575 (This is the original version that was accepted for publication, before the page cut requested for the final version. This version contains additional examples).
[24] A. Ricci, M. Viroli, G. Piancastelli, Simpa: an agent-oriented approach for programming concurrent applications on top of java, Science of Computer Programming 76 (2011) 37–62. Selected papers from the 6th International Workshop on the Foundations of Coordination Languages and Software Architectures—FOCLASA'07.
[25] L. Cardelli, A language with distributed scope, in: POPL'95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, 1995, pp. 286–297.
[26] S. Briais, U. Nestmann, Mobile objects "must" move safely, in: B. Jacobs, A. Rensink (Eds.), FMOODS, in: IFIP Conference Proceedings, vol. 209, Kluwer, 2002, pp. 129–146.
[27] A. Jeffrey, A distributed object calculus, in: ACM SIGPLAN Workshop Foundations of Object Oriented Languages.
[28] A.D. Gordon, P.D. Hankin, S.B. Lassen, Compilation and equivalence of imperative objects, in: Proceedings FST+TCS'97, in: LNCS, Springer-Verlag, 1997.
[29] R.H. Halstead Jr., Multilisp: a language for concurrent symbolic computation, ACM Transactions on Programming Languages and Systems (TOPLAS) 7 (1985) 501–538.
[30] A. Yonezawa, E. Shibayama, T. Takada, Y. Honda, Modelling and programming in an object-oriented concurrent language ABCL/1, in: A. Yonezawa, M. Tokoro (Eds.), Object-Oriented Concurrent Programming, MIT Press, Cambridge, MA, 1987, pp. 55–89.
[31] J. Niehren, D. Sabel, M. Schmidt-Schauß, J. Schwinghammer, Observational semantics for a concurrent lambda calculus with reference cells and futures, Electron. Notes Theor. Comput. Sci. 173 (April) (2007) 313–337.
[32] F.S. de Boer, D. Clarke, E.B. Johnsen, A complete guide to the future, in: R.D. Nicola (Ed.), ESOP, in: Lecture Notes in Computer Science, vol. 4421, Springer, 2007, pp. 316–330.
[33] E. Abrahám, I. Grabe, A. Grüner, M. Steffen, Behavioral interface description of an object-oriented language with futures and promises, Journal of Logic and Algebraic Programming 78 (2009) 491–518.
[34] J. Dedecker, T.V. Cutsem, S. Mostinckx, T. D'Hondt, W.D. Meuter, Ambient-oriented programming in ambienttalk, in: D. Thomas (Ed.), ECOOP, in: LNCS, vol. 4067, Springer, 2006, pp. 230–254.
[35] A. Cansado, L. Henrio, E. Madelaine, Transparent first-class futures and distributed component, in: International Workshop on Formal Aspects of Component Software, FACS'08, in: Electronic Notes in Theoretical Computer Science (ENTCS), Malaga, 2008.
[36] J. Bengtson, J. Parrow, Formalising the pi-calculus using nominal logic, in: Proc. of the 10th International Conference on Foundations of Software Science and Computation Structures, FOSSACS, in: LNCS vol. 4423, 2007, pp. 63–77.
[37] J. Bengtson, M. Johansson, J. Parrow, B. Victor, Psi-calculi: mobile processes, nominal data, and logic, in: LICS'09: Proceedings of the 2009 24th Annual IEEE Symposium on Logic In Computer Science, IEEE Computer Society, Washington, DC, USA, 2009, pp. 39–48.
[38] T. Ridge, Operational reasoning for concurrent caml programs and weak memory models, in: K. Schneider, J. Brandt (Eds.), Theorem Proving for Higher Order Logics, TPHOLs'07, in: LNCS, vol. 4732, Springer, 2007.
[39] C. Flanagan, M. Felleisen, The semantics of future and an application, Journal of Functional Programming 9 (1999) 1–31.
[40] L. Henrio, F. Kammüller, Functional active objects: typing and formalisation, in: 8th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA'09, in: ENTCS, vol. 255, Elsevier, 2009, pp. 83–101. Satellite to ICALP'09.
[41] A. Fleck, F. Kammüller, Implementing privacy with erlang active objects, in: 5th International Conference on Internet Monitoring and Protection, ICIMP'10, IEEE, 2010.
[42] F. Kammüller, Using functional active objects to enforce privacy, in: 5th Conf. on Network Architectures and Information Systems Security, SAR-SSI, 2010.
[43] M. Cole, Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming, Parallel Computing 30 (2004) 389–406.

2.5 Summary and conclusion

In this chapter we described our development on distributed object calculi, and more specifically on the formalisation and the implementation of the active object programming model. This work led both to classical proofs on paper, but also to mechanised proofs with an increased reliability. These contributions have been the opportunity to formalise several crucial concepts of the active object programming model: objects, first class futures, request/replies mechanism, typing, remote method invocations. The coexistence of those notions provide a rich calculus expressing quite precisely the programs that can be written in active object programming, like in the ProActive library.

We reviewed some of the practical impact of those theoretical developments, but the influence of this work is probably wider. First, we will see in Section 4.2 how this model can be extended to express multi-active objects. Also, one of the crucial consequence of this formalisation is the design of future update strategies and their implementation in the ProActive library as shown in [45]. However, this work on future updates was omitted in this chapter, because it will be presented in Section 3.4 in the context of the GCM component model where it has been mechanically formalised. Indeed, in order to prove runtime properties on the applications programmed in GCM, we chose a semantics for GCM components *à la* active objects, we will see that this semantics has some similarities with ASP, while being more general. This more general context gave us a good opportunity to formally specify and prove the properties of one of the future update protocols that we implemented in ProActive.

Next chapter will first present the GCM components, and then several works we realised for formally specifying and verifying the properties of both the component model, and the applications programmed in GCM/ProActive.

Chapter 3

Composing Distributed Applications

Object-orientation provides a programming model that is limited in terms of re-usability and dynamic adaptation of the programmed applications. Indeed, nicely written objects specify very well the interfaces they provide to the other objects, but on the other hand they do not specify precisely what other objects they use. More globally, object-oriented programming would benefit from better architectural informations, this would provide a higher-level view of the program structure, which would ease the program design and analysis, but also its dynamic adaptation.

In this chapter, I will review our work related to composition of applications, mainly in the domain of component models, but also with behavioural skeletons. Those highly structured composition models are a good opportunity for the use of formal methods, as shown by the existence of several conferences dedicated to formal methods for component models (FMCO, FACS, ...)

3.1 The Grid Component Model: GCM

Component models provide a structured programming paradigm, and ensure a very good re-usability of programs. Indeed in component applications, dependencies are defined together with provided functionalities by the means of provided/required ports; this improves the program specification and thus its re-usability. Some component models and their implementations additionally keep a trace at runtime of the component structure and their dependencies. Knowing how components are composed and being able to modify this composition at runtime provides great adaptation capabilities: the application can be adapted, e.g. to evolution in the execution environment, by changing some of the components taking part in the composition or changing the dependencies between the involved components. We call *reconfiguration* the actions consisting in changing at runtime the component structure, by adding or removing components in the system, or by changing the way components are bound together.

In distributed systems, reconfiguration takes even more importance as the structure of components can also be used at runtime to discover services and use the most efficient service available. Also, as some distributed components will naturally migrate from one location to another, they will change their execution environment. Again, reconfiguration is quite often necessary in order to adapt components to different execution environments. Several effective distributed component models have been specified, developed, and implemented in the last years [CF05, Obj06, BBB+07] ensuring different kinds of properties to their users. We also took part in the design and implementation of one of them, the Grid Component model [9] described in the paper included in Section 3.1. This paper also provides a comparison of the main distributed component models; I will thus not present a section dedicated to generic purpose related works on component models here. GCM has been proposed in the CoreGrid Network of Excellence, it is an extension of the Fractal component model [BCL+04, BCS04] to better address large-scale distributed computing. GCM builds above Fractal and thus inherits its hierarchical structure, the enforcement of separation between functional and non-functional concerns, its extensibility, and the separation between interfaces and implementation. The main extensions provided by GCM are the following:

- GCM supports *collective communications*: one-to-many, many-to-one, but also many-to-many. For example GCM defines multicast interfaces allowing a single port to be connected to several, with the possibility that one message emitted from a multicast port is broadcasted to all the ports connected to it. Such multicast interfaces must be attached a policy defining the way message arguments are distributed among the many destination ports.
- GCM also comes with a support for autonomic aspects and better separation between functional and

non-functional concerns: more precisely, in GCM non-functional concerns can also be defined as a component assembly.

Those extensions required to extend the architecture description language (ADL[1]), and also the API of Fractal to take into account the aspects mentioned above. The definition of the GCM has been standardised as a set of 4 ETSI standards dealing with deployment aspects, but also specifying GCM architecture description language, and the API for manipulating components at runtime.

Figure 3.1 shows a GCM assembly and introduces most of the terminology used to describe GCM components and their composition. Among the notions presented in the figure only multicast interfaces and gathercast interfaces are specific to GCM. We will see in Section 3.2 that we also refined the structure of the membrane of GCM components in order to better structure and adapt the component management aspects.

A reference implementation for GCM

ProActive/GCM is a reference implementation of the GCM component model that has been implemented during the GridComp European project. It is based on the ProActive Java library and relies on the notion of active objects. It is important to note that each component corresponds at runtime to an active object and consequently each component can easily be deployed on a separate JVM and can be migrated. Of course, this implementation relies on design and implementation choices relatively to the purely structural definition provided by the model. Section 3.4 will provide a possible semantics for GCM components that is more general than the reference implementation but still allowed us to prove properties on ProActive/GCM.

One of the main advantage of using active objects to implement components is their adaptation to distribution. Indeed, by nature active objects provide a natural way to provide loosely coupled components. By loose coupled component, we mean components responsible for their own state and evaluation, and only communicating via asynchronous communications. Asynchronous communications increase and automate parallelism; and absence of sharing eases the design of concurrent systems. Additionally, loose coupling reduces the impact of latency, and limits the interleaving between components. Finally, independent components also ease the autonomic management of component systems, enabling systems to be more dynamic, more scalable and easily adaptable to different execution contexts. That is why we think that a distributed component system should rely on loosely coupled components communicating via asynchronous communications, and not sharing any memory. That is thus the reason why we think active objects are particularly adapted to implement a distributed component model.

We also provided a framework for interconnecting ProActive/GCM and CCA components in [4]. This study allowed us to show that GCM model fits quite well with CCA notions, and more generally to study how to make different component models interoperable.

Discussion: granularity of the component model

A general issue when designing a component model is the foreseen granularity of the components: "what is the size of a component?" In the case of a hierarchical component model like GCM, this question can be refined into "what is the size of a primitive component?" When addressing distribution aspects of a component model, the same question arises again, but becomes more complex: "what is the relation between the unit of composition (the primitive component) and the unit of distribution?".

Fractal does not impose any granularity for the components, but the existence of composite bindings and some of the features of the model suggest a rather fine grained implementation: a primitive component should contain a small number of objects. Like Fractal, the GCM does not enforce precisely any granularity of the component systems. However, in order to allow GCM primitive components to be the unit of distribution for a GCM implementation, we consider that GCM components would probably have a coarser granularity than Fractal ones. Overall, the GCM has been conceived with a granularity that is somehow in middle between small grain Fractal components and very coarse grain component models, like CCM where a component is of a size comparable to an application. Somehow, GCM has been conceived thinking of components of the size of an MPI process, though it can be used in a much finer or coarser grain way.

This difference of granularity between Fractal and the GCM partially explains why some of the features that could be implemented by a small Fractal component and are highly used in a Grid setting have been defined as first class citizens in the GCM. For example, multicast interfaces could be express in Fractal by binding components that perform the broadcast, but such components would be too small to be used as the unit of distribution. Also the structure of non-functional aspects that we proposed for the GCM (see Section 3.2) is somehow influenced by the foreseen component granularity.

[1]the ADL is a domain specific language dedicated to the definition of components: from an ADL component description, a new component system can be instantiated.

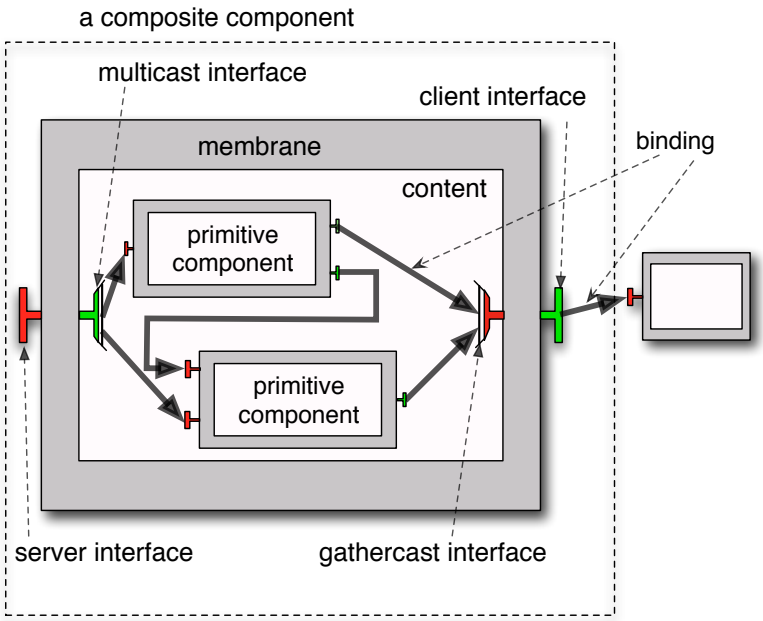


Figure 3.1: A typical GCM assembly

In ProActive/GCM the primitive components (and the composite ones too) have this intermediate size: they contain an activity, i.e. an active object, its dependencies, request queue, and thread.

Paper from Annals of Telecommunication, 2009

This paper presents the GCM component model, it is one of the strong results of the CoreGrid Network of Excellence. Several partners contributed to the definition of the component model, and to the following article.

Ann. Telecommun. (2009) 64:5–24
DOI 10.1007/s12243-008-0068-8

GCM: a grid extension to Fractal for autonomous distributed components

Françoise Baude · Denis Caromel · Cédric Dalmasso · Marco Danelutto · Vladimir Getov · Ludovic Henrio · Christian Pérez

Received: 30 July 2007 / Accepted: 16 July 2008 / Published online: 17 December 2008

Abstract This article presents an extension of the Fractal component model targeted at programming applications to be run on computing grids: the grid component model (GCM). First, to address the problem of deployment of components on the grid, deployment strategies have been defined. Then, as grid applications often result from the composition of a lot of parallel (sometimes identical) components, composition mechanisms to support collective communications on a set of components are introduced. Finally, because of the constantly evolving environment and requirements for grid applications, the GCM defines a set of features intended to support component autonomicity. All these aspects are developed in this paper with the challenging objective to ease the programming of grid applications, while allowing GCM components to also be the unit of deployment and management.

Keywords Distributed components · Autonomous components · Adaptable components · Collective communications · Grid component model

F. Baude · D. Caromel · C. Dalmasso · L. Henrio (✉)
INRIA Sophia-Antipolis, I3S, Univ. de Nice Sophia-Antipolis, CNRS, INRIA, Sophia Antipolis, France
e-mail: ludovic.henrio@inria.fr

F. Baude
e-mail: francoise.baude@inria.fr

D. Caromel
e-mail: denis.caromel@inria.fr

M. Danelutto
University of Pisa, Pisa, Italy
e-mail: marcod@di.unipi.it

V. Getov
Harrow School of Computer Science, University of Westminster, London, UK
e-mail: V.S.Getov@westminster.ac.uk

C. Pérez
INRIA/IRISA, Rennes, France
e-mail: christian.perez@inria.fr

1 Introduction

Grid computing raises a lot of challenges for programming models because it consists in programming and running applications running over *large-scale heterogeneous* resources that evolve *dynamically*. The grid component model (GCM) addresses the characteristic challenges in terms of programmability, interoperability, code reuse, and efficiency. Programming large-scale distributed systems as grids can be considered as a matter of distributed services deployment and further integration. In this paper, we advocate the idea that a hierarchical and distributed software component-based approach is an effective solution to this.

1.1 Objectives

The research challenges dealt with by the GCM are, thus, the support at the application level for *heterogeneity*, *large-scale distribution*, and *dynamic management and adaptivity* by means of a component model to provide a high *programmability of grid applications*.

Programmability deals with the expressive power of the language mechanisms that are offered to the programmers and what is the burden for them to

effectively use those mechanisms. A short overview of current proposed grid frameworks makes us believe that it is in the programmability dimension that resides the greatest divergence between those solutions. Schematically, these solutions range

- From low-level message-passing [for example, message passing interface (MPI)] remote procedure call (RPC)- or remote method invocation (RMI)-based traditional parallel and distributed programming models—simply ported to tackle grid issues—by which the program itself dictates and orchestrates the parallelism and distribution of computing and communicating entities [24]
- To solutions in which the orchestration or choreography of the set of parallel and distributed entities is guided from the extern of these entities, not necessarily in a centralised manner by using for example workflow languages and programming [23]

We think that these two categories are not exclusive because the spectrum of applications that could benefit from running on grids is not closed. The purpose of the GCM is to reconcile those two extreme points of view: a component approach allows both explicit communications between distributed entities like in MPI and high-level management of the distribution of those entities and their interactions, like in workflows. GCM mainly focuses on the programmability of end-user grid applications, but is also suited to program tools and middleware in the computing grid context: those can be designed and implemented as GCM components featuring specific services.

So, we aim at proposing a solid and adequate parallel and distributed programming model laying the foundation for building any form of grid application. Its qualities must be those of *expressiveness*, *extensibility*, solid *theoretical foundation* and suitability for *optimisation* and competitive implementations. In light of this, we selected the Fractal component model as the starting point for offering a versatile yet structured and tractable grid programming model.

1.2 Approach and contribution

A general issue when designing a component model is the advised granularity of the components: "what is the size of a component?" This issue is often overridden in the presentation of a component model, but is crucial to understand the decisions taken in such a design. In the case of a hierarchical component model like Fractal, this question becomes "what is the size of a primitive component?", or "what is the unit of composition?" Fractal does not impose any granularity for the components, but the concept of binding component [10] and some of the features of the model suggest a fine-grained implementation: a primitive component is assimilated to one or a few objects.

The granularity of the component model is, to our mind, a crucial aspect because it influences the expressive power and the overhead of the component architecture: a fine-grain system increases the ability to compose components but generally entails additional cost to manage a larger number of entities and to make them interact.

When addressing distribution aspects of a component model, the same question arises again, but becomes more complex: "what is the relation between the unit of composition (the primitive component) and the unit of distribution?" Like Fractal, the GCM does not enforce precisely any granularity of the components. However, in order to allow GCM primitive components to be also the unit of distribution for a GCM implementation, we consider that GCM component implementations would probably have a coarser granularity than Fractal ones. This difference in the advocated component granularity partially explains why some of the highly used features in a grid setting as collective communication mechanisms have been defined as first-class citizens in the GCM. For example, multicast communication could be expressed in Fractal by relying on binding components, but such components would be too small to be used as the unit of distribution. In brief, in GCM, *each component is subject to distribution*.

Compared to other component models, the GCM has been conceived around a component granularity that is somehow in middle between small grain Fractal components and very coarse grain ones, like those suggested by CORBA component model (CCM) where a component is of a size comparable to a full-fledged application. Somehow, GCM has been conceived thinking of components of the size of a process (i.e., one or a few threads per primitive component), though it can be used in a much finer or coarser grain way.

To address the challenges expressed above, the GCM is a specification taking the following approach. *Distribution* concerns are specified at the composition level by specific entries in the Architecture Description Language (ADL) relying either on a controlled or on an automatic mapping between computing resources of the infrastructure and primitive components. Many-to-one and one-to-many communications are key mechanisms for optimising communications in a *large-scale* environment; they are also key *programming constructs* for distributing computations and synchronising their results. This paper also studies the effective combination of one-to-many and many-to-one interfaces: the

MxN problem. Finally, *heterogeneous dynamic* large infrastructures require the adaptation of the application and its management to be totally distributed and, consequently, preferably autonomous. For this, GCM extends Fractal with controllers as components, and with the definition of interfaces for autonomicity, to enable the autonomous control to be designed as a component-based system.

The point of view we adopt here is close to Fractal: we are not tied to any programming language; however, like in Fractal, we reuse the terminology of object-oriented programming. Components are thought of as autonomous service entities exchanging messages or requests according to precisely defined ports (named interfaces in Fractal).

1.3 Foundations

The GCM has been defined by the CoreGRID European Network of Excellence gathering researchers in the area of grid and peer-to-peer technologies. It relies on the following aspects inherited from existing works:

- *Fractal as the basis for the component architecture:* We summarise the characteristics we benefit from Fractal in Section 2.1.
- Communication semantics: GCM components should allow for any kind of communication semantics (e.g., streaming, file transfer, event-based) either synchronous or asynchronous. Of course, for dealing with high latency, asynchronous communications will probably be preferred by most GCM frameworks.

1.4 Outline

This paper starts with an overview of existing component models that can be used in the area of grid computing, in Section 2. Among the central features of the GCM, this article will focus on the most innovative ones:

- Support for deployment: distributed components need to be deployed over various heterogeneous systems. The GCM defines deployment primitives for this. Deployment aspects will be developed in Section 3.
- Support for one-to-many, many-to-one and many-to-many communications: often, grid applications consist of a lot of similar components that can be addressed as a group, and that can communicate together in a very structured way. The GCM also intends to provide high-level primitives for a better design and implementation of such *collective communications* which will be detailed in Section 4.
- Support for non-functional adaptivity and autonomic computation: the grid is an highly evolving environment, and grid applications must be able to adapt to those changing runtime conditions. For this reason, we propose to allow for both reconfiguration of the component control aspects, and autonomic computation support. Adaptivity and autonomicity in the GCM will be presented in Section 5.

2 Other distributed component models

This section reviews the main component models; it first briefly presents what peculiar and interesting features the Fractal abstract component model provides, consequently arguing why we selected it as the basis for the GCM. Next, we review some other software component models that are targeted at the programming of distributed applications, or even of middleware, taking into account constraints raised by distribution.

2.1 Fractal

Fractal [10] is a general component model which is intended to implement, deploy and manage (i.e. monitor, control and dynamically configure) complex software systems, including in particular operating systems and middleware. Among Fractal's peculiar features, below are those that motivated us to select it as the basis for the GCM.

- Hierarchy (composite components can contain subcomponents), to have a uniform view of applications at various levels of abstraction
- Introspection capabilities, to monitor and control the execution of a running system
- Reconfiguration capabilities, to dynamically configure a system

To allow programmers to tune the control of reflective features of components to the requirements of their applications, Fractal is defined as an extensible system.

Fractal comes with a formal specification. It can be instantiated in different languages such as Java and C. In addition, the Fractal specification is a multi-level specification, where, depending on the level, some of the specified features are optional. That means that the model allows for a continuum of reflective features or *levels of control*, ranging from no control (black-boxes, standard objects) to full-fledged introspection and intercession capabilities (including, e.g., access and

manipulation of component contents, control over components life-cycle and behaviour, etc.).

Fractal already has several implementations in different languages. The GCM is not tied to Fractal's reference implementation (Julia), which is not targeted at distributed architectures. Dream is a library built using Julia Fractal components targeting distribution, but specifically aimed at building message-oriented middleware, and not grid applications or even grid middleware as we intend to do.

To sum up, it is because of its extensible and hierarchical nature that Fractal has been chosen as the basis for the definition of the GCM. Fractal does not constrain the way(s) the GCM can be implemented, but it provides a basis for its formal specification, allowing us to focus only on the grid-specific features. Eventually, platforms implementing the GCM should constitute suitable grid programming and execution environments. ProActive offers one such implementation [5].

2.2 Distribution-aware component models

This section focuses on some of the main distributed component models and on what is missing in these models in order to fully support a structured approach to grid programming, underlying the necessity for an innovative and new component model.

Let us first focus on two commonly known models for a component-oriented approach [38] to distributed computing: the *common component architecture (CCA)* [3, 12] and the *CCM* [33].

- *CCA* has been defined by a group of researchers from laboratories and academic institutions committed to specifying standard component architectures for high performance computing. The basic definition in CCA states that a component "is a software object, meant to interact with other components, encapsulating certain functionality or a set of functionalities. A component has a clearly defined interface and conforms to a prescribed behaviour common to all components within an architecture." Currently, the CCA forum maintains a web-site gathering documents, projects and other CCA-related work (www.cca-forum.org) including the definition of a CCA-specific format of component interfaces (Babel/SRPC Interface Description Language) and framework implementations (Ccaffeine, Xcat)
- *CCM* is a component model defined by the Object Management Group, an open membership for-profit consortium that produces and maintains computer industry specifications such as CORBA, UML and XMI. The CCM specifications include a Component Implementation Definition Language; the semantics of the CCM; a Component Implementation Framework, which defines the programming model for constructing component implementations, and a container programming model. Important work has been performed to turn the CCM in a grid component model, like GridCCM [18].

In recent years, the US-based CCA initiative brought together a number of efforts in component-related research projects, with the aim of developing an interoperable GCM and extensions for parallelism and distribution [9]. However, the CCA model is non-hierarchical, thereby making it difficult to handle the distributed and possibly large set of components forming a grid application [22] in a structured way. Indeed, hierarchical organisation of a compound application can prove very useful in getting scalable solutions for management operations pertaining to monitoring, life-cycle, reconfiguration, physical mapping on grid resources, load-balancing, etc. Unfortunately, the CCA model is rather poor with regards to managing components at runtime. It means a CCA component per se does not have to expose standard interfaces dedicated to non-functional aspects as it is the case for Fractal, and consequently, GCM components. This makes it hard to realise certain features, for instance, dynamic reconfiguration based on observed performance or failures. However, some implementations of the model, like, e.g. XCAT, can provide some additional components (like an application manager) dedicated to manage the non-functional aspects of a CCA-based application. However, this has to be considered as an additional and optional feature, not defined by the component model, so it prevents interoperability between CCA components running onto different platforms. Consequently, we think that the GCM is a richer programming model than CCA and allow the effective design and management of distributed applications at a grid scale.

CCM presents the same limitations than CCA with the exception that CCM handles quite well the heterogeneity of resources. In CCM, the ADL is able to deal with distributed resources but it is outside the scope of the specifications to describe how such a description has been generated. However, this task requires a high level of knowledge of the application structure, as well as the resource properties. This approach is not satisfactory for grids where resources are provided dynamically. Hence, while CCM has some very interesting features for grids—in particular because CCM has

been designed for distributed applications—it appears as a model where distribution is too coupled to the resources for grid applications.

Even if CCA and CCM components can fit into a distributed infrastructure, they are not designed as being distributed per se, and possibly parallel entities to be mapped onto a set of grid resources, nor having the capability to self-adapt to the changing context. By contrast, the Enterprise Grid Alliance effort [40] is an attempt to derive a common model adopting grid technologies for enhancing the enterprise and business applications. The model, which is aligned with industry-strength requirements, strongly relies on component technology along with necessary associations with component-specific attributes, dependencies, constraints, service-level agreements, service-level objectives and configuration information. One of the key features that the EGA reference model suggests is the life-cycle management of components which could be governed by policies and other management aspects. The level of this specification, however, is very coarse-grain, focusing on system integration support rather than providing an abstract model and specification for grid programming, which is the main goal of GCM.

Most of grid-oriented component models use components to wrap complete, possibly parallel, applications. This is sufficient to build new grid-wide HPC applications, e.g. multi-disciplinary ones, by composition of a few separate software modules. This also means that a such components must not be considered as the unit of distribution, but as a coarse-grain unit wrapping a full-fledged software exposed as a grid service, to be composed with a few others. On the contrary, a GCM primitive component is a well delimited unit of distribution and management at the scale of the grid, and a GCM composite component is a suitable abstraction to hierarchically handle at once any sort of distributed and parallel composition, including ones that may be formed of a very large set of software units spread all over the grid and running in parallel. Of course, this does not prevent a primitive GCM component to itself wrap a legacy, e.g. MPI, parallel application, but in this case, it is clear that the resulting set of parallel processes, probably co-located on the same cluster of machines, is under the management responsibility of the primitive component itself.

In terms of grid middleware, there have been a few platforms such as ICENI [21] that enable users to build grid applications out of software components. On several platforms, applications running on the grid are interconnected by some kind of collective binding mechanisms, notably in Xcat and ICENI. However, most of the component-oriented platforms that we are aware of support components at application level only without any componentisation at the runtime environment level. Instead, the design of ICENI follows the classical service-oriented architecture (SOA) approach [20]. Obviously, a side-effect of such SOA-based approaches is the strong importance given to interoperability through, for example, the WSDL-based exportation of the component interfaces. Interoperability is also recognised as a key aspect of the GCM, in order to be capable of loosely connecting external applications based upon any kind of technology to a GCM-based one [19].

One of the exceptions among the existing component-oriented platforms is the GRIDKIT project [15]. In GRIDKIT, the middleware itself is designed as components, derived from OpenCOM. In addition, the GRIDKIT team identified the need for support of multiple complex communication paradigms, non-functional (horizontal) services, autonomicity and reconfiguration. The GCM addresses these concerns but at a different level by providing corresponding support as an integral part of the component model itself so that GCM-based grid middleware and applications can benefit from those features. Thus, an interesting perspective could be to adopt the GCM in future versions of the GRIDKIT middleware in order to benefit from these advanced features both at the component model level and at the middleware one. GCM has already proved to be efficient for conceiving a grid runtime support inside the CoreGRID project [13].

Compared to related works, GCM originality lies in its adopted model, at the level of components themselves, for deployment, collective communications, adaptivity and autonomicity.

3 Deploying components

GCM applications are primarily designed to be run on grids, that is to say on a complex and dynamic distributed system. Hence, a major question is how to express the mapping of the components on the resources. Grid environments usually provide job schedulers whose task is to compute when and where to launch an application. However, job schedulers are system-level entities: as such, they are only able to deal with simple jobs such as sequential jobs and MPI-like jobs for the most advanced. It is far behind the current state of the art of the schedulers to deal with complex structures such as a hierarchy of distributed components. Hopefully, grid environments also provide information services. Hence, it is possible to imagine a dedicated

deployment service that can take care of selecting adequate resources for an application.

Component models usually enable the description of the initial structure of an application thanks to some ADL. However, for distributed platforms, ADL files include the name of the resources. It is well suited for a *particular* deployment of an application on a known set of resources. However, it is inappropriate to have to change these files each time the application is deployed on a different platform, whereas the application architecture and implementation did not change. Therefore, the explicit mentioning of the name of resources inside an ADL is not well suited to describe a grid application.

The GCM provides two strategies, a simple and a more advanced one, to deal with this issue. The first strategy is based on the virtual node concept. It aims at enabling a logical grouping of the components on a virtual infrastructure. The second strategy aims at not presenting any infrastructure concept to the application. The remainder of this section presents them.

3.1 Controlled mapping through virtual nodes

A first strategy for supporting deployment is to rely on virtual nodes. Virtual nodes are abstractions allowing a clear separation between design infrastructure and physical infrastructure. This concept already exists both in the standard Fractal ADL and the ProActive middleware. Virtual nodes can be used in the ADL and they can abstract away names, but also creation and connection protocols. Consequently, applications remain independent from connection protocols and physical infrastructure. A virtual node contains one or more nodes. A node represents a location where a component can be created and executed. This can be a single physical machine (a host), or, in the case of a multi-processor/multi-core machine, a single processor or a single core within a machine.

The *virtual-node* element, in ADL files, offers distributed deployment information. To better specify the deployment constraints on a component, the standard Fractal ADL has been extended. The *cardinality* attribute has been added to the *virtual-node* element. In addition to this element, the GCM adds the possibility to export and compose virtual nodes in the *exportedVirtualNodes* element. We will describe how these elements can be used to control the component/virtual-node mapping in ADL files.

The syntax is similar to the Fractal ADL, features specific to the GCM are:

- Virtual nodes have a cardinality: either *single* or *multiple*. *Single* means the virtual node in the deployment descriptor should contain one node; *multiple* means the virtual node in the deployment descriptor should contain more than one node. For example, the following element in a component definition indicates that we want to create the component in the virtual node *client-node* which contains one node.

```
<virtual-node name="client-node"
  cardinality="single"/>
```

- Virtual nodes can be exported and composed. Export and compose allow, respectively, to rename and merge virtual nodes. This extends re-usability of existing components. When exported, a virtual node can take part in the composition of other exported virtual nodes. The following composition code creates a new virtual node named *client-node*, composed from two virtual nodes, *client1* and *client2*, defined in components *c1* and *c2*.

```
<exportedVirtualNodes>
<exportedVirtualNode
  name="client-node">
<composedFrom>
 <composingVirtualNode component="c1"
   name="client1"/>
 <composingVirtualNode component="c2"
   name="client2"/>
</composedFrom>
</exportedVirtualNode>
</exportedVirtualNodes>
```

Then, mapping from virtual nodes to the infrastructure is defined in separate files, called deployment descriptors. Those files describe the real infrastructure and the way to acquire resources; we do not detail the format of deployment descriptors here, see [5]. Components are deployed on a node included in the virtual node that is specified in their definition; it has to appear in the deployment descriptor unless this virtual node is exported.

A component will be instantiated on the node associated to the virtual node given in its ADL (modulo the renaming entailed by exportation). In case several components use the same virtual node with a multiple cardinality, we do not specify on which node we create each component.

3.2 Automatic mapping to the infrastructure

Deployment descriptors provide a mean for expert programmers/deployers to control how a particular application is deployed on a set of resources. Another

abstraction step is needed to further decouple an application from the resources. The underlying idea is to let a programmer specify its component assembly within a model without any resource concept, i.e. without any knowledge on the physical architecture. Then, an automatic deployment process is needed to derive a mapping of the components to the available resources. This section reviews the needed steps to achieve such an automatic mapping. It shows that most steps are already provided by current grid environments and details what is still needed.

Overview. Starting from a description of an application and a user objective function, the deployment process is responsible for automatically performing all the steps needed to start the execution of the application on a set of selected resources. These steps are illustrated in Fig. 1. The logical order of the activities is fixed (submission, discovery, planning, enactment, execution). Some steps have to be re-executed when the application configuration is changed at run-time. Moreover, the steps in the gray box, that interact closely, can be iterated until a suitable set of resources is found.

The following describes the activities involved in the deployment of an application. This process only takes as input a file describing the components of the application, their interactions, and the characteristics of the required resource.

Application description. The application may be described in a variant of Fractal ADL, which contains several kinds of data: the description of the component types and their implementations, as well as information to guide the mapping of the application onto resources. It may consist of the *resource constraints*, characteristics that resources (computational, storage, network) must possess to execute the application; the *execution platform constraints*, software (libraries, middleware systems) that must be installed to satisfy application dependencies; the *placement policies*, restrictions or hints for the placement of subsets of application processes (e.g. co-location, location within a specific network domain, or network performance requirements), and the *resource ranking*, an objective function provided by the user, stating the optimisation goal of application mapping. Resource ranking is exploited to select the best resource, or set of them, among those satisfying the given requirements for a single application process. Resource constraints can be expressed as *unitary requirements*, that must be respected by a single module or resource (e.g. CPU rata), and as *aggregate requirements*, that a set of resources or a module group must respect at the same time (e.g. all the resources on the same LAN, access to a shared file system); some placement policies are implicitly aggregate requirements. As of today, there is no standard format for describing the constraints, the placement policies, or the resource ranking.

Resource discovery. This activity finds the resources compatible with the execution of the application. Resources satisfying unitary requirements can be discovered, interacting with grid information services [16]. Then, the information needed to perform resource selection (that considers also aggregate requirements) must be collected for each suitable resource found. Existing grid technologies are quite satisfactory with respect to this point, but co-allocation support in grid scheduler is still quite uncommon.

Deployment planning. When information about available resources is collected, the proper resources that will host the execution of the application must be selected, and the different parts of each component

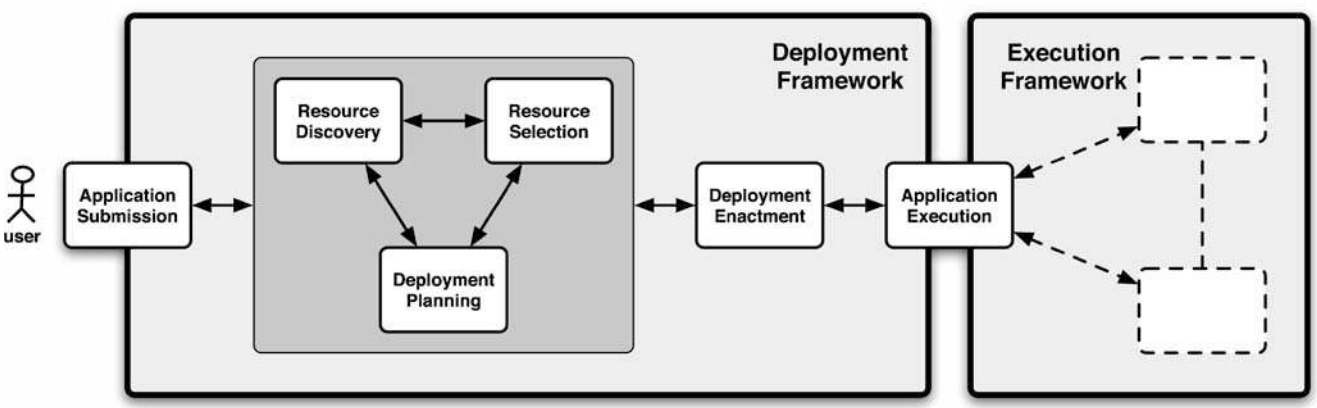


Fig. 1 Deployment process for automatic mapping

have to be mapped on some of the selected resources. This activity also implies satisfying all the aggregate requirements within the application. Thus, repeated interaction with the resource discovery mechanisms may be needed to find the best set of resources, also exploiting dynamic information.

At this point, the user objective function must be evaluated against the characteristics and available services of the resources (expressed in the resource description schema). When appropriate, a resource ranking is established to find a suitable solution.

An abstract deployment plan is computed by gathering the deployment schema of all application components. The abstract plan is then mapped onto the resources, and turned into a concrete plan, identifying all the services and protocols that will be exploited in the next phase on each resource, in order to set up and start the runtime environment of the application. This step is probably the most challenging one as it requires advanced algorithms (heuristics) to compute a plan, as the problem is generally NP-hard.

Deployment enactment. The concrete deployment plan developed in the previous phase is submitted to the execution framework, which is in charge of the execution of the tasks needed to deploy the application. This service must ensure a correct execution of the deployment tasks while respecting the precedences described in the deployment plan. At the end of this phase, the execution environment of the application is ready to start its actual execution. This step is nowadays quite well mastered.

Application execution. The deployment process for adaptive grid applications does not finish when the application is started. Several activities have to be performed while the application is active. The whole application life-cycle must be managed, in order to support new resource requests for application adaptation, to schedule a restart if a failure is detected, and to release resources when the normal termination is reached. These monitoring and controlling activities are mediated by the autonomic part of the components, which performs some dynamic deployment action.

3.3 Discussion

This section has presented two deployment strategies for a grid application: one strongly driven by the user and a much more automatic one. The first deployment strategy provides a mechanism to capture some topological constraints of the mapping of the component hierarchy to the resources. The application can map its elements to the virtual nodes independently of the real resource names: the application is portable. Moreover, the mapping of the virtual nodes to the physical nodes appears at the level of current grid schedulers.

The second deployment strategy aims at providing an automatic mapping of the application on the resources. It requires to extend ADL with constraints and placement policies, as well as some more advanced schedulers. This strategy should lead to a real autonomicity of components. It seems a prerequisite for adaptivity and autonomicity as discussed in Section 5.

Both strategies have been validated through prototypes, the first in ProActive/GCM, the second in ADAGE [28] and GEA [17]. They run on top of various environments, from cluster-like environments (ssh, batch, etc) to grid environments such as Globus.

4 Supporting *M* to *N* communications

To meet the specific requirements and conditions of grid computing for multiway communications, *multicast* and *gathercast* interfaces give the possibility to *manage a group of interfaces as a single entity*, and *expose* the collective nature of a given interface. Multicast interfaces allow to distribute method invocation and their parameters to a group of destinations, whereas, symmetrically, gathercast allow to synchronise a set of method invocations toward the same destination. Solutions to the problem of data distribution have been proposed within PaCO++/GridCCM [18]; these solutions can be seen as complementary to the basic distribution policy specified in this section.

4.1 Collective interfaces

In pure Fractal, collective bindings could be performed using composite bindings,[1] which would accept one input and a collection of output, or a collection of inputs and one output. Collective interfaces allow GCM components to perform operations collectively on a set of components without relying on intermediate components. The objective is to simplify the design of component-based applications and ensure type compatibility in a direct manner. Of course, the model still allows for the use of explicit binding components, in case of specific requirements for inter-component communications, for instance when binding interfaces of incompatible types. Though the alternative relying on composite binding could have a similar behaviour to the

[1] In Fractal, a composite binding is a communication path composed of a set of primitive bindings and binding components.

collective interfaces, we consider collective interfaces better adapted to the GCM as explained below.

First, we think that, for design purposes, the collective nature of the connection should be attached to the definition of the component, not to its binding. This also allows control of the collective behaviour at the level of the component containing the interface, not in an external component.

Second, suppose collective interfaces would be implemented by additional components, possibly belonging to composite bindings. As in GCM, the component is the unit of distribution, the question of the localisation of the additional components implementing the collective behaviour arises. The best choice would probably be to allocate the binding at the same place as one of the functional components they bind, depending on the nature of the interface; in the GCM, this choice is made clear by the design of the collective interfaces. Moreover, if such binding components would be distributed, they would need to be instrumented with remote communication capabilities which would make them bigger and less efficient than collective interfaces.

Here again, the granularity of the envisioned component model plays a crucial role: making the component the unit of distribution and mobility requires primitive components to encapsulate code for managing those aspects. This makes such components inadequate for encoding basic features like collective interfaces. Indeed, it would be inefficient to attach to interfaces, or to composite binding implementing collective communication, the code necessary to manage local threads and mobility, for example.

Preliminary remark. In the sequel, we use the term *list* to mean *ordered set of elements of the same type* (modulo sub-typing). This notion is not necessarily linked to the type List in the chosen implementation language; it can be implemented via lists, collections, arrays, typed groups, etc. To be more precise, we use `List<A>` to mean *list of elements of type A*.

The notion of collective interface is not linked to any communication semantics: communication between components can be implemented for example by message passing, remote procedure calls, or streaming. However, we present the particular case of remote method invocations in the remaining of this section because of its richer implications on typing of interfaces and on the component composition. Experiments on the implementation of collective communications for components interacting by asynchronous remote method invocations have been conducted over the ProActive middleware, and proved to be quite efficient and convenient to program distributed applications [7]. However, the notions of multicast and gathercast interfaces are clearly also adapted to other communication semantics, the consequence on type compatibility between interfaces can be inferred from the case presented in this section.

4.2 Multicast interfaces: 1 to *N* communications

Multicast interfaces provide abstractions for one-to-many communication. First, we will define this kind of interface, next we will detail the needed update for interface signature and at the end of this section we will address the distribution of parameters and invocations.

Multicast interfaces can either be used internally to a component to dispatch an invocation received by the components to several of its sub-entities or externally to dispatch invocations emitted by the component to several clients.

4.2.1 Definitions

A multicast interface transforms a single invocation into a list of invocations.

A single invocation on a multicast interface is transformed into a set of invocations. These invocations are forwarded to a set of connected server interfaces (Fig. 2). The semantics concerning the propagation of the invocation and the distribution of parameters are customisable. The result of an invocation on a multicast interface—if there is a result—is a list of results. Invocations on the connected server interfaces may occur in parallel, which is one of the main reasons for defining this kind of interface: it enables *parallel invocations*.

For example, in a composite component, a *multicast internal client interface* transforms each single invocation into a set of invocations that are forwarded to bound server interfaces of inner components.

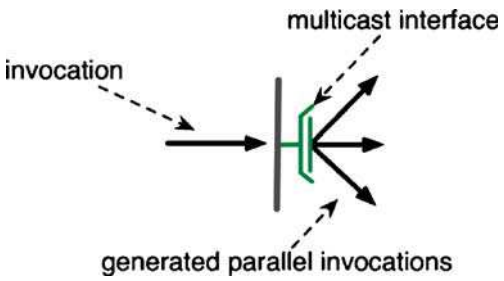


Fig. 2 Multicast interfaces

To support multicast interfaces, we need to extend the type system of Fractal by adding the `String getFcItfCardinality ()` method to the InterfaceType interface. The interface type is extended for dealing with new cardinalities: the `getFcItfCardinality()` method returns a string element, which is convenient when dealing with more than two kinds of cardinalities. The type factory method `createFcItfType` is extended with the `String cardinality` parameter.

The BindingController also needs an extension to support only removing of some bound interface: `void unbindFcMulticast(String name, Object itf)`. This specification does not make any assumption about the communication paradigm used to implement the multicast invocations [31, 35].

4.2.2 Automatic data distribution

The signature of multicast interface can be different from the single interfaces it is bound to. We detail this typing issue and its relation with data distribution in this section and the following. This section focuses on a simple view where the parameters that are to be distributed are lists, and thus, the distribution can be performed automatically: lists are distributed element-wise, and other elements are kept as non-splittable. Consequently, we provide in this section two basic distribution policies for parameters: *broadcast* consists in sending the same parameters to each of the connected server interfaces and *scatter* is only available for lists; it strips the parameter so that the bound components will work on different data.

Returned result. For each method invoked and returning a result of type `T`, a multicast invocation returns an aggregation of the results: a `list<T>`.

For instance, consider the signature of a server interface:

```
public interface I {
  public void foo();
  public A bar();
}
```

A multicast interface may be connected to the server interface with the above signature only if its signature is the following (recall that `List<A>` can be any type storing a collection of elements of type `A`):

```
public interface J {
  public void foo();
  public List<A> bar();
}
```

In that case, we say that `I` is the type of the multicast interface *on the server side*, i.e. the type of the server interfaces the multicast can be bound to, and `J` is the type *on the client side*, i.e. the type of the mutlicast interface itself.

Where to define multicast interfaces? Collective interfaces are defined in the ADL; two new *cardinalities*—multicast and gathercast—has been added to Fractal specification. The cardinality of an interface can be single, collection, *multicast*, or *gathercast*.

Where to specify parameters distribution? The ADL files are not the right place to specify the parameter distribution because distribution is too dependent on the implementation. Thus, the best place to specify distribution policy is inside the interface definition, e.g. using annotations in the case of Java. In addition, we propose to specify and modify the distribution policy in a dedicated controller, named CollectiveInterfacesController. The policy for managing the interface is specified as a construction parameter of the CollectiveInterfacesController. This policy is implementation-specific, and a different policy may be specified for each collective interface of the component.

How to specify the distribution of parameters into a set of invocations? Remember we focus on two possible data distribution basic policies: broadcast and scatter. In the broadcast mode, all parameters are sent without transformation to each receiver. In the scatter mode, however, many configurations are possible, depending upon the number of parameters that are lists and the number of members of these lists. In the automatic distribution policies, parameters to be scattered are of type `list<T>` on the client side, and of type `T` on the server side. Parameters to be broadcasted must be of the same type on the client and on the server side. A general solution in the case of a single parameter to be distributed is to perform as many invocations as there are elements in the list.

When several parameters are to be distributed, there is not a single general solution. We propose to define, as part of the distribution policy, the multiset[2] F of the combination of parameters, where each element $f_j \in F$ is such that, $f_j \in [1..k_1] \times [1..k_2] \times .. \times [1..k_n]$, where n is the number of formal parameters of the invoked method which are to be scattered, and k_i, $1 \leq i \leq n$ the number of values for each scattered actual parameter. This multiset allows the expression of all the pos-

[2] A multiset is a set where the number of occurrences of each element matters.

sible distributions of scattered parameters, including Cartesian product and one-to-one association. The cardinal of F also gives the number of invocations which are generated, and which depends on the configuration of the distribution of the parameters.

As an illustrative example, the Cartesian product of n parameters is expressed as follows:

$$\{(i_1, \ldots, i_n) | \forall l \in [1..n],\ i_l \in [1..k_l]\}$$

One-to-one association is expressed as follows when $k_1 = k_2 = \ldots k_n$:

$$\{(i, \ldots, i) | i \in [1..k]\}$$

The number of occurrences in the multiset is useful when several identical calls have to be produced, e.g. to duplicate the computation in order to tolerate the failure of some of the clients.

To summarise, for automatic data distribution in multicast interfaces:

- If the return type of the function is `T` on the server side, it must be `list<T>` on the client side.
- For each parameter, if the type of the parameter is `list<T>` on the client side and `T` on the server side, then this parameter is scattered, the combination of scatter modes is defined by an external function; else, if the type of the parameter is `T` on both client and server side, the parameter is broadcasted.

4.2.3 Defining complex distribution policies

This section releases the strict constraints on typing for multicast interfaces given in the preceding section by relying on user-defined distribution or aggregation functions and involving constraints on the resulting typing of multicast interfaces. In the general case, distribution policies may depend on the number of bound components, but for simplicity, we will not explicitly use this parameter in this section. The constraints specified in this section should be used when type checking the bindings between components involved in the multicast interface.

Aggregating results. The constraint of having lists as results for multicast invocations may be relaxed by providing an aggregation mechanism that performs a reduction. Until now, we have defined a basic aggregation function, which is concatenation, but any function can be used for aggregating results, leading to the following typing constraint (relate to Fig. 3 for name convention):

> If the returned type of the multicast interface is of type `S`, on the left side (i.e. if `S` is the type of the client interface), and of type `T`, on the right side (i.e. if `T` is the type of the server interfaces the multicast is connected to), then the multicast interface should be attached an aggregation function of type:
>
> `List<T>` → `S`

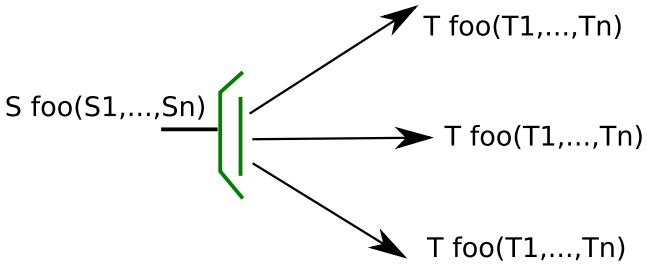


Fig. 3 General case of type conversion through a multicast interface

Section 4.2.2 discussed the simplest case where `S = List<T>` and the aggregation function is the identity.

Depending on the operations performed by this last function, it might be necessary to synchronise the achievement of the different calls dispatched by the multicast operation. For example, it is impossible to return the maximum of all results before waiting for all of them to be arrived at the multicast interface.

Here are a few examples illustrating different possible aggregations of results for a multicast interface:

- The result is the sum of the results computed for each of the n calls distributed to the destination components:

  > n integers are summed into one integer; the signature of the aggregation function is: `List<int>` → `int`. The multicast interface has the return type: `int`.

- The multicast interface returns the result given by the majority of calls.

  > n results are reduced to a single one plus an occurrence count. The signature of the aggregation function becomes: `List<T>` → (`T`, `int`). The multicast interface returns a `(T,int)`.

- n pieces of an array are gathered into one single array to be returned.

  > The signature of the aggregation function is: `List<Array<A>>` → `Array<A>`. The multicast interface has the return type: `Array<A>`.

Distributing parameters. This generalisation could also be applied to the distribution of invocation parameters. In Section 4.2.2, if an argument of a call toward a multicast interface is of type `S`, then the type of the argument received on one of the bound interfaces is either `S` (argument broadcasted as it is) or `T` if `S` is of the form `List<T>`. More generally, we can have any transformation of argument type through the multicast interface:

> If the arguments of the multicast interface (i.e. the parameters of the call) are of type `Si`, $1 \leq i \leq n$ on the client side (left part of Fig. 3), and of type `Ti`, $1 \leq i \leq n$ on the server side (right part of Fig. 3), then the multicast interface should be attached a distribution function returning a list of parameter sets to be sent, its type should be:

`S1..Sn` → `List <(T1, .., Tn)>`

We provide a few examples illustrating different possible type conversions for arguments of a multicast interface (the last two being the ones already presented in Section 4.2.2):

- Blocks of an array to be dispatched differently depending on the number of destination components in parallel (N):

  > One call with parameter of type `Array<A>` becomes N calls with parameter of type `Array<A>` containing pieces of the original array. Distribution function is of type: `Array<A>` → `List< Array<A> >`.

- Scatter:

  > One call with parameter of type `List<A>` becomes *length*(`List < A >`) calls with parameter of type `A`. Distribution function is of type: `List<A>` → `List<A>`.

- Broadcast: same invocation replicated to N components in parallel:

  > One call with parameter of type `A` becomes N calls with parameter of type `A`. Distribution function is of type: `A` → `List<A>`.

4.2.4 Distribution of invocations

Once the distribution of the parameters is determined, the invocations that will be forwarded are known. A new question arises: how are these invocations dispatched to the connected server interfaces? This is determined by a function, which, knowing the number of server interfaces bound to the multicast interface and the list of invocations to be performed, describes the dispatch of the invocations to those interfaces.

Consider the common case where the invocations can be distributed regardless of which component will process the invocation. Then, a given component can receive several invocations; it is also possible to select only some of the bound components to participate in the multicast. In addition, this framework allows us to express naturally the case where each of the connected interfaces has to receive exactly one invocation in a deterministic way.

4.3 Gathercast interfaces: M to 1 communications

Gathercast interfaces provide abstractions for many-to-one communications. Gathercast and multicast interface definitions and behaviours are symmetrical [4]. Gathercast interfaces can either be used internally to a component to gather the results of several computations performed by several sub-entities of the component or externally to gather and synchronise several invocations made toward the component.

4.3.1 Definition

> A gathercast interface transforms a set of invocations into a single invocation.

Gathercast interfaces gather invocations from multiple source components (Fig. 4). A gathercast interface coordinates incoming invocations before continuing the invocation flow: it may define synchronisation barriers and may gather incoming data. Return values are redistributed to the invoking components.

For example, in a composite component, a *gathercast internal server interface* transforms a set of invocations coming from client interfaces of inner components into

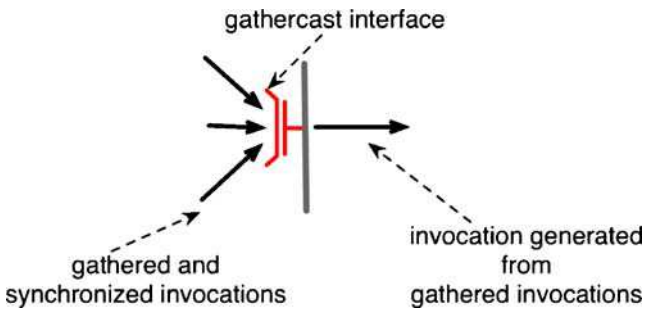


Fig. 4 Gathercast interface

a single invocation from the component to the external world.

For synchronisation purposes, gathering operations require knowledge of the participants (i.e. the clients of the gathercast interface) in the collective communication. As a consequence, *bindings to gathercast interfaces are bidirectional links*; in other words: a gathercast interface is aware of which interfaces are bound to it; this should be realised by the binding mechanism.

4.3.2 Synchronisation operations

Gathercast interfaces provide one type of synchronisation operation, namely message-based synchronisation capabilities: the message flow can be blocked upon user-defined message-based conditions. *Synchronisation barriers* can be set on specified invocations, for instance, the gathercast interface may wait—with a possible timeout—for all its clients to perform a given invocation on it before forwarding the invocations. It is also possible to define more complex or specific message-based synchronisations, based on the content and number of the messages, or based on temporal conditions, and it is possible to combine these different kinds of synchronisations.

4.3.3 Automatic data aggregation and redistribution

This section details the parameter gathering and result redistribution that can be performed automatically by a gathercast interface.

Gathering parameters. The gathercast interface aggregates parameters from method invocations. Thus, the parameters of an invocation coming from a gathercast interface are actually lists of parameters. If, on the client side, invocations are on the form `void foo(T)`, then the generated invocations necessarily have the type `void foo(list<T>)` on the server side. In other words, if the client interfaces connected to the gathercast are of type `void foo(T)`, then the gathercast (server) interface itself is of type `void foo(list<T>)`.

Redistributing results. The distribution of results for gathercast interfaces is symmetrical with the distribution of parameters for multicast interfaces, and it raises the question: where and how to specify the redistribution?

The place where the redistribution of results is specified is similar to the case of multicast interfaces: the redistribution is configured through metadata information for the gathercast interface. This could, for example, be specified through annotations or be inferred from the type of interface.

The way redistribution is performed is also similar to multicast interfaces. It also necessitates a comparison between the client interface type and the gathered interface type. If the return type of the invoked method in the client interfaces is of type `T` and the return type of the bound server interface is `List<T>`, then results can be scattered: each component participating in the gather operation receives a single result (provided the result is a list of the same length as the number of participants). Otherwise, results should be broadcasted to all the invokers and the return type must be *identical* on the client and the server side. A redistribution function can also be defined as part of the distribution policy of the gathercast interface, it is configurable through its collective interface controller.

4.3.4 Defining complex distribution policies

The symmetric of multicast interfaces general specification can be defined for redistribution of results for gathercast interfaces and aggregation of parameters of calls toward a gathercast interface. For example, the constraint of having lists as parameters for gathercast invocations may be relaxed by providing a reduction function and verifying at connection type the type compatibility between the reduction function and the bound interfaces.

4.4 The MxN problem

The support of parallel components raises the concern of efficient communications between them. This section focuses on the MxN problem, i.e., efficient communication and exchange of data between two parallel programs, consisting, respectively, of M and N entities. In the GCM, such a pattern can be straightforwardly realised by binding a parallel component with a gathercast internal server interface to a component with a multicast internal client interface. However, efficient communications between two parallel components requires direct binding so as to support direct communications between the involved inner components on both sides; this mechanism is called MxN communications. End users expect to have MxN communications to provide performance scalability with the parallelism degree. Whereas Sections 4.4.1 to 4.4.3 focus on data distribution and establishment of bindings, Section 4.4.4 discusses synchronisation of such parallel components.

4.4.1 Principles

A naive and not optimised solution for MxN coupling is shown in Fig. 5. The respective output of the M inner components is gathered by the gathercast interface; then, this result is sent as it is to the multicast interface; finally, the message is scattered to the N inner components connected to the multicast interface, so data are redistributed by the multicast interface.

Obviously, this naive solution creates a bottleneck both in the gathercast and in the multicast interfaces. Efficient communications require some forms of direct bindings between the inner components according to the redistribution pattern, like that shown by the arrow of Fig. 5 drawn between an arbitrarily chosen pair of inner components from both sides. In the general case, implementing such direct bindings requires to replace the couple gathercast + multicast interfaces by M multicast interfaces plus N gathercast interfaces. Each inner component on the left-hand side is responsible for sending its own data to all the concerned components; on the right-hand side, each inner component is responsible for gathering the messages it receives and performing its piece of the global synchronisation. This creation of additional collective interfaces avoids the bottleneck occurring in the single gathercast or multicast interface. We show below how such an optimisation can be implemented in the case of a specific but classic scenario.

4.4.2 Example of a direct binding

This section illustrates a way to ensure the M-by-N optimisation in a particular case that is relatively frequent. It both illustrates the possibility for multicast and gathercast interfaces to enable optimised communications and it shows the necessity for highly parameterisable collective interface. Indeed, optimised communication patterns are simply performed by connecting additional gathercast and multicast interfaces, parameterised depending on the data distribution and the topology.

Suppose two composites CM and CN, composed of M and N components, respectively, must exchange data by blocks, as shown in Fig. 6. For the sake of simplicity, we suppose that each of the M inner components send data of size d and each of the N components must receive data of size d' ($M \times d = N \times d'$).

We denote M_i, $0 \leq i < M$ the inner components of CM, and symmetrically, N_j, $0 \leq j < N$ the inner components of CN. Consequently, considering the data to be exchanged as an interval of size $d \times M = d' \times N$, each component exchanges the data in the following range:

M_i produces $[d \times i, d \times (i+1)[$

N_j consumes $[d' \times j, d' \times (j+1)[$

Bindings. Each of the M_i components will have its client interface turned into a multicast client interface with the same signature (called IM_i). Symmetrically, each of the N_j components will have its server interface turned into a gathercast server interface (called IN_j). The direct bindings that must occur should ensure the direct communication between components having to transmit data. Components are connected if there is an intersection between the range of data sent by one and the range that must be received by the other. Bindings are formally defined as follows: IM_i is to be bound to IN_j iff $\exists l \in [d \times i, d \times (i+1)[$ s.t. $l \in [d' \times j, d' \times (j+1)[$. In a more constructive manner, one can specify the indices of the client components:

IM_i must be bound to all the IN_j

s.t. $\left(\left(d/d'\right) \times i\right) - 1 < j < \left(d/d'\right) \times (i+1)$

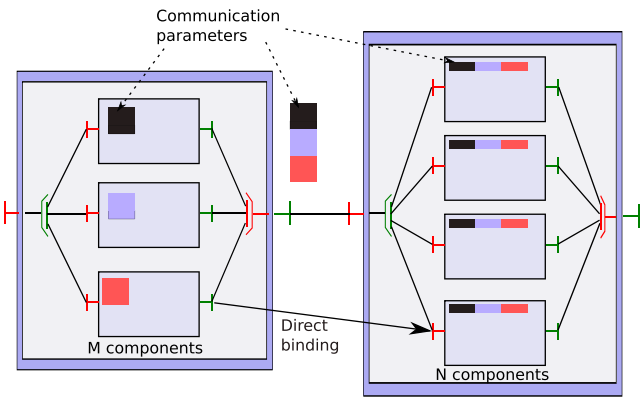


Fig. 5 Gathercast to multicast

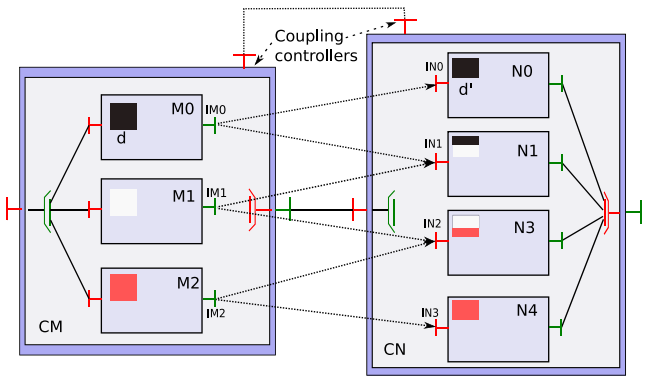


Fig. 6 Communications resulting from an MxN direct binding

Communications. We define now what elements are directly sent from one inner component of CM to the inner components of CN. Each M_i has to send to N_j the elements in the global range:

$$[d \times i, d \times (i+1)[\cap [d' \times j, d' \times (j+1)[$$

which is necessarily non-empty if M_i is connected to N_j. This set logically represents the intersection between the produced data and the consumed one.

4.4.3 Using controllers to set up MxN bindings

This section defines a possible configuration phase for coupling two parallel components, in a MxN manner in a very general setting. It relies on the existence of controllers (called coupling controllers, that could be associated to collective controllers) both at the level of parallel components (Fig. 6) and at the level of the inner ones.

When binding two parallel components, both coupling controllers exchange information about their respective collective interfaces (cardinality, data distribution pattern, size and type of data…) and the reference of internal components attached to this collective port. Relevant information is then passed to the coupling controllers of the inner components so as to configure them correctly. Once configured, the direct communication (data redistribution and synchronisation) is straightforward: every inner component is aware of the components it communicates with, as well as data distribution information.

This controller-based approach is suitable to implement the redistribution described in the example above (Section 4.4.2). In this case, controllers just have to exchange the cardinality of their respective interfaces (M and N), and the references to the inner components. Controllers create and configure interfaces in the inner components accordingly to the formulas of Section 4.4.2.

4.4.4 Synchronisation issues

Additionally to the data redistribution, the gathercast–multicast composition plays a synchronisation role. Indeed, in Fig. 5, thanks to the gathercast interface, the computation can only start on the right-hand side when all the inner components on the left-hand side have sent their output. The introduction of the gathercast interfaces on the right-hand side (Fig. 6) moves this synchronisation behaviour to the N inner components. If the MxN direct communication pattern is such that each of the N processes receives data from all the M processes, then the behaviour of the system with direct bindings is equivalent to the original one. Else performing the same synchronisation as the not optimised version requires all the clients to send a message to all the gathercast interfaces, some of them being only synchronisation signals. However, global synchronisation is not required by all the applications, and in this case, more optimisation is possible: if only data redistribution is important, then only bindings for transmitting data must be carried out.

4.5 Collective interfaces and hierarchy

Let us conclude this section by a study on the influence of hierarchy on the notion of collective interfaces. Basically, the existence of composite components entails the existence of internal interfaces, allowing collective interfaces to act internally to a component and collectively on the content of a composite component.

Except for this, the existence of collective interfaces is not related to hierarchy at the level of the component model. However, at the applicative level, composition of hierarchy and collective operation allows the programmer to easily design complex component systems, like hierarchical master–slave for example. To summarise, the impact of collective interfaces associated with hierarchy is that any component of a system can be considered as, and transformed into, a parallel component in a very natural manner.

5 Adaptivity and autonomicity

To provide dynamic behaviour of the component control, we propose to make it possible to consider a controller as a sub-component, which can then be plugged or unplugged dynamically. As in [37], we promote the idea to adopt a component-oriented approach to express the control part of a component. On the contrary, in the Julia Fractal implementation, for instance, control part is expressed in an object-oriented fashion. Adaptivity of a component in an open and large-scale system as a computing grid can be a complex task to orchestrate and implement. So, relying on a component-oriented approach for the control part can ease its design and implementation, thus increasing the component adaptation ability.

Additionally, autonomicity is the ability for a component to adapt to situations, without relying on the outside. Several levels of autonomicity can be implemented by an autonomic system of components. The

GCM defines four autonomic aspects, and it gives a precise interface for each of these four aspects. These interfaces are non-functional and exposed by each component.

5.1 A refinement of Fractal for non-functional adaptivity

In component models as Fractal, or Accord [29], for example, adaptation mechanisms are triggered by the control, also named non-functional (NF), part of the components. This NF part, called the *membrane* in Fractal and GCM, is composed of *controllers* that implement NF concerns. Interactions with execution environments may require complex relationships between controllers. Examples of use-cases include changing communication protocols, updating security policies, or taking into account new runtime environments in case of (mobile) components running on mobile devices interconnected to the core computing grid.

In this section, we focus on the adaptability of the *membrane*. Adaptability means that evolutions of the execution environment have to be detected and acted upon; this process may imply interactions with the environment and with other components. Our purpose in the GCM definition with respect to adaptivity is not to provide adaptive algorithms but to offer the support for implementing them as part of the control part of the components, and even more, the possibility to plug dynamically different control strategies, i.e. to adapt the control part itself to the changing context.

In the GCM, we want to provide tools for adapting controllers. This means that these tools have to manage (re)configuration of controllers inside the membrane and the interactions of the membrane with membranes of other components. For this, we provide a model and an implementation, applying a component-oriented approach for both the application (functional) and the control (NF) levels. Applying a component-oriented approach to the non-functional aspects allows them to feature structure, hierarchy and encapsulation. The same method has been followed or advocated in [26, 32, 36, 37].

The solution adopted in the GCM is to allow, like [32, 37], the design of the membrane as a set of components that can be reconfigured [14]. Baude et al. [8] goes more into details and describes a structure for the composition of the membrane, its relationships with the content of the component and an API for manipulating it. Note that it does not seem reasonable to implement, like in AOKell, the membrane as a composite GCM component: due to the distributed nature of GCM (implying that a GCM component would, in general, involve a much higher overhead than a Fractal one), having to cross an additional composite component boundary to switch into or from the control part of a GCM component would involve a sensible overhead. So, we came to the idea of having the component-based system defining the non-functional features be totally diluted in the membrane of the component containing the functional code (called the *host* component in this case).

In order to be able to compose non-functional aspects, the GCM requires the NF interfaces to share the same specification as the functional ones: role, cardinality and contingency. For example, in comparison to Fractal, the GCM adds *client non-functional interfaces* to allow for the composition of non-functional aspects, reconfigurations and component re-assembling at the non-functional level. To summarise, the GCM is provided with the possibility to implement as components (part of) the membrane and, thus, benefit from strong component structure and reconfiguration capabilities.

A small example. Figure 7 illustrates the structure of the membrane using components. In the figure, two non-functional components are assembled in the component's membrane, but more importantly, the membrane can rely on client non-functional interfaces, both internal to allow connection to inner components, and external to allow connections with other components, dedicated to the management and monitoring of the application, for example. This both gives a structure to the non-functional concerns of the component *Comp* and allows the reconfiguration at the non-functional level, in order to adapt it to the changes in the environment.

Life-cycle issue. This new structure for controllers raises the following question: "What is the life-cycle of a component used inside the membrane?" In Fractal, invocation on controller interfaces *must* be enabled

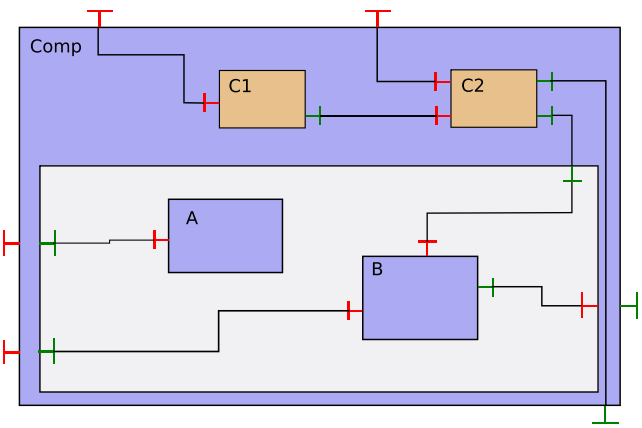


Fig. 7 A composite with pluggable controllers

when a component is (functionally) *stopped*, and obviously, for changing the bindings of a component, this component must be *stopped*. In other words, a controller of a Fractal component is an entity that does not follow the classical life-cycle of a component, in particular, it can never enter a stop state. Consequently, GCM components cannot adhere to this specification; otherwise, their membrane could not be reconfigured.

The solution we propose consists in a more complex life-cycle for component controllers, allowing to separate partially the life-cycle states of the membrane and of the content. When a component is *functionally stopped* (which corresponds to the stopped state of the Fractal specification), invocation on controller interfaces are enabled and the content of the component can be reconfigured, whereas, when a component is *stopped*, only the controllers necessary for configuration are still active (mainly binding, content, and life-cycle controllers), and the other components in the membrane can be reconfigured. Thanks to the new component architecture defined, two kinds of reconfiguration are possible: reconfiguration of the functional inner component system, following the idea of hierarchical autonomic decision paths [1], and reconfiguration of the membrane itself when the adaptation is done along the NF properties of the host component.

5.2 Autonomic components

GCM supports all the mechanisms needed to implement autonomic components, as stated in the previous sections. In particular, the availability of membrane components, as well as the possibility to hierarchically compose new components from simpler, existing ones, can both be exploited to support different autonomic management features. More in detail, two distinct kinds of autonomic management are considered as *first-class citizens* in GCM:

- The one taking care of simply adapting the single component to the changing conditions of the component"external" environment; a notable example could be the one wrapping component interaction mechanisms in such a way that the interactions can be performed using *secure* communication mechanisms rather than insecure ones. This is the kind of self-configuring, adaptation autonomic behaviour expected from components aware of the fact they can live in secure or insecure frameworks.
- The one taking care of adapting the component internal behaviour to match external, non-functional requirements; a notable example could be the one adjusting the parallelism degree of a parallel composite component in such a way that a non-functional performance contract is kept satisfied during the execution of the composite component activities. This is again a kind of self-configuring and self-healing autonomic behaviour [27].

In order to provide autonomic component management, GCM programming model supplies two different facilities to the GCM user/programmer. On the one hand, GCM provides all those mechanisms needed to implement the autonomic managers. These mechanisms include the ability to implement membrane as components discussed in the previous section. However, they also include some lower-level mechanisms that can be used to "sense" both the component execution environment and some component internal features of interest for the autonomic management. As an example, mechanisms are provided to "introspect" features of the component related to its implementation. An autonomic manager supposed to control component performance must be enabled to test component response/service time, for instance. Therefore, some mechanisms are supplied within GCM that allow to probe such values. The set of mechanisms of this type provided to the autonomic manager programmers define, *de facto*, the kind of managers implementable in the GCM framework.

On the other hand, a methodology aimed at supporting autonomic component managers is provided, such that programmers of the manager components do not have to rewrite from scratch each one of the managers included in the components. Such a methodology can be basically stated as a set of guidelines and rules to be adopted when programming the autonomic component managers, of course. In order to be more effective, GCM also provides the autonomic manager programmers with a set of autonomic manager skeletons/design patterns that can be easily customised properly supplying the skeleton/design pattern parameters. These manager patterns capitalise on the experiences coming from the software engineering autonomic management research track, as well as all the experience acquired in the algorithmic skeletons and design pattern areas.

Following this approach, the GCM autonomic manager programmer can pick up one of two ways:

- He/she can customise a (composition of) autonomic manager skeleton(s) by providing proper parameters, and therefore, he can get very rapidly a complete manager whose behaviour (*modulo* the provided parameters) has already been tested, debugged and proven correct.

– In case the provided manager skeletons do not fit user requirements, he/she can go through the complete (re-)writing of a new autonomic manager, exploiting the provided API to access component internal features, as well as component environment features, and implementing his own autonomic policies.

In [2], it has already been outlined how autonomic manager skeletons (called "behavioural skeletons" to distinguish them from the classical "algorithmical skeletons" that are only related to the functional computation features) can be designed in GCM that can autonomically take care of the performance issues of notable parallel component compositions. Behavioural skeletons abstract common autonomic manager features, leaving the autonomic manager programmer the possibility to specialise the skeleton to implement the particular autonomic manager he has in mind. More in detail, behavioural skeletons aim to describe recurring patterns of component assemblies that can be (either statically or dynamically) equipped with correct and effective management strategies with respect to a given management goal. Behavioural skeletons help the application designer to (1) design component assemblies that can be effectively reused and (2) cope with management complexity by providing a component with an explicit context with respect to top-down design (i.e. component nesting).

Parallelism management can be designed and parameterised in the same way as classical, functional algorithmical skeletons abstract common features of parallelism exploitation patterns, leaving the programmers the possibility to model their own parallel computations by providing suitable skeleton parameters, including, in the general case, sequential code parameters completely specifying the actual computation to be performed.

Technically, because the membrane components are still under development, the behavioural skeletons discussed in [2] have been currently implemented as inner components of composite components. An implementation of behavioural skeletons based on membrane components can now be considered; it will exploit several useful membrane component features, such as the ability to implement client interfaces.

6 Summary and conclusion

In this paper, we presented the key features of a grid-oriented component model: the GCM. Relying on Fractal as the basic component structure, the GCM defines a set of features which are necessary to turn the Fractal model into a grid compliant one. GCM is more than a small extension of Fractal: it provides a new set of component composition paradigm through multicast and gathercast, addresses the issue of distributed component deployment and provides support for autonomous components. Overall, the GCM can be considered as a component model on its own. Conformance to the GCM can be summarised as follow:

– Support for deployment of components, either relying on deployment descriptors, or featuring an automatic mapping to the infrastructure
– Possibility to collectively compose and target sets of components: existence of multicast and gathercast interfaces
– Support for autonomic components: possibility to design membranes as component systems, to compose (i.e. bind together) non-functional features possibly distributed over several components and support for self-adaptation of the components to both evolving environments and evolving requirements

GCM has been used in different settings showing the effectiveness of this approach. First, a prototype of the ProActive implementation of the GCM has already been used to build and deploy over a grid a numerical computation application for electromagnetism [34]. Moreover, in the context of the *common component modeling example (CoCoME)*, GCM components have been modeled and specified, and a prototype implementation has been realised [11]. The CoCoME consists of a point-of-sale example featuring distribution, asynchronism and collective communications.

Interoperability between GCM and other standards or component models has been demonstrated, first through effective interactions between CCA and GCM components [30], and second by the possibility to expose component interfaces as web services [19].

The CoreGRID Institute on Grid Systems, Tools, and Environments has been working on a methodology for the design and implementation of a generic component-based grid platform [13] collating the innovative efforts of a number of partners from several European countries. The research activities and results show that the GCM can be used to implement a grid runtime environment. GCM has been proved to be adequate to implement development, deployment, monitoring and steering tools. As a result, the grid integrated development environment and the component-based integrated toolkit, based on the GCM, provide a framework that enables rapid development of grid applications and the transparent use of available resources at

runtime. These achievement show the adequacy of the GCM for developing not only grid applications but also a grid runtime and environment.

The experiences mentioned above allow us to evaluate the GCM relatively to the objectives given in the introduction. First, the hierarchical aspect is really a key feature to better address scalability in practice. Second, expressiveness of the collective interfaces is generally adequate as showed by the programming of SPMD-like interactions, but the specification of distribution policies is still to be improved. Indeed, allowing the GCM implementation to reach the expressiveness of the distribution policies described in Sections 4.2.3 and 4.3.4, and thus allowing real program to express simply complex distributions, is still a real challenge. Finally, we also showed the adequacy of the GCM to express autonomic adaptations [2, 6]. Thus, we estimate that the GCM greatly improves expressiveness, scalability and adaptivity of grid applications, even if the model and its implementation are still to be improved. One difficulty that has been encountered several times is the management of distributed asynchronous components and, in particular, the problem of stopping such components; however, some solutions have been recently suggested for this problem [25, 39].

References

1. Aldinucci M, Bertolli C, Campa S, Coppola M, Vanneschi M, Zoccolo C (2006) Autonomic grid components: the GCM proposal and self-optimising ASSIST components. In: Joint workshop on HPC grid programming environments and components and component and framework technology in high-performance and scientific computing at HPDC'15, Paris, June 2006
2. Aldinucci M, Campa S, Danelutto M, Dazzi P, Kilpatrick P, Laforenza D, Tonellotto N (2008) Behavioural skeletons for component autonomic management on grids. In: Danelutto M, Frangopoulou P, Getov V (eds) Making grids work. CoreGRID. ISBN 978-0-387-78447-2
3. Armstrong R, Gannon D, Geist A, Keahey K, Kohn S, McInnes L, Parker S, Smolinski B (1999) Toward a common component architecture for high-performance scientific computing. In: Proceedings of the 1999 conference on high performance distributed computing, Amsterdam, 12–14 April 1999
4. Badrinath B, Sudame P (2000) Gathercast: the design and implementation of a programmable aggregation mechanism for the Internet. In: Proceedings of IEEE international conference on computer communications and networks (ICCCN), Las Vegas, 16–18 October 2000
5. Baduel L, Baude F, Caromel D, Contes A, Huet F, Morel M, Quilici R (2006) Grid computing: software environments and tools, chap. Programming, deploying, composing, for the grid. Springer, Heidelberg
6. Baude F, Henrio L, Naoumenko P (2007) A component platform for experimenting with autonomic composition. In: First international conference on autonomic computing and communication systems (Autonomics 2007). ACM Digital Library (invited paper)
7. Baude F, Caromel D, Henrio L, Morel M (2007) Collective interfaces for distributed components. In: CCGrid 2007: IEEE international symposium on cluster computing and the grid. ISBN 0-7695-2833-3, pp 599–610
8. Baude F, Caromel D, Henrio L, Naoumenko P (2007) A flexible model and implementation of component controllers. In: CoreGRID workshop on grid programming model, grid and P2P systems architecture, grid systems, tools and environments, Crete, 12–13 June 2007
9. Bertrand F, Bramley R, Damevski KB, Kohl JA, Bernholdt DE, Larson JW, Sussman A (2005) Data redistribution and remote method invocation in parallel component architectures. In: Proceedings of the 19th international parallel and distributed processing symposium: IPDPS, Denver, 3–8 April 2005
10. Bruneton E, Coupaye T, Leclercq M, Quéma V, Stefani JB (2006) The fractal component model and its support in java. Softw Pract Exp 36:11–12 (Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems)
11. Cansado A, Caromel D, Henrio L, Madelaine E, Rivera M, Salageanu E (2007) A specification language for components implemented in GCM/ProActive. LNCS series. Springer, Heidelberg
12. CCA forum (2005) The common component architecture (CCA) forum home page. http://www.cca-forum.org/
13. CoreGRID Institute on Grid Systems, Tools, and Environments (2008) Design methodology of the generic component-based grid platform. Deliverable D.STE.07
14. CoreGRID, Programming Model Institute (2006) Basic features of the grid component model (assessed). Deliverable D.PM.04. http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf
15. Coulson G, Grace P, Blair G, Mathy L, Duce D, Cooper C, Yeung WK, Cai W (2004) Towards a component-based middleware framework for configurable and reconfigurable grid computing. wetice 00:291–296. ISSN 1524-4547. doi:http://doi.ieeecomputersociety.org/10.1109/ENABL.2004.69
16. Czajkowski K, Kesselman C, Fitzgerald S, Foster I (2001) Grid information services for distributed resource sharing. HPDC 00:0181. doi:http://doi.ieeecomputersociety.org/10.1109/HPDC.2001.945188
17. Danelutto M, Vanneschi M, Zoccolo C, Tonellotto N, Orlando S, Baraglia R, Fagni T, Laforenza D, Paccosi A (2005) HPC application execution on GRIDs. Future Gener Grids 263–282
18. Denis A, Perez C, Priol T, Ribes A (2004) Bringing high performance to the corba component model. In: SIAM conference on parallel processing for scientific computing, San Francisco, 25–27 February 2004
19. Dünnweber J, Baude F, Legrand V, Parlavantzas N, Gorlatch S (2006) Invited papers from the 1st CoreGRID integration workshop, Pisa, Novembre 2005, chap. Towards automatic creation of web services for grid component composition. Volume 4 of CoreGRID series. Springer, Heidelberg
20. Furmento N, Hau J, Lee W, Newhouse S, Darlington J (2004) Implementations of a service-oriented architecture on Top of Jini, JXTA and OGSI. In: AxGrids 2004, no. 3165 in LNCS. Springer, Heidelberg, pp 90–99
21. Furmento N, Lee W, Mayer A, Newhouse S, Darlington J (2002) Iceni: an open grid service architecture implemented with jini. SC 00:37. ISSN 1063-9535. doi:http://doi.ieeecomputersociety.org/10.1109/SC.2002.10027

22. Gannon D (2002) Programming the grid: distributed software components. http://citeseer.ist.psu.edu/gannon02programming.html
23. Gannon D, Fox G (2006) Workflow in grid systems meeting. Concurrency & computation: practice & experience vol 18, issue 10 (Based on GGF10 Berlin meeting)
24. Getov V, von Laszewski G, Philippsen M, Foster I (2001) Multiparadigm communications in java for grid computing. Commun ACM 44(10):118–125. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/383845.383872
25. Henrio L, Rivera M (2008) Stopping safely hierarchical distributed components. In: Proceedings of the workshop on component-based high performance computing (CBHPC'08), Karlsruhe, 14–17 October 2008
26. Herault C, Nemchenko S, Lecomte S (2005) A component-based transactional service, including advanced transactional models. In: Advanced distributed systems: 5th international school and symposium, ISSADS 2005, revised selected papers, no. 3563 in LNCS
27. Kephart JO, Chess DM (2003) The vision of autonomic computing. Computer 36(1):41–50. ISSN 0018-9162. doi:http://dx.doi.org/10.1109/MC.2003.1160055
28. Lacour S, Pérez C, Priol T (2005) Generic application description model: toward automatic deployment of applications on computational grids. In: 6th IEEE/ACM international workshop on grid computing (Grid2005). Springer, Seattle
29. Liu H, Parashar M (2004) A component based programming framework for autonomic applications. In: 1st IEE int. conference on autonomic computing (ICAC), New York, 17–18 May 2004
30. Malawski M, Bubak M, Baude F, Caromel D, Henrio L, Morel M (2007) Interoperability of grid component models: GCM and CCA case study. In: CoreGRID symposium in conjunction with Euro-Par 2007, CoreGRID series. Springer, Heidelberg
31. Mayer A, Mcough S, Gulamali M, Young L, Stanton J, Newhouse S, Darlington J (2002) Meaning and behaviour in grid oriented components. In: Third international workshop on grid computing, GRID, vol. 2536 of LNCS, pp 100–111
32. Mencl V, Bures T (2005) Microcomponent-based component controllers: a foundation for component aspects. In: APSEC. IEEE Computer Society, Piscataway
33. OMG.ORG TEAM (2005) CORBA component model, V3.0. http://www.omg.org/technology/documents/formal/components.htm
34. Parlavantzas N, Morel M, Getov V, Baude F, Caromel D (2007) Performance and scalability of a component-based grid application. In: 9th int. worshop on java for parallel and distributed computing, in conjunction with the IEEE IPDPS conference
35. Partridge C, Menedez T, Milliken W (1993) Host anycasting service. RFC 1546
36. Seinturier L, Pessemier N, Coupaye T (2005) AOKell: an aspect-oriented implementation of the fractal specifications. http://www.lifl.fr/~seinturi/aokell/javadoc/overview.html
37. Seinturier L, Pessemier N, Duchien L, Coupaye T (2006) A component model engineered with components and aspects. In: Proceedings of the 9th international SIGSOFT symposium on component-based software engineering, Västerås, June 2006
38. Szyperski C (1998) Component software: beyond object-oriented programming. ACM/Addison-Wesley, New York. ISBN 0-201-17888-5
39. Tejedor E, Badia RM, Naoumenko P, Rivera M, Dalmasso C (2008) Orchestrating a safe functional suspension of gcm components. In: CoreGRID integration workshop. Integrated research in grid computing
40. Thome B, Viswanathan V (2005) Enterprise grid alliance–reference model v1.0

3.2 Design and structure of non-functional aspects

In Fractal, each component is made of two parts: the content that contains the code or sub-components dealing with the functional code of the application, and the membrane that contains the objects managing the non-functional (NF) aspects. In order to better design the non-functional aspects but also to better adapt non-functional features at runtime, we proposed to organise the membrane as a component system. A first version of this work was suggested in the definition of GCM, then the model was refined during Paul Naoumenko's PhD thesis [30, 39], [3].

In Fractal, the AOKell [SPC05, SPDC06] framework already allows the design of NF features as a component system, in a non-distributed setting. In AOKell, non-functional concerns are expressed as components and can be composed using the Fractal API. However, NF components cannot be distributed, neither collaborate with object controllers. Moreover, the membrane is designed and executed as a composite component entailing one additional level of indirection for requests toward the membrane components. Following AOKell developments, Julia's Fractal implementation moved towards components in the membrane; their approach is similar to the one of AOKell.

Control microcomponents from the Asbaco project [MB05] are specific components using a control API different from Fractal's, and requiring a specific ADL language. Microcomponents are very simple components that do not support hierarchy, and thus provide efficiency at the expense of expressiveness of the NF aspects. No reconfiguration of microcomponents is allowed.

Compared to the preceding approaches, we proposed a componentisation of the component membrane such that the NF features can be composed in the same way and with the same expressing power as functional composition.

First, in order to allow NF aspects to be connected in the same way as functional ones, we added the possibility to have NF client interfaces in GCM components. Indeed Fractal components provide functional client and server interfaces, and only *server* NF interfaces. Thanks to our new NF client interfaces, NF interfaces can be bound together and components can be interconnected also for triggering NF actions. This enhances the re-usability of components since cooperation of NF features of different components can be configured, and even reconfigured dynamically.

Second, we allowed components to be part of the membrane, we extended both the API to allow components to be added in the membrane, connections to be performed between membrane components, and more generally between NF interfaces of components. We also proposed an extension to the GCM ADL to allow the definition and management of componentised membranes. A complete definition of the new API can be found in [3]. This API has been implemented as part of ProActive/GCM.

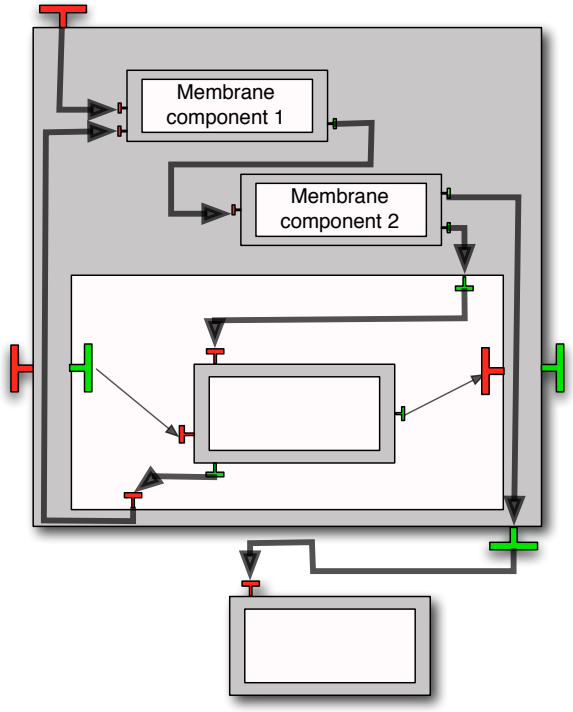


Figure 3.2: Structuring the Non-functional concerns in GCM

The new membrane structure enabled by our framework is shown in Figure 3.2. The figure illustrates also the different kind of bindings existing between NF interfaces; one can notice that there are many more different kinds of bindings for the NF aspects than for the functional aspects. For example, the two client interfaces of the "Membrane component 2" are connected in a very different way: one is connected to the inside of the composite component, and the other is connected to the outside world, finally a third connection for client interface is illustrated by the connection between "Membrane component 1" and "Membrane component 2".

From this implementation and in relation with the Bionets European project, we experimented our new architecture for creating and managing at runtime autonomic components. The design of autonomic components revealed easier with our approach. Even more importantly, we were able to make the management of the components itself evolve at runtime. We were able for example to plug different adap-

tation strategies at runtime, for example depending on the execution context or on the requirements of the application.

GCM components revealed to be nicely adapted to the programming of autonomic entities, first thanks to the low coupling between components offered by active objects, but also the new design of the membrane makes it easy to program decentralised decision systems as bindings between components responsible for the management of the system can be changed at design time and at runtime. This makes component management more adaptive and re-usable.

The component structure of membranes raises a frequent question: "if the membrane can contain components, what about the membrane of those components? At which level of management will componentisation stop?". In Microcomponents and AOKell this level is explicitly limited: microcomponents are very simple and cannot be managed by components, while in AOKell one further level of management exists but with limited capacities. In our case, we envision that a component can be arbitrarily big and we think that such a limit should depend on the application; thus we do not enforce a strong limit on the maximum management level but at some point the application designer will instead choose that a membrane is only composed of objects, which gives a limit to the management hierarchy.

Since their creation, componentised membranes have been reused in the GCM developments, in particular during the PhD of Cristian Ruz [Ruz11,RBS11] for realising MAPE (Monitor – Analyse – Plan – Execute) autonomicity loops. This framework provides a very modular framework for the autonomic management of applications.

3.3 Reconfiguring distributed components

By nature, GCM inherits from Fractal reconfiguration capacities. The component structure is known at runtime and can be introspected and modified dynamically. Mainly, Fractal and GCM provide capacities for adding and removing components inside a composite component, and for changing bindings between components. However, those reconfiguration capacities could be better adapted to distributed loosely coupled components.

As underlined in [Hil04], maintaining integrity during reconfiguration and adaptation is crucial. More precisely, in order to be reconfigured, a component assembly should reach a state where components are stopped, and considered as easily reconfigurable. In most implementations of Fractal it is necessary to stop the target components (but not the whole application) before reconfiguring them. Stopping a component can then be hierarchical (stopping all sub-components) or not, depending on the implementation.

In the case of GCM/ProActive, components behave asynchronously as they consist of active objects with futures. We provided an algorithm to stop a ProActive/GCM composite component together with all its subcomponents. The difficulty in this task is to synchronise several active objects which provide only weak synchronisation capacities. Additionally a stopped component cannot perform any more operations which can easily lead to a dead-lock during the stopping process. It is a very challenging task to design a protocol to stop a GCM component subsystem in a safe manner and without deadlock. We tried to address this challenge with Marcela Rivera in [37], and that will be briefly presented in section 3.3.2.

Once the component has been stopped, existing Fractal/GCM primitives for reconfigurations could be applied, however, those primitives are rather low-level. Different approaches can be adopted for designing a reconfiguration language for distributed components.

First, in [4] we designed GScript, a scripting language for CCA and GCM components. It provides a scripting language for high-level orchestration and interaction with distributed components. GScript programs can trigger reconfiguration of GCM components by direct invocation on the adequate component interfaces. They can also trigger any action, including computation, on any port of a CCA assembly.

Then, we designed a reconfiguration language closer to GCM [38], it is an extension of FScript [DLLC09, DL06] dedicated to distributed components. It allows the distributed interpretation of reconfiguration scripts. Overall we consider it greatly increases the capacities of FScript in the context of distributed components by providing the functionalities necessary to turn a centralised script interpretation into a distributed one, where script for reconfiguration can be interpreted in several components of the system, in a parallel and distributed manner.

3.3.1 Related works

In [RP03], the authors provide a framework for dynamic component reconfiguration on Microsoft .NET environment. Their reconfiguration capacities are limited to three basic operations: adding a component, removing a component, and setting its attributes. The goal of this work is to find an algorithm able to reconfigure an application without disrupting the service. Their proposal consists in reconfiguring an application if its components don't interact

and if there are no pending transactions. However, they do not have an algorithm for stopping the system. The algorithm blocks the communications in order to reach a reconfigurable state.

The CASA approach [MG05b] also provides a framework for enabling the development and the execution of autonomic applications. In this framework Mukhija and Glinz propose a sequence of steps for dynamic replacement of some components of a system [MG05a]. The adaptation policy is defined by a contract written in XML. CASA takes care of resource allocation, application-level adaptation, but also lower-level service adaptation. This process doesn't apply to our kind of components, but rather to object-like entities. The communications that go out of the component to be stopped are marked in our algorithm, which would not be possible if required interface were not clearly identified as in CASA. As identified by [CS02], we use dependency informations (in our case provided by the component model) to efficiently stop and reconfigure the system.

In the context of component models, in [LP04] the authors presents a component-based programming framework. In this framework the applications can be autonomically reconfigured to manage the dynamism and uncertainty of the applications and the environment. They enable the description of the dynamic replacement / addition / deletion of components, and the change of interaction relationships. Autonomic composition and evolution is expressed as a list of rules. Expressing autonomicity of component systems as adaptation rules is a quite classical approach. By contrast, our contribution here is simply to provide a high-level reconfiguration language that can be used as the effect of rules; in our work, we make this language adapted to distribution

The works mentioned above do no support hierarchical components. On the other side, hierarchical component models like SOFA [HB07] or Fractal [BCL$^+$04] have great advantages concerning management and design of component systems. We focus on such hierarchical models that already provide the basic primitives for reconfiguration to increase the safety and the possibilities offered by reconfiguration in those distributed component frameworks.

In [TBN$^+$08] an alternative way of safely stopping a component system is proposed, also for GCM. It builds on a language for specifying the sequence required to stop a component system. This sequence is implemented by the assembler, who is aware of the behaviour of the complete system. On the contrary, the algorithm presented in the next section is independent of the component topology, and does not require any specification from the programmer or the assembler.

Let us now focus on languages for expressing dynamic evolution of component systems. We focus on the two closest works: FScript and GScript.

FScript [DL06] is a domain-specific language to program dynamic reconfigurations of Fractal architectures. FScript directly triggers actions on the non-functional interfaces of Fractal components. FScript programs can easily navigate inside, and reconfigure a Fractal assembly. Each instruction results in one or several invocations on component interfaces, which can either introspect components or reconfigure them. For navigation, FScript uses the FPath notation. The expressivity of FScript is close to Fractal. In FScript for example it is not possible to add new interfaces to an existing component because the Fractal model forbids it.

GScript is a scripting language we presented in [4]. It provides generic primitives for component configuration and for triggering communications towards components. Its main advantage is its generic nature, and its particular support for Grid deployment. Unfortunately, it is not particularly targeted at GCM and does not feature specific constructs like FScript. That is why we chose in [38] to design an extension of FScript targeted at the distributed execution of the reconfiguration scripts.

3.3.2 Stopping components

Considering components are active objects treating incoming requests and replying by means of futures our objective is to stop a component with all its subcomponents. Then the subsystem will be in an adequate state to be reconfigured safely: neither communications, nor any local execution will be interleaved with the reconfiguration process. Additionally, to guarantee that no computation is being performed in the subsystem, we require that all the inner components of the stopped components are stopped with an empty request queue. This last condition delays the eventual stop of the subsystem but guarantees that it will not be stopped in the middle of some crucial operations. Also, without this condition, the stopping protocol becomes much more difficult to design, if it ever exists ...

If the system cannot be stopped safely, the algorithm never finishes but does not block, this keeps the integrity of the system and lets it run normally. It is always possible to add a non-safe stopping algorithm based on a timeout to stop any system, but as safety is lost we do not consider such a solution as reasonable.

Our algorithm relies on the following assumptions relatively to the component model:

- Components do not share memory, and there is no shared component, i.e. component hierarchy is a tree.
- Components communicate by asynchronous *requests* which can be remote method calls or any other asynchronous communication.

- All communications are performed using the bindings defined by the component structure, and respecting component hierarchy: a component can only communicate with components at the same level of hierarchy or with its parent or children.
- The communication mechanism can be instrumented by adding information to every message.

The reached state is sometimes called *quiescent* state in the literature: it is a state where the inner components of the stopped component have a minimal internal state and will not trigger any new communication or computation.

We refer the reader to [37], [73] for details on the algorithm, but we explain its principles below.

We call *master* the component to be stopped, at the end this component will be stopped, and all its inner components (if it has some) will be stopped with an empty request queue.

A first phase of the stopping protocol consists, for the master component in marking all outgoing requests. This phase stops when the master has no more reference to a future corresponding to a non-marked request. Consequently, at the end of this phase, each request sent by (or through) the master to the outside world will either be finished or marked. This marking algorithm allows the identification of re-entrant requests: each marked request may have to be treated to ensure that the component can be stopped consequently, each request issued during the treatment of a marked request is marked too, and in the second phase, the composite component serves no request coming from the outside, except marked ones.

In the second phase each inner component should be stopped with an empty request queue, this is done by some form of two phase commit. Each component signals when it is ready to be stopped, i.e. it has no request to serve and is idle. However a component that was ready might become non-ready if it receives a request from another non-ready component. At the end, when all the sub-component are ready, they will not have any more request to serve, and the master with all its subcomponents can be stopped in order to be safely reconfigured.

We experimented this algorithm on two case studies, trying to stop different components. While experimentally we verified its good behaviour, and we are convinced that probably no dead-lock and no live-lock exist in the system, the component system will be stopped safely, no formal proof of the correctness of the algorithm has been written. We however highlighted the properties of the algorithm and explained informally in [37] why they are verified. Considering the complexity of the algorithm proving formally its correctness is a real challenge.

3.3.3 A language for distributed reconfiguration

We also designed a framework dedicated to the reconfiguration of distributed components, and in particular to the reconfiguration of GCM components. When studying the adequacy of languages dedicated to the reconfiguration of components to the GCM, we realised that FScript was close to be adequate but was too much centralised. Indeed, FScript provides the primitives that are necessary to reconfigure a component system made of Fractal or GCM components, but the reconfiguration scripts were designed to be interpreted in a centralised manner.

In a distributed component model, it seems more reasonable to allow the distributed interpretation of scripts: several composite components can be responsible for reconfiguring their subcomponents independently. This distributed approach allows parallelism, and thus allows reconfiguration procedures to be run on large-scale infrastructures. It is also better adapted to program autonomic adaptation procedures, as each component can embed the adaptation scripts necessary for its adaptation and trigger them autonomically, when needed.

For this, our approach was quite simple but really effective, it is based on two extensions of the FScript framework.

A controller for reconfiguration

We added a non-functional port, localised in several (possibly all) components. This port is able to interpret reconfiguration orders. We called the non-functional object able to interpret reconfiguration scripts a `ReconfigurationController` and embedded it inside the membrane of the desired components.

This controller embeds itself an instance of the FScript interpreter and provides a method `loadScript` for loading reconfiguration (sub)scripts, and `executeAction` for triggering a reconfiguration action.

Note that stopping a component does not stop the membrane and thus a stopped component can still interpret reconfiguration scripts and be reconfigured.

A primitive for distributed reconfiguration

We extended FScript with primitives for triggering the remote execution of reconfiguration scripts. The primitive `remote_call` triggers the execution of a reconfiguration action on a remote component. The target component is given by its node, specified as a FPath expression.

Upon remote script invocation, if no remote interpreter is available then one is automatically created by calling the `setInterpreter` method on the remote reconfiguration controller. After this call, the target component becomes in charge of the interpretation of the reconfiguration. Then the calling interpreter can continue the execution of its local script. Each reconfiguration script then runs independently.

We also defined some helper functions that extend the FScript language and revealed to be interesting in the context of a distributed interpretation of reconfigurations. The most interesting function is a function `evaluate` that takes as argument a string that contains a FPath and evaluates it locally. This function allows the programmer to pass an FPath as an argument of a remote script interpretation and interpret it at the destination side (instead of the caller side if the FPath was not embedded in a string).

3.3.4 Concluding remarks

Those works leaded to very few formal developments even if the algorithm has been specified in a relatively formal way allowing it for example to be easily encoded and verified on a specific example with a model-checker. However the general proof of validity of this algorithm involves too many (simple) cases and steps to be convincing by hand, it is also too complex and relies on too many notions to be encoded easily in a theorem prover. Additionally, theorem provers are not particularly suited to encode this kind of algorithm, where each component has a state, and the decision process is a complex stateful procedure. Note however that we recently made some form of progress on the verification of distributed protocols/algorithm but in a different context, as we will show in Section 4.1.

Note that in our reconfiguration framework, when the reconfiguration action finishes, no automatic notification is triggered; it is not possible to know automatically whether a remotely invoked script succeeded or not. However, callbacks can be used to return the status of the remote script, and further synchronisation primitives could also be added to the language to synchronise the different reconfiguration controllers. One could note that we loose part of the guarantees of FScript by our extension: in FScript it is possible to rollback a failed reconfiguration script. In our approach we wanted the reconfiguration to be efficient and the synchronisation between scripts to be lightweight, that is why we did not provide any distributed rollback mechanism. However, if consistency of the reconfiguration is necessary, it is still possible to interpret a reconfiguration procedure inside a single script interpreter.

Those contributions have been implemented in the ProActive/GCM framework, together with several use-cases like CoCoME [5] which showed that our approach was effective. Our extension of FScript has been adopted as the basis for a language for the reconfiguration of GCM components: GCMScript.

3.4 A semantics for GCM: specification, formalisation and futures

In this section we will review the efforts that we made around GCM to give it a semantics, and to prove properties on the component systems. First, one can notice that neither Fractal nor GCM give any semantics (neither formal nor informal) to the behaviour at runtime of components. However, to be able to prove properties on the execution of component applications, one must rely on some well defined semantics for the underlying programming language and/or middleware.

The first work we did in that domain relied directly on the ASP calculus. It demonstrates how we can go from asynchronous distributed objects to asynchronous distributed components, including collective remote method invocations (group communications), while retaining determinism [26]. It simply consists in expressing the way active objects can be instantiated from the definition of a component system (either keeping the hierarchical structure or flattening the composition). It expresses quite well the active object instantiation of components featured by ProActive/GCM and relies on a translation from an ADL into an ASP term.

However we think this work did not enable direct reasoning on the component model and on the component system. That is why we provided a semantics for GCM components, directly in terms of component behaviour. It relies on notions closed to Actors and active objects to ensure loose coupling of components, and to give them a data-flow oriented synchronisation.

We need a model for distributed components, and think it should be based on one key principle: *Components are the unit of concurrency.* More precisely, similarly to active objects, components only communicate by sending requests or results for those requests, and requests are sent along bindings. We say that this model is asynchronous because requests can be treated in an asynchronous manner thanks to the introduction of *futures* (place-holders for request results). In order to prevent other communications or concurrency to occur, we require that *components do not share memory*, as explained above this also makes our model adapted to distribution. From a computational point of view, components are loosely coupled: the only strong synchronisation consists in waiting for the result of a request,

and can be performed only when and where this result is really needed thanks to the use of futures.

We thus consider the GCM model, where communication is chosen to be a request / reply mechanism with futures. Our objective is to provide a programming model more general than the one adopted in ProActive/GCM, but more precise than the purely structural GCM definition. ProActive/GCM can then be considered as *a possible* implementation of our model where components are implemented as active objects. However our semantics is more general for several reasons: first it does not rely on the notion of objects, and primitive components are just defined by their behaviour, second primitive components do not need to be mono-threaded, as it is the case for active objects. Overall our model does not deal explicitly with states or object manipulation, or with the programming of basic business code. Instead it composes the behaviour of the primitive components by giving a semantics to the communications and to the composition.

3.4.1 Informal semantics

We presented in [41] a component semantics and its formalisation based on the idea that interaction between components is limited to communications, and more precisely to a request/reply mechanism. We present below the principles of this semantics. Its formalisation in Isabelle/HOL is then presented in a paper included below.

Communication

The basic communication paradigm we consider is asynchronous message sending: upon a communication the message is enqueued at the receiver side in a queue. To prevent shared memory between components, messages can only transmit parameters which are copied at the receiver side; no object or component can be passed by reference.[2] This communication semantics is similar to requests in an active object models and actors. We call *requests* messages sent between components. References to components cannot be passed between components, for example, method parameters cannot contain references to components. More precisely, only non-functional features should be able to manipulate component and interface references, but we do not describe them for the moment.

[2]To be precise, only futures are passed by reference, because their value will be finally transmitted by a copy semantics.

Returning results

We call our component model asynchronous because communication does not trigger computation on the receiver side immediately, it just enqueues a request. However such a mechanism can be implemented with synchronous or asynchronous communications. Like in ASP, we consider here that enqueueing a request is done synchronously but the receiver component is always ready to enqueue a request. This has the great advantage to ensure causal ordering [CBBM+96] of messages. To allow for transparent asynchronous requests with results, we use transparent first-class futures *à la ASP*. Remember this means that futures are created automatically, can be transmitted between components, and are subject to wait-by-necessity.

Primitive component behaviour

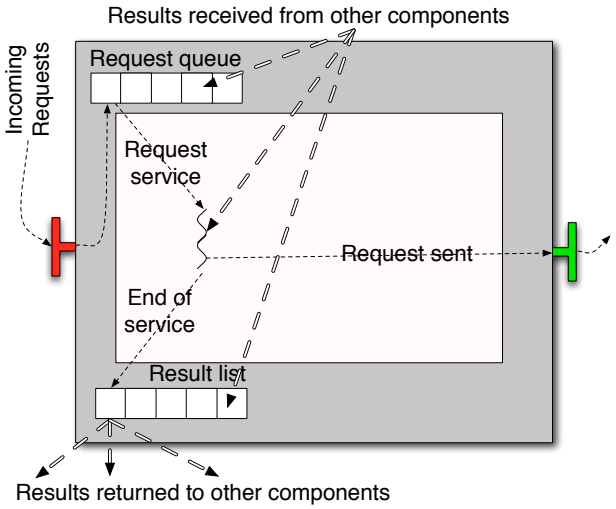


Figure 3.3: Component behaviour

The primitive components encapsulate the business code, thus in our model we consider they can have, internally, any behaviour. They will serve requests in the order they wish, providing answer for all the requests they receive. They can call other components by emitting a request on one of the client interfaces. However, each primitive component must always be able to accept a request (that will just be enqueued in its request queue), and to receive a result (that will replace a future reference by the received value).

Figure 3.3 illustrates a primitive component and its behaviour. A primitive component consists of a request queue, a content, a membrane, and a result list. Its content contains the business code that serves the requests; requests arrive from the server interfaces on the left and are emitted by the client interface on the right. An incoming request is

enqueued immediately; it is always associated with a future identifier. Later this request is served and treated by the component content, possibly emitting new requests to the clients. When the service is finished and a value is calculated for the result, this value is stored in the result list: the future for the request now has a value, which is the newly calculated result. The calculated value can itself contain references to other futures. Later, the result will be sent from the result list to the components that hold a reference to the corresponding future. As future references can spread in all the components, including requests, results, and current component states, received results are used to update future references in all parts of the component. In principle, like in ASP, future values can be returned at any time, however we performed a study of different strategies for returning future values and formally specified some of them (see Section 3.4.2).

In our model, a given thread manipulates a single component, but nothing prevents our components from being multi-threaded, and a component can serve several requests at the same time.

Composite component behaviour

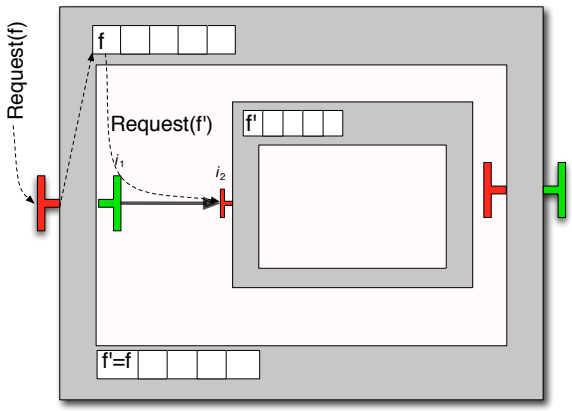


Figure 3.4: Request delegation

The behaviour of the primitive components is highly parameterised because they contain the application logic. By contrast, composite components have a predefined behaviour because they are only used as composition tools and the programmer expects them to only transmit the requests according to the specified composition. Composite components serve requests in a FIFO order, delegating request to the bound components or to the external ones. Globally, a request emitted by the client interface of a primitive component will be sent unchanged to the server interface of the primitive component that is bound to it, following one or several binding (several bindings are used when several composite components are crossed). The composite component performs almost no computation: it only delegates calls immediately.

The delegation of requests from a composite component to its sub-component is illustrated by Figure 3.4. Consider one request (associated with the future f). Suppose the request has been received from the outside of the composite, i.e., it was received on an external server interface. There is necessarily an internal client interface matching this external one. Handling the requests consists in sending another request from the internal client interface matching the interface that receives the request ($i1$). This request is sent to the interface bound to $i1$, $i2$ in the figure; this interface necessarily belongs to an inner component. This new request corresponds to a future f', and the result for the first one is just a reference to f', denoted $f = f'$ in the figure. In case the request was received from the inside of the composite, the mechanism is similar: the request received at the internal server interface is delegated to the matching external client interface, through the composite component request queue.

An alternative approach would consist in implementing a delegation mechanism allowing a component to *delegate* the calculation of a result to another component, like handlers of [NSS06]. More precisely, with delegation, a component could state that it is a role of another component to give the answer for a request (f in the example) instead of stating that the result of the request is known but is another future ($f = f'$ in the example). However, we did not choose this technique in order to avoid introducing a new mechanism, but also to ensure that the component calculating a value for a given future will not change along time.

Mono-threaded components

While our specification allows multi-threading (or cooperative multi-threading *à la* Creol), the ProActive/GCM implementation of the component model is single-threaded as it relies on active objects *à la* ASP. As in active objects, this can create deadlocks in case of cycle of dependencies between requests (sort of re-entrance problem).

Also in ProActive/GCM, composite themselves are active objects, and inside composites first-class futures are necessary to avoid an almost systematic deadlock. Indeed, in a first ProActive/GCM implementation first-class futures were not activated by default, and the component applications almost systematically deadlocked.

To understand the reason of this synchronisation problem, consider the example component of Figure 3.5. In this example, a request `Foo()` arrives in the composite component from the left. The request is delegated to the subcomponent that is a primitive. The primitive performs a

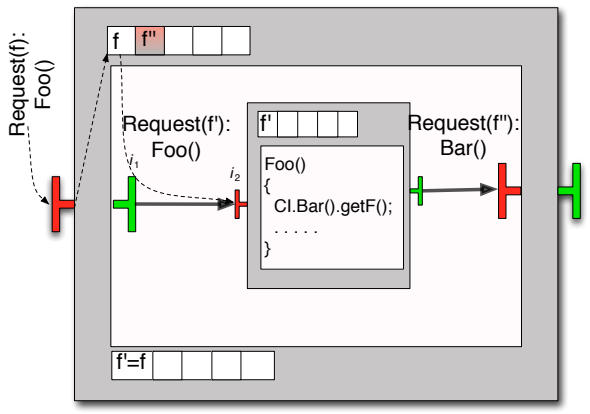


Figure 3.5: Why first-class futures are necessary inside composites.

call on its client interface (`Bar` request), and accesses the result (`getF` method call). The call on the client interface passes through the composite again to be sent to another component. Consider now the case where the composite is an active object without first-class futures. The service of request `Foo()` in the composite consists in returning the result of the call delegated to the primitive, in Java, the method `Foo` of the composite simply performs a: `return Primitive.Foo();` As futures are not first class, the composite is stuck waiting for the result of this invocation and cannot handle the request `Bar()`. Both the primitive and the composite components are stuck. If on the contrary futures are first-class, the composite can return the future corresponding to the delegated `Foo` request to the invoker, and consequently serve the `Bar` request, sending it to an outside component. This real-life example shows the importance of first-class futures in our context. Consequently, and thanks to the properties we proved on ASP, first-class futures are now activated by default in ProActive.

3.4.2 Paper from FMCO 2009

We wrote a formal version of the semantics described above in [41]. This semantics was specified in Isabelle/HOL and extended in [44] to specify a given future update strategy and prove its properties. In a formal semantics the moment when the REPLY rule is applied can be unspecified, allowing a future value to be sent at any time. However, in the context of a real implementation a protocol should specify when future values are to be returned. We call such a protocol a *future update strategy*.

The main future update strategies, first mentioned in [1] were precisely defined and experimented during Muhammad Khan PhD thesis [45]. One strategy was then specified in Isabelle/HOL [44] and proved to be correct and complete. The article below [43] presents the formalisation in Isabelle/HOL of the GCM component structure that allowed us to perform all those proofs. Additionally to the structure this article shows that it is possible to reason on component structure of components at runtime with our framework. It is important to note that comparatively to GCM our specification misses the collective interfaces (one-to-many and many-to-one), the collection interfaces of Fractal, and the non-functional structure. Those could be added in the future, but already we showed that our model was sufficient to formalise crucial proofs on the component framework. The reader is referred to the article below for a study of the closest related works on the specification of component models.

The formalisation of the GCM component model in Isabelle/HOL is available at: `www-sop.inria.fr/oasis/Ludovic.Henrio/misc.html`

Our definition of components being both precise and formalised, we expect it to be a strong guide and a reliable basis for both component system implementations and the proof of their properties.

A Framework for Reasoning on Component Composition

Ludovic Henrio[1], Florian Kammüller[2], and Muhammad Uzair Khan[1]

[1] INRIA – CNRS – I3S – Université de Nice Sophia-Antipolis
{mkhan,lhenrio}@sophia.inria.fr
[2] Institut für Softwaretechnik und Theoretische Informatik – TU-Berlin
flokam@cs.tu-berlin.de

Abstract. The main characteristics of component models is their strict structure enabling better code reuse. Correctness of component composition is well understood formally but existing works do not allow for mechanised reasoning on composition and component reconfigurations, whereas a mechanical support would improve the confidence in the existing results. This article presents the formalisation in Isabelle/HOL of a component model, focusing on the structure and on basic lemmas to handle component structure. Our objective in this paper is to present the basic constructs, and the corresponding lemmas allowing the proof of properties related to structure of component models and the handling of structure at runtime. We illustrate the expressiveness of our approach by presenting component semantics, and properties on reconfiguration primitives.

Keywords: Components, mechanised proofs, futures, reconfiguration.

1 Introduction

Component models focus on program structure and improve re-usability of programs. In component models, application dependencies are clearly identified by defining interfaces (or ports) and connecting them together. The structure of components can also be used at runtime to discover services or modify component structure, which allows for dynamic adaptation; these dynamic aspects are even more important in a distributed setting. Since a complete system restart is often too costly, a reconfiguration at runtime is mandatory. Dynamic replacement of a component is a sensitive operation. Reconfiguration procedures often entail state transfer, and require conditions on the communication status. A suitable component model needs a detailed representation of component organization together with precise communication flows to enable reasoning about reconfiguration. That is why we present here a formal model of components comprising both concepts.

This paper provides support for proving properties on component models in a theorem prover. Our objective is to provide an expressive platform with a wide range of tools to help the design of component models, the creation of adaptation procedures, and the proof of generic properties on the component model.

F.S. de Boer et al. (Eds.): FMCO 2009, LNCS 6286, pp. 1–20, 2010.

Indeed most existing frameworks focus on the correctness or the adaptation of applications; we focus on generic properties.

In this context, introduction of mechanised proofs will increase confidence in the properties of the component model and its adaptation procedures. We start from a formalisation close to the component model specification and implementation; then we use a framework allowing us to express properties in a simple and natural way. This way, we can convince the framework programmer and the application programmer of the safety of communication patterns, optimisations, and reconfiguration procedures.

We write our mechanised formalisation in Isabelle/HOL but we are convinced that our approach can be adapted to other theorem provers. The generic metalogic of Isabelle/HOL constitutes a deductive frame for reasoning in an object logic. Isabelle/HOL also provides a set of generic constructors, like datatypes, records, and inductive definitions supporting natural definitions while automatically deriving proof support for these definitions. Isabelle has automated proof strategies: a simplifier and classical reasoner, implementing powerful proof techniques. Isabelle, with the proof support tool Proofgeneral, provides an easy-to-use theorem prover environment. For a precise description of Isabelle/HOL specific syntax or predefined constructors, please refer to the tutorial [20].

We present here a framework that mechanically formalizes a distributed hierarchical component model and its basic properties. We show that this framework is expressive enough to allow both the expression of component semantics and the manipulation of the component structure. Benefiting from our experiences with different possible formalisations, and from the proof of several component properties, we can now clearly justify the design choices we took and their impact[1]. The technical contributions of this paper are the following:

- formal description in Isabelle of component structure, mapping component concepts to Isabelle constructs,
- definition of a set of basic lemmas easing the proof of component-related properties,
- additional constructs and proofs to ensure well-formedness of component structures,
- proposal for a definition of component state, and runtime semantics for components communicating by asynchronous request-replies,
- application to the design and first proofs about component reconfiguration.

The remainder of the paper is organised as follows. Section 2 gives an overview of the context of this paper: it positions this paper relatively to related works and previous works on the formalisation of the GCM component model, which is also described in Section 2.2. Section 3 presents the formalisation of the component model in Isabelle/HOL highlighting design decisions and their impact on the basic proof infrastructure. We then summarize a semantics for distributed components with its properties, and present a few reconfiguration primitives in Section 4.1. Section 5 concludes and presents future directions.

[1] The GCM specification framework is available at `www.inria.fr/oasis/Ludovic.Henrio/misc`

2 Background

Component modelling is a vast domain of active research, comprising very applied semi-formal approaches to formal methods. In this section, we first give an overview of the domain, starting from well-known approaches, summarizing some community activities, and focusing on the most relevant related works. Then we present the GCM component model and existing formalisation of GCM. Finally we position this paper relatively to the other approaches presented here.

2.1 Related Work

Some well-known component models like CCA [11] are not hierarchical – their intent is the efficient building, connecting and running of components but they neglect structural aspects. We rather focus on hierarchical component models like Fractal[6], GCM[4], or SCA[5].

Recent years have shown several opportunities for the use of formal methods for the modelling and verification of component-based applications as shown in several successful conferences like FMCO, FOCLASA, or FACS.

For example, in [8, 9] the authors investigate the use of formal methods to specify interface adaptation and generation of interface adaptors, based on behavioural specification of interfaces to be connected. Also, in [10, 3] the authors focus on the verification of the behaviour of component-based application. They provide tools to specify the behaviour of a component application, and check that this application behaves correctly. Their model is applied to the GCM component model too but they prove properties of specific applications whereas we formalise the component model itself. In [18], the authors present a comprehensive formalisation of the Fractal component model using the Alloy specification language. Additionally, the consistency of resulting models can be verified through the automated Alloy Analyzer. These contributions are close to our domain but focus on the use of formal methods to facilitate the development and ensure safety of component applications, while our aim is to provide support for the design of component models and their runtime support.

SCA (Service Component Architecture) [5] is a component model adapted to Service Oriented Architectures. It enables modelling service composition and creation of service components. FraSCAti [21] is an implementation of the SCA model built upon Fractal making this implementation close to GCM. It provides dynamic reconfiguration of SCA component assemblies, a binding factory, a transaction service, and a deployment engine of autonomous SCA architecture. Due to the similarity between FraSCAti and GCM, our approach provides a good formalisation of FraSCAti implementation. There are various approaches on applying formal and semi-formal methods to Service Oriented Architectures (SOA) and in particular SCA. For example, in the EU project SENSORIA [1] dedicated to SOA, they propose Architectural Design Rewriting to formalize development and reconfiguration of software architectures using term-rewriting [7].

Creol [15, 16] is a programming and modelling language for distributed systems. Active objects in Creol have asynchronous communication by method calls

and futures. Creol also offers components; the paper [12] presents a framework for component description and test. A simple specification language over communication labels is used to enable the expression of the behaviour of a component as a set of traces at the interfaces. Creol's component model does not support hierarchical structure of components. In [2], the authors present a formalisation of the interface behaviour of Creol components. Creol's operational semantics uses the rewriting logic based system Maude [19] as a logical support tool. The operational semantics of Creol is expressed in Maude by reduction rules in a structural operational semantics style enabling testing of model specifications. However, this kind of logical embedding does not support structural reasoning.

2.2 Component Model Overview

Our intent is to build a mechanised model of the GCM component model [4], but giving it a runtime semantics so that we can reason on the execution of component application and their evolution. Thus we start by describing the concepts of the GCM which are useful for understanding this paper. We will try in this paper to distinguish clearly structural concepts that are proper to any hierarchical component model and a runtime semantics that relies on asynchronous requests and replies. Structurally, the model incorporates hierarchical components that communicate through well defined interfaces connected by bindings. Communication is based on a request-reply model, where requests are queued at the target component while the invoker receives a future. The basic component model has been presented in [13] and is summarized below.

Component Structure. Our GCM-like component model allows hierarchical composition of components. This composition allows us to implement a coarse-grained component by composition of several fine-grained components. We use the term *composite component* to refer to a component containing one or more *subcomponents*. On the other hand, *primitive components* do not contain other components, and are leaf-level components implementing business functionality. A component, primitive or composite, can be viewed as a container comprising two parts. A central *content part* that provides the functional characteristics of the component and a *membrane* providing the non-functional operations. Similarly, interfaces can be functional or non-functional. In this work and in the following description, we focus only on the functional content and interfaces.

The only way to access a component is via its interfaces. *Client* interfaces allow the component to invoke operations on other components. On the other hand, *Server* interfaces receive invocations. A *binding* connects a client interface to the server interface that will receive the messages sent by the client.

For composite components, an interface exposed to a subcomponent is referred to as an *internal* interface. Similarly, an interface exposed to other components is an *external* interface. All the external interfaces of a component must have distinct names. For composites, each external functional interface has a corresponding internal one. The implicit semantics is that a call received on a server

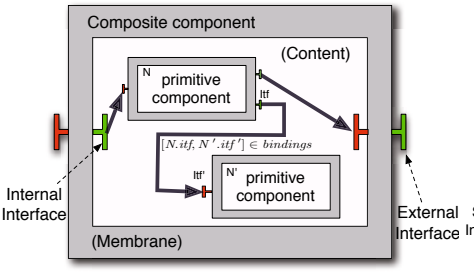


Fig. 1. component composition

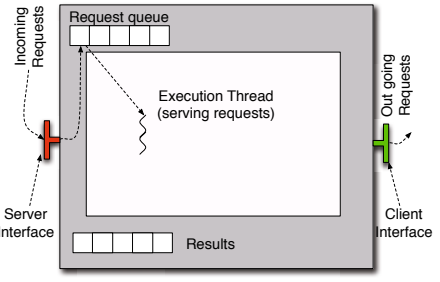


Fig. 2. component structure

external (resp. internal) interface will be transmitted – unchanged – to the corresponding internal (resp. external) client interface.

The GCM model allows for a client interface to be bound to multiple-server interfaces. For the moment, in our model, we restrict the binding cardinality such that bindings connect a client to a single server. Note that several bindings can anyway reach the same server interface.

Figure 1, shows the structure of a composite component. The composite component contains two primitive subcomponents N and N'. The binding ($N.itf$, $N'.itf'$) connects the client interface itf of subcomponent N to the server interface itf' of subcomponent N'.

Communication Model. Our GCM-like components use a simple communication model relying on asynchronous request and replies, as presented in [13]. Communication via requests is the only means of interaction between components. We avoid shared objects or component references, and use a pass-by-copy semantics for request parameters. A component receives the requests on its external server interface. The received requests are then enqueued in the *request queue*, which holds the messages until they can be treated.

Our communication model is asynchronous in the sense that the requests are not necessarily treated immediately upon arrival. Requests are only enqueued at the target component, then the component invoking the request can continue its execution without waiting for the result. Enqueuing a request is done synchronously but the receiver is always ready to receive a request. To ensure transparent handling of asynchronous requests with results, we utilise *futures*. Futures are created automatically upon request invocation and represent the request result, while the treatment of the request is not finished. Once the result of the computation is available, the future is replaced by the result value. Futures are first class objects: they can be transferred as part of requests or results.

Figure 2 gives the internal structure of a component. Incoming requests are enqueued in the *request queue*. The requests are dequeued by the execution threads, when computed; the results are placed in the *results list*.

Component Behaviour. In our model, the primitive components represent the business logic and can have any internal behaviour. Primitive components treat all the requests they receive, choosing a processing order and the way to treat them. On the other hand, the behaviour of a composite component is more restricted: it is strictly defined by its constituent subcomponents and the way they are composed. A composite component serves its requests in a FIFO manner, delegating them to other components bound to it. A delegated request is delivered unchanged to the target component. Once the service of a request is finished, the produced result is stored in the computed results for future use. It can then be transmitted to other components, as determined by the reply strategy [17, 14].

2.3 Positioning

This paper provides formalisation of hierarchical components and their structure. At our level of abstraction, this structure is shared by several component levels like Fractal, GCM, and SCA. However most implementations of SCA (except FraSCAti) do not instantiate the component structure at runtime. By contrast, to allow component introspection and reconfiguration at runtime, we consider a specification where structural information is still available at runtime. This enables adaptive and autonomic component behaviours. Indeed, component adaptation in those models can be expressed by reconfiguration of the component structure. For example, reconfiguration allows replacement of an existing component by a new one, which is impossible or very difficult to handle in a model where component structure disappears at runtime.

Most existing works on formal methods for components focus on the support for application development whereas we focus on the support for the design and implementation of component models themselves. To our knowledge, this work is the only one to support the design of component models in a theorem prover. It allows proving very generic and varying properties ranging from structural aspects to component semantics and component adaptation.

A formalisation of our communication model along with the component semantics appear in [13]. An extended version of the formal semantics is presented in [14], providing formalisation of one particular reply strategy. Other possible strategies are discussed in [17]. Compared to our previous works, this paper relies on the experience gained in specification and proof and demonstrated in [13, 14] to design a framework for supporting mechanised proofs for distributed components. In particular this paper focuses on the handling of component structure, on a basic set of lemmas providing valuable tooling for further proof, and the illustration of the presented framework to prove a few properties dealing with component semantics and reconfiguration.

3 Formalisation of Component Model in Isabelle/HOL

Our component model is a subset of the GCM component model, but with a precisely defined structure and semantics. It incorporates hierarchical components

that communicate via asynchronous requests and replies. We start with formalising the structure of our components. Based on the structure defined, we present some of the various *infrastructure* operations that allow us to manipulate the components for proving properties. Then we formalise additional constructs to define component's state and request handling, and correctness of a component assembly. Finally we provide a set of very useful lemmas dealing with component structure and component correctness.

3.1 Component Structure

As we have seen in Section 2.2, a component in our model can either be a composite or primitive. A composite component comprises one or more subcomponents. On the other hand, a primitive component is a leaf-level component encapsulating the business logic.

```
datatype Component = Primitive Name Interfaces PrimState
    | Composite Name Interfaces (Component list) (Binding set) CompState
```

The above Isabelle/HOL datatype definition for `Component`s has two constructors `Primitive` and `Composite`. We present below the various elements that make up the structure of our components.

NAME: Each component has a unique name. We use this name as the component identifier/reference.

INTERFACES: Each component has a number of public interfaces. All communication between components is via public interfaces. An interface can be either client or server and by construction a component cannot have two interfaces with the same name.

SUBCOMPONENTS: Composite components have a list of subcomponents, given by the `Component list` parameter. Primitive components do not have subcomponents.

BINDINGS: In composite components, a binding allows an interface of one component to be plugged to an interface of a second component. `(N1.i1, N2.i2)`$\in$`bindings` if the interface `i1` of component `N1` is plugged to the interface `i2` of `N2` where `N1` or `N2` can either be component names or *This* if the plugged interface belongs to the composite component that defines the binding.

STATE: All components, primitive or composite have an associated state. Component state is discussed in more detail in Section 3.3.

Design decisions. In the Isabelle/HOL formalisation we chose to include the name of the component into the component itself. Like for interfaces, a first intuitive approach could be rather to define subcomponents as mappings from names to components. There are, however, major advantages to our approach. When we reason about a component we always have its name, which makes the expression of several semantic rules and lemmas more natural. The main advantage of maps is the implicit elegant encoding of the uniqueness of *Name*(s). As mentioned before, *Name*(s)are used as component references. Unfortunately, this advantage of maps is quite low in a multi-layered component model because

a map can only serve one level. As we want component names to be unique globally, a condition on name uniqueness is necessary.

Subcomponents are defined as lists rather than finite sets because lists come with a convenient inductive reasoning easing proofs involving component structure. Of course it is easy to define an equivalence relation to identify components modulo reordering. On the contrary the bindings of a component are defined as a set because no inductive reasoning is necessary on bindings, and sets fit better to the representation of this construct.

Having a formalisation of component structure alone, although useful, is not sufficient. An adequate infrastructure needs to be developed to help in reasoning on the component model. The next section describes some of the infrastructure operations that allow us to manipulate components inside component hierarchies.

3.2 Efficient Specification of Component Manipulation

This section provides various operations that allow us to effectively manipulate components. These include operation for accessing component fields, mechanisms for traversing component hierarchies, and means for replacing and updating components inside the hierarchical structure. All these operations are primitive recursive functions enabling an encoding in Isabelle/HOL using the `primrec` feature. Using this feature has great advantages for the automation of the interactive reasoning process. Automated proof procedures of Isabelle/HOL, like the simplifier, are automatically adapted to the new equations such that simple cases can be solved automatically. Moreover, the definitions themselves can use pattern matching leading to readable definitions.

Field access. We define a number of operations for accessing various fields. These include the function GETNAME that returns the `Name` of the component.

```
primrec getName:: Component ⇒ Name   where
        getName (Primitive N itf s) = N |
        getName (Composite N itf sub b s) = N
```

Similarly, we define `getItfs` , `getQueue`, and `getComputedResults` for getting interfaces, request queues and replies. Requests and replies are part of the component state described in Section 3.3.

Accessing component hierarchy. In order to support hierarchical components, we need a number of mechanisms to access components inside hierarchies. These range from simply finding a suitable component inside a component list to updating the relevant component with another component. The most useful of these operations are detailed below.

CPLIST: returns a list of all subcomponents of a component recursively. It uses the predefined Isabelle/HOL list operators `#` for constructing lists and `@` for appending two lists. Note that the following primitive recursive function is mutually recursive and needs an auxiliary operation dealing with component lists.

```
primrec cpList:: Component ⇒ Component list and
        cpListlist:: Component list ⇒ Component list
where
  cpList (Primitive N itfs s) = [(Primitive N itfs s)] |
  cpList (Composite N itfs subCp bindings s) =
              (Composite N itfs subCp bindings s)#(cpListlist subCp) |
  cpListlist [] = [] |
  cpListlist (C#CL) = (cpList C)@ cpListlist CL
```

CpSet: gives a set representation of the `cpList` of a component. This allows us to write properties in a much more intuitive way, for example, quantifying over sub-components is easily written as $\forall$ `C' ∈ CpSet(C)`. Note however that a few proofs require to stick to the `CpList` notation; indeed when switching to cpSet construct, one cannot reason on the coexistence of two identical components.

```
constdefs :: Component ⇒  Component set
            cpSet C == set (cpList C)
```

getCp: allows for retrieving a given component from a component list based on the component Name. The constructors `Some` and `None` represent the so-called `option` datatype enabling specifications of partial functions. Here, a component with the given name might not be defined in the list – this is nicely and efficiently modelled by a case distinction over the option type. Note the definition of `^` as an infix operator synonymous for `getCp`. This so-called pretty printing syntax of Isabelle supports natural notation of the form `CL^N = Some C'`.

```
primrec getCp:: Component list ⇒ Name ⇒ Component option where
   getCp [] N' = None |
   getCp (C#CL) N' = if (getName C=N') then Some C else (CL^N')
```

changeCp CL C: written `CL<-C` replaces the component in the list `CL` that has the same name as `C` by `C`; it does nothing if there is no component with the given name.

```
primrec changeCp::Component list⇒Component⇒Component list where
  changeCp [] C = [] |
  changeCp (C#CL) C' = if getName C=getName C' then C'#CL else C#(CL<-C')
```

removeSubCp C N: removes the subcomponent of `C` with name `N` but does nothing if there is no subcomponent with this name. Note, here the use of a `case` switch supporting again pattern matching in Isabelle/HOL definitions.

```
primrec removeSubCp:: Component ⇒ Name ⇒ Component where
  removeSubCp (Primitive N itf s) N' = (Primitive N itf s) |
  removeSubCp (Composite N itf sub b s) N' = (case sub^N' of
            None => (Composite N itf sub b s) |
            Some C => Composite N itf (remove1 C sub) b s)
```

Similar operations are needed for dealing with requests and results. This includes operations for building lists of all referenced requests inside a component (and

its subcomponents), finding a result for a given future inside a component hierarchy, etc. In all we provide almost 30 functions and predicates to help express structured component specifications efficiently.

Design decisions. It is crucial for the reasoning process whether one chooses lists or sets to represent various parts of the specified component structure. As we have seen above the basic infrastructure we have built up to handle our hierarchical components is mainly based on lists. Consequently, we can define operations over components and their constituents by primitive recursion and thereby decisively improve automated support. However, sets come with a more natural notation. Often set theoretic properties can be simply decided by boolean reasoning that poses no problems for logical decision procedures integrated in Isabelle/HOL, and Isabelle/HOL comes with numerous lemmas for reasoning on sets. On the other side, inductive reasoning on finite sets is less convenient than on lists. In places where we want to combine the merits of both worlds, the `CpSet` function provides a convenient translation.

3.3 Component State

Our component model shall not only allow structural reasoning on hierarchical components but also reasoning about dynamic component state. While the preceding sections provided a good formalisation valid for any hierarchical component model, we now define component state in order to support communication by request and replies. Those constructs are used to define our component semantics, as shown in Section 4.1. Let us first focus on the high level definition of states which provide the constructs relating the component structure with the dynamic semantics[2]. We show below the two types of component states (for composite and primitive components) used in the definition of `Component` presented in Section 3.1.

```
record CompState =                    record PrimState =
Cqueue:: Request list                 Pqueue:: Request list
CcomputedResults:: Result list        PcomputedResults:: Result list
                                      PintState:: intState
                                      behaviour:: Behaviours
```

Each state contains a queue of pending requests, and a list of results computed by this component. Additionally, primitive components have an internal state and a behaviour for encoding the business logic, see below. We use the Isabelle/HOL `record` type constructor here; it automatically defines field projection as functions, e.g. for a `Compstate s`, `(Cqueue s)` accesses its request queue. Note that uniqueness of fields identifier required us to add a '`C`' or '`P`' prefix to fields of component states to distinguish them.

The definition of the component state relies on the definitions of requests (characterized by a future identifier, a parameter, and a target interface), and results (characterized by the future identifier and its value).

[2] The real definition of component states contains additional fields; only the fields of interest for this paper are shown here.

```
record Request =                    record Result =
id::Fid                             fid::Fid
parameter:: Value                   fValue:: Value
invokedItf:: Name
```

An interesting construct is the representation of component behaviour. Each primitive component has an internal state. A behaviour specifies how a primitive component passes from an internal state to another. It is defined as a labeled transition system between internal states of a component:

```
typedef  Behaviours={ beh::(intState × Action × intState) set.
   (∀ s s'. ((s,Tau,s')∈ beh ⟶ (set (PRqRefs s')⊆set (PRqRefs s))
                        ∧ PcurrentReqs s' = PcurrentReqs s)) ∧
 . . . }
```

The type `Behaviours` is defined as a set of triples (internal state, action, internal state). In our case actions are: internal transition (`Tau`, shown here), request service, request emission, result reception, and end of service which associates a result to a request. More than the precise definition of our actions, it is interesting to focus on the way behaviour can be defined and further refined by constraints. Additional rules are specified to restrain the possible behaviours, preventing incorrect transitions to occur; for example, we forbid replying to a non-existing request. In the piece of code above we require conditions on the internal state before and after an internal transition: the set of referenced futures can only be smaller after an internal transition, and the set of currently served requests is unchanged. More complex conditions are imposed for other actions.

Design decisions. Isabelle/HOL extensible records are the natural choice for representing states, requests, and results. They are better suited than simple products because they support qualified names implicitly. We did, however, not use the additional extension property of records which is similar to inheritance known from object-orientation. It could have been used to factor out the shared parts of primitive and composite components but this is not worthwhile – properties specific to the shared parts are few. Hence, there is practically no overhead caused by duplicating basic lemmas. The use of lists for requests and results is important for the efficient specification and proof of structural properties (see the design decisions in the previous section). The definition of behaviours in the internal state of primitive components uses an Isabelle/HOL type definition. This way, we can encapsulate the predicate defining the set of all well-formed behaviours into a new HOL type. These constraints are thereby implicitly carried over and can be re-invoked by using the internal isomorphism with the set `Behaviours`.

3.4 Correct Component

We presented the structure of our components in Section 2.2, while the various constructs designed to manipulate hierarchical components appear in Section 3.2.

However, we only reason on a subset of all possible components that can be constructed according to the described component structure. We refer to this subset of components as *correct components*. Correct components are not only well-formed, but they adhere to some additional constraints. The various well-formedness rules along with the correctness constraints are presented in the following.

We start with specifying the structure of a well-formed component. A composite component is considered as correctly structured if it passes the criteria specified by the function `CorrectComponentStructure` given below.

```
primrec CorrectComponentStructure :: Component ⇒ bool where
 CorrectComponentStructure (Composite N itfs sub b s) =
   ((∀ b ∈ bindings. (GetQualified (src b) (Composite N itfs sub b s =
                          Some (| kind=Client, cardinality=Single |))
    ∧ (GetQualified (dest b) (Composite N itfs sub b s) =
                          Some (| kind=Server, cardinality=Single |)))
    ∧ NoDuplicateSrc b
    ∧ distinct (map getName sub)
    ∧ (∀ Q∈ set (Cqueue s). (invokedItf Q)∈  dom itfs
                      ∧ kind (the (itfs (invokedItf Q))) = Server)
```

A composite component has a correct structure if: each binding only connects an existing client interface to another existing server interface; each client interface is connected only once; all subcomponents have distinct names; and all requests in the request queue of the composite refer to existing server interfaces. A primitive component has a correct structure if it follows the last requirement plus a couple of constraints relating its behaviour with its interfaces.

```
constdefs CorrectComponent :: Component ⇒ bool
CorrectComponent c == CorrectComponentStructure c ∧ distinct(RqIdList c)
        ∧ (ReferencedRqs c) ⊆ (set(RqIdList c))
        ∧ distinct (map getName (cpList c))
        ∧ (∀ f∈ set (RqIdList c). snd f ∈ set(map getName(cpList c)))
```

A correct component is a correctly structured component that also has uniquely defined request identifiers (`RqIdList c` gives all requests computed by `c` and its subcomponents), and all future referenced by the components should correspond to an existing request. Finally, names of all components in the composition should be unique. This differs from the well-formedness requirement which only requires the names of all direct subcomponents to be unique. The requirement of checking correct future referencing throughout the composition hierarchy is stronger than what is needed for most proofs, and can at times be relaxed resulting in a weaker correctness requirement `CorrectComponentWeak`. `CorrectComponentWeakList` gives similar constraints but for a list of components. Using `CorrectComponentWeak` eases proofs involving component hierarchy because if a component verifies `CorrectComponentWeak` then all its subcomponents also verify it.

```
constdefs CorrectComponentWeak:: Component ⇒ bool
CorrectComponentWeak c == CorrectComponentStructure c
    ∧ distinct (RqIdList c) ∧ distinct (map getName(cpList c))

constdefs CorrectComponentWeakList:: Component list ⇒ bool
CorrectComponentWeakList CL == (CorrectComponentStructureList CL)
  ∧ distinct (RqIdListList CL)∧ distinct (map getName (cpListlist CL))
```

3.5 Basic Properties on Component Structure and Manipulation

In this section, we present a few properties that we proved. They deal with the constructs presented in Section 3.2, and are unrelated to our definition of states presented in the last section. Those lemmas are the basic building blocks on which most of our proofs rely. On the set of more than 80 lemmas dealing with `cpSets` and `cpLists`, we focus on the most useful and significant ones. In particular, we choose to show rather lemmas dealing with the `cpSet` construct because it is a higher-level one and thus reasoning on sets of components is often preferable, when possible. Note however that most of the proofs dealing with distinctness of component names will rather use `cpLists`.

We start by an easy lemma quite heavily used and very easy to prove. It states that `C` is always in `cpSet(C)` (it is proved by cases on `C`).

```
lemma cpSetFirst: C ∈ cpSet C
```

The set of components inside a composite one can be decomposed as follows. It can be separated into the composite itself plus all the components in the `cpSet` of each sub-component.

```
lemma cpSetcomposite:
cpSet (Composite N itfs sub b s)={Composite N itfs sub b s}
                                ∪ {C.∃ C'∈set sub. C∈ cpSet C'}
```

This lemma is proved by an induction on lists of subcomponents. Conversely, we can prove that, if a component is in the `cpSet` of a subcomponent of a composite, it is in the `cpSet` of the composite. We also present a more general variant of this lemma stating that if `C''` is inside `C'` and `C'` is inside `C` then `C''` is inside `C`.

```
lemma cpSetcomposite_rev:
⟦ C∈ set sub; C'∈ cpSet C ⟧⟹ C'∈ cpSet (Composite N itfs sub b s)

lemma cpSetcpSet: ⟦C''∈ cpSet C';C'∈cpSet C⟧⟹ C''∈ cpSet C
```

Although those two lemmas are very easy to prove (by induction on the component structure), they are massively used in the other proofs.

Another theorem almost automatically proved by Isabelle, but exceedingly useful is the following one. It gives another formulation of the `getCp` construct.

```
lemma getCp_inlist: CL^N=Some C ⟹ C∈ set CL ∧ getName C=N
```

It is used to relate hypotheses in which a component name occurs and the component name, or the component structure. The reverse direction holds only if the component names inside CL are distinct as shown by the next lemma.

```
lemma getCpIdistinct:
⟦ distinct (map getName CL); getName C=N; C∈ set CL⟧ ⟹ CL^N=Some C
```

As the tools provided for the distinct construct in the Isabelle/HOL framework are a little weaker than for manipulating sets and lists, this proof is slightly longer and less automatic but still quite simple. Finally, the next lemma relates the changeCp primitive with the getCp one for the case that the name of the accessed component and the name of the changed one are different.

```
lemma upd_getCpunchanged: N≠getName C'⟹(CL <- C')^N = CL^N
```

Impact of design choices. As a consequence of the mapping between component structure and Isabelle's structural support, it has been relatively easy to prove properties of component structure by automatic steps plus induction on the component structure. Consequently, the basic proofs on component sets and lists were relatively easy to handle: approximately 700 lines of code for the 80 lemmas dealing with component sets, component lists, and request identifiers, including the getCp, getRecSubCp, and changeSubCp primitives. By contrast, the proofs dealing with the semantics or correctness are generally much longer (several hundreds of lines per proof). However, the structural lemmas presented above are heavily used in the other proofs and strongly facilitate them.

3.6 Properties on Component Correctness

Based on the infrastructure for structural reasoning on the composition structure of components, we can now prove properties on the correctness of component structure presented in Section 3.4. The properties logically relate the degree of correctness of the structure. We present some of these lemmas here.

The lemma CorrectCompWeak establishes the relationship between CorrectComponent and CorrectComponentWeak.

```
lemma CorrectCompWeak: CorrectComponent C ⟹ CorrectComponentWeak C
```

CorrectComponentListComp establishes the correctness of the list of subcomponents given that the parent composite component is correct. Similarly, a member of a weakly correct component list is also weakly correct.

```
lemma CorrectComponentListComp:
     CorrectComponentWeak (Composite N  itfs subCp bindings s)
                                  ⟹ CorrectComponentWeakList subCp
lemma CorrectComponentListComp_rev:
     ⟦CorrectComponentWeakList CL; C∈set CL⟧ ⟹ CorrectComponentWeak C
```

As a consequence, and as mentioned in Section 3.4, weak correctness entails weak correctness of subcomponents. Those lemmas imply that, when proving properties by induction, relying on weak correctness is very convenient as weak correctness can be used as the hypothesis of the recurrence hypothesis.

```
lemma SubComponent_CorrectComponentWeak:
    ⟦C'∈cpSet C; CorrectComponentWeak C⟧ ⟹ CorrectComponentWeak C'
```

The following property expresses a condition entailed in `CorrectComponentWeak`. `C^^N` returns the first subcomponent of `C` having the name `N`. If `C` is a weakly correct component, then there is a single component with that name, and thus the following hold:

```
lemma getRecSubCp_getName:
⟦CorrectComponentWeak C; C'∈ cpSet C⟧ ⟹ C^^(getName C') = Some C'
```

The proof of this property depends on properties on distinct names, and on the lemmas shown in this section and the preceding one.

Impact of design choices. The proofs in Isabelle/HOL are, for the most part of the correctness lemmas, almost automatic: unfolding the definitions, the proofs are mostly solved by applying the automatic tactic `auto`. Yet, these lemmas are important because they precisely relate different correctness conditions and consequently clarify subsequent proofs. They also entail properties of *compositionality*, i.e. what are the properties of a composite with respect to its constituents.

Other properties, like `getRecSubCp_getname` are harder to prove. Their proofs rely strongly on the provided infrastructure for structured components presented earlier in this section. Feasibility and readability of the proofs at the correctness level depends decisively on this clearly structured support with lemmas. Often the amount of automated proof work can be increased by adding our basic lemmas to the simplification sets of Isabelle/HOL.

4 Components at Runtime

4.1 Semantics

The formal semantics of our component model is given by a number of reduction relations defined by a set of inductive rules. These reduction relations along with the formal semantics of our component model appear in [13]; they were informally summarized in Section 2.2. This section illustrates the usefulness of the presented framework to specify and prove properties on the semantics by focusing on one reduction rule and one property. A smoothly working infrastructure of well-designed structural definitions and accompanying lemmas are prerequisite for mechanically proving properties over a structured component semantics.

We define a reduction relation $S \vdash C \rightarrow_R C', RL$ stating that in the component system S, a given component C reduces to a component C'. The list RL is used for specification of reply strategy that is not detailed here. We show below one specific communication rule CommChild, illustrated in Figure 3, and encoding the delegation of requests to a contained subcomponent.

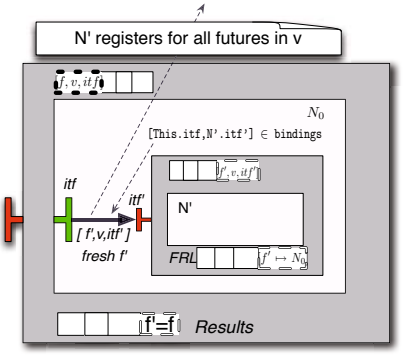


Fig. 3. CommChild rule

```
CommChild:
⟦ Cqueue s= R#Q; (|src=This(invkItf R), dest=N'.i2|) ∈ bindings;
                 f'∉set (RqIdList S) ; subCp^N' = Some C' ⟧ ⟹
S ⊢ Composite N itf sub b s→R Composite N itf
    (sub<-(C'←(|id=f', parameter=(parameter R),invokedItf=i2|))) b
    (s(|Cqueue:=Q,CcomputedResults:=CcomputedResults s @
                                    [(|fid=id R,fValue=(0,[f'])|)]|)),
                          (f,N)#(map (λ id.(id,N')) (snd(parameter R)))
```

The rule expresses request delegation between a composite component N and one of its subcomponents N'. The request R (shown as its constituents $[f, v, itf]$ in Figure 3) that has been sent to the parent N is dequeued from its request queue. A new future f' is created and added to the result list (`CcomputedResults`) of the parent as the result for this request R. A new request (shown as its constituents $[f', v, itf']$) is enqueued in the subcomponent N'. In the Isabelle code snippet, we use the shortcut notation ← for the enqueue operation. The target subcomponent is determined using the bindings: if *This.itf* is bound to $N'.itf'$ then the request is sent to the interface itf' of the subcomponent N', where itf is the external interface of N by which the request had arrived before. Note the use of the getCp primitive: `subCp^N'=Some C'` ensures that subcomponent of name `N'` exists and is `C'`. Also the `changeCp` primitive (`<-`) is quite useful here to update the subcomponent by enqueueing a new request to it.

Let us conclude this section by showing a property we proved in our framework that deals with component semantics. The following lemma shows that the set of names of components inside a component is unchanged by reduction.

```
lemma red_names_eq: ⟦S ⊢ c1 →R c2, RL; CorrectComponentWeak c1⟧
        ⟹ getName ‘ (cpSet c2) = getName ‘(cpSet c1)
```

The proof is approximately 60 lines long, it is done by analysis on the reduction rule. It relies on a few lemmas relating names with reduction rules, and on most of the lemmas presented in Section 3. A crucial auxiliary lemma is the following one that is purely structural and unrelated with our semantics.

```
lemma upd_names_eq:
⟦CL^(getName c2)= Some c1; getName‘(cpSet c2)=getName‘(cpSet c1)⟧
          ⟹ getName‘(cpListset CL) = getName‘(cpListset (CL<-c2))
```

4.2 Reconfiguration

Reconfiguration represents all the transformations of the component structure or content that can be handled at runtime. We consider here mainly structural reconfiguration, which includes changes of the bindings, and of the content of a component. For example replacement of a primitive component by a new one is a form of reconfiguration that allows evolution of the business code.

In Fractal or GCM, configuration primitives are bind/unbind to manipulate bindings, add/remove to change the set of subcomponent of a composite component; also it is possible to start/stop a component.

Our framework enables reasoning on reconfiguration primitives and behaviour of a reconfigured component system. We illustrate below a few encodings of reconfiguration primitives and some theorems that can be proved in Isabelle/HOL thanks to our framework.

We illustrate reconfiguration capacities of our approach by defining two reconfiguration primitives and proving two related lemmas. But beforehand, we define the notion of *complete component*.

Completeness. Similarly to [6], we say that a composite component is complete if all interfaces of its sub-components and all its internal interfaces are bound. This can be easily defined in Isabelle by the following primitive recursive predicate.

```
primrec Complete::Component⇒ bool where
 Complete (Primitive N itf s) = True |
 Complete (Composite N itf sub bindings s) =
   (∀ C∈set sub. allExternalItfsBound C bindings) ∧
   (allInternalItfsBound (Composite N itf sub bindings s) bindings) ∧
   (CompleteList sub)
```

Here, `allInternalItfsBound C b` checks that all external interfaces of `C` are bound by bindings `b`, and `allExternalItfsBound C b` that all internal interfaces of `C` are bound by bindings `b`. Finally, similar to `cpListlist` in Section 3.2, `CompleteList` recursively checks that all subcomponents are complete.

As there is no notion of optional interface in our model, this definition is really straightforward. For a complete component, any request emitted by a component will arrive at a destination component.

Unbind primitive. The unbind primitive removes one of the bindings defined by a composite component.

```
primrec unbind:: Component⇒Binding⇒Component where
  unbind (Primitive N itf s) b = (Primitive N itf s) |
  unbind (Composite N itf sub bindings s) b =
           (Composite N itf sub (bindings-{b}) s)
```

Of course, un-binding does not maintain completeness, and this can be proved in our framework.

```
lemma unbinding_incomplete:
⟦b∈bindings; CorrectComponentStructure (Composite N itf sub bindings s)⟧
        ⟹ ¬ Complete (unbind (Composite N itf sub bindings s) b)
```

This lemma is proved in only 35 lines of simple Isabelle/HOL code, thanks to the properties presented in Section 3.5. The proof can be sketched as follows. `CorrectComponentStructure` imposes that in bindings `src b` is connected only once, thus, in `bindings-{b}`, `src b` is not connected anymore. Now, `src b` can be either `This N` if `b` connects an internal client interface to a sub-component, or of the form `CN.N` if it connects a sub-component to another interface. In the first case, the new component does not ensure `allInternalItfsBound` anymore, and in the second case, it is `allExternalItfsBound` that is not true for the component with name `CN`; note that `CorrectComponentStructure` ensures the existence of such a component.

Component replacement. Let us now introduce a reconfiguration primitive that would automatically maintain completeness.

```
primrec Replace:: Component⇒Name⇒Component⇒Component where
Replace (Primitive N itf s) N1 C = (Primitive N itf s) |
Replace (Composite N itf sub binds s) N1 C = addSubCp (removeSubCp
(Composite N itf sub ((λb.RenameBinding b N1 (getName C))‘binds) s) N1) C
```

This primitive maintains completeness of a correct component as expressed in the following lemma:

```
lemma replace_complete:
 ⟦sub^(getName C')=None; sub^N'=Some oldC; getItfs oldC=getItfs C';
  Complete C'; Complete (Composite N itf sub bindings s);
  CorrectComponentStructure C';
  CorrectComponentStructure (Composite N itf sub bindings s)⟧
     ⟹ Complete (Replace (Composite N itf sub bindings s) N' C')
```

This lemma requires that all involved original components are correct and complete, that the replaced component is in the composition, but not the replacement one, and that those two components have the same interfaces. A similar lemma proving `CorrectComponentStructure` for the result of the replacement operation is also proved.

Of course, the replace primitive can be expressed by lower level reconfiguration operations, i.e. an unbind, remove, add, bind sequence. A lemma equivalent to the preceding one could also be proved. Such a lemma would be more general but a little more complex to express because it would need to relate the set of unbound bindings, the set of re-bound ones, and the component involved in the add-remove operations.

5 Conclusion

This paper presented the logical machinery of a mechanized framework for reasoning about structured component systems; especially targeting distributed components. We have first illustrated and motivated the specification of components and the provided proof infrastructure. Furthermore, we have shown this machinery in action by showing how reconfiguration of components can be formally specified, and how properties over component structure and reconfiguration can be handled. This paper also illustrated our approach by showing the specification of a semantics for components, and associated proofs. Overall, the developed framework consists of more than 4000 lines, including almost 300 lemmas and theorems, approximately 500 lines for defining the component model and its semantics, and 1800 lines focusing on properties specific to future registration which were not presented here. As usual with mechanised proofs, the main difficulty is the choice of the right structures providing the suitable level of abstraction. Some proofs are lengthy and technical but no major difficulty was encountered.

In contrast to existing works, our approach focuses on increasing confidence in global properties of component models. For this, we provide a framework and apply it to prove generally valid results. The established infrastructure of structured components with asynchronous communication provides an elegant abstraction from implementation detail while fully preserving the communication structure and defining a precise semantics. One limiting factor of our framework is that a precise semantics for components had to be chosen to allow mechanised proofs. Overall we have developed a reliable basis for the mechanical proofs of properties of hierarchical component models, and we have shown its adequacy to deal with first proofs entailing reconfiguration, or component semantics. We additionally provide subsequent support for distributed components communicating by asynchronous requests with futures.

A promising follow up project would be to analyse information flows based on this model, or properties entailing component synchronisation at reconfiguration time. More generally we expect to prove properties on reconfiguration that will entail reasoning simultaneously on component execution and on evolution of component structure. This would show the correctness of complex adaptation procedures that can be applied in autonomous component systems.

References

[1] Sensoria – software engineering for service-oriented overlay computers (2005)
[2] Ábrahám, E., Grabe, I., Grüner, A., Steffen, M.: Behavioral interface description of an object-oriented language with futures and promises. Journal of Logic and Algebraic Programming 78(1-2), 491–518 (2008)
[3] Barros, T., Ameur-Boulifa, R., Cansado, A., Henrio, L., Madelaine, E.: Behavioural models for distributed fractal components. Annales des Télécommunications 64(1-2), 25–43 (2009)
[4] Baude, F., Caromel, D., Dalmasso, C., Danelutto, M., Getov, V., Henrio, L., Pérez, C.: GCM: A Grid Extension to Fractal for Autonomous Distributed Components. Annals of Telecommunications (2008) (accepted for publication)

[5] Beisiegel, M., Blohm, H., Booz, D., Edwards, M., Hurley, O.: SCA service component architecture, assembly model specification. Technical report (March 2007), www.osoa.org/display/Main/Service+Component+Architecture+Specifications
[6] Bruneton, E., Coupaye, T., Stefani, J.B.: The Fractal Component Model. Technical report, ObjectWeb Consortium (February 2004), http://fractal.objectweb.org/specification/index.html
[7] Bruni, R., et al.: Service oriented architectural design. In: Barthe, G., Fournet, C. (eds.) TGC 2007 LNCS, vol. 4912, pp. 186–203. Springer, Heidelberg (2008)
[8] Cámara, J., Salaün, G., Canal, C., Ouederni, M.: Interactive Specification and Verification of Behavioural Adaptation Contracts. In: Ninth International Conference on Quality Software, pp. 65–75 (August 2009)
[9] Canal, C., Poizat, P., Salaün, G.: Synchronizing behavioural mismatch in software composition. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 63–77. Springer, Heidelberg (2006)
[10] Cansado, A., Madelaine, E.: Specification and verification for grid Component-Based applications: From models to tools. In: Formal Methods for Components and Objects, pp. 180–203 (2009)
[11] CCA-Forum. The Common Component Architecture (CCA) Forum home page (2005), http://www.cca-forum.org/
[12] Grabe, I., Steffen, M., Torjusen, A.B.: Executable interface specifications for testing asynchronous creol components. Technical Report Research Report No. 375, University Of Oslo (July 2008)
[13] Henrio, L., Kammüller, F., Rivera, M.: An asynchronous distributed component model and its semantics. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, Springer, Heidelberg (2009) (to appear)
[14] Henrio, L., Khan, M.U.: Asynchronous components with futures: Semantics and proofs in isabelle/hol. In: Proceedings of the Seventh International Workshop, FESCA 2010. ENTCS (2010) (to appear)
[15] Broch Johnsen, E., Owe, O.: An asynchronous communication model for distributed concurrent objects. In: Proceedings of the Software Engineering and Formal Methods, SEFM 2004, Washington, DC, USA, pp. 188–197. IEEE Computer Society Press, Los Alamitos (2004)
[16] Broch Johnsen, E., Owe, O., Yu, I.C.: Creol: a type-safe object-oriented model for distributed concurrent systems. Theor. Comput. Sci. 365(1), 23–66 (2006)
[17] Khan, M.U., Henrio, L.: First class futures: a study of update strategies. Research Report RR-7113, INRIA (2009)
[18] Merle, P.B., Stefani, J.B.: A formal specification of the Fractal component model in Alloy. Research Report RR-6721, INRIA (2008)
[19] Meseguer, J.: Conditional reqriting logic as a unified model of concurrency. Journal of Theoretical Computer Science 96, 73–155 (1992)
[20] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. In: Isabelle/HOL. LNCS, vol. 2283, Springer, Heidelberg (2002)
[21] OW2.Consortium. FraSCAti, Open SCA middleware platform (2009), https://wiki.objectweb.org/frascati/Wiki.jsp?page=FraSCAti

3.5 Algorithmic skeletons

With Mario Leyton, we performed several works on the formalisation of algorithmic skeletons.

Algorithmic skeletons (*skeletons* for short) are a high level programming model for parallel and distributed computing, introduced by Cole in [Col91]. Skeletons take advantage of common programming patterns to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by nesting the basic ones. All the non-functional aspects regarding parallelisation and distribution are implicitly defined by the composed parallel structure. Once the structure has been defined, programmers complete the program by providing the application's sequential blocks, called *muscle* functions. Here is an example algorithmic skeleton: $farm(pipe(\Delta_1, \Delta_2))$. This skeleton consist in a farm of skeletons (that can be replicated to be able to treat a stream of input data), treatment of an input consists in executing a sequence of two instructions Δ_1 and Δ_2, that can be muscle functions or other skeletons.

Mario Leyton has developed two different algorithmic skeleton frameworks: Calcium featuring distributed computation, and Skandium dedicated to multi-core programming. During those works, we worked together at the formal specification of the new features of the skeleton frameworks. We worked on a type system for algorithmic skeletons, on exception handling for skeletons, and on the specification of the compilation of algorithmic skeletons into lower-level skeletons easier to interpret efficiently. The last work was published in Mario Leyton's PhD thesis [Ley08] and is not presented here.

Our skeletons include task parallel skeletons: *seq* for wrapping execution functions; *farm* for task replication; *pipe* for staged computation; *while/for* for iteration; and *if* for conditional branching. There are also data parallel skeletons: *map* for single instruction multiple data; *fork* which is like *map* but applies multiple instructions to multiple data; and *d&c* for divide and conquer.

3.5.1 Typing algorithmic skeletons

In [35] we designed a type system for skeletons. It specifies the way types are transmitted between muscle functions: depending on their semantics, different skeleton patterns impose typing rules on their sub-elements. Typically a pipe skeleton, $pipe(\Delta_1, \Delta_2)$, imposes the return type of Δ_1 to be the input type of Δ_2; also the input type of the pipe is the one of Δ_1 and its return type is the one of Δ_2.

We specified in this work a big step operational semantics for algorithmic skeletons, and typing rules for skeletons, those typing rules express types transmitted between sub-skeletons as explained above. We then proved the classical and crucial subject-reduction property: types are preserved during reduction. This shows the correctness of our type-system.

On the practical side, we extended our Java skeleton libraries with type annotations, using Generics. The type enforcements are ensured by the Java type system, and reflect the typing rules introduced in the theoretical section.

Overall, this work relieves the programmer from relying on type-casts at the entry of each muscle functions, which is the classical way of writing skeletons, at least in Java. Thus compared to existing frameworks, in our skeleton libraries *type errors can be detected when composing the skeleton programs*. We removed type-casts at the entrance of muscle functions, thus ensuring the absence of type-cast errors at these points during execution.

3.5.2 Exception handling in algorithmic skeletons

In [46] we designed a model for the management of exceptions in algorithmic skeletons. In our model, exceptions can be raised and handled at each level of the skeleton nesting structure.

Each skeleton can be attached handlers capable of catching errors and return regular results, else they can raise the exception to be handled by the parent skeleton. We provide the programmer with an API for attaching handlers to skeletons. Additionally, the raised exceptions are dynamically modified to reflect the nesting of skeleton patterns instead of the nesting of method calls of the skeleton interpreter.

We extended the semantics for skeletons published in [Ley08] with exception handling. The semantics works as follows. The skeleton program Δ_2 is transformed into a lower level representation formed of instructions, with its corresponding muscle functions and exception handlers. Transforming high-level skeletons to lower-level constructs is a standard methodology in many skeleton frameworks such as P3L, Lithium, Muesli, QUAFF, etc. On these lower-level instructions the parallel semantics is expressed, by operations on instruction stacks. The reduction of a skeleton consists in evaluating a set of stacks: the first instruction of each stack is evaluated until a result is obtained, then the result is passed to the next instruction of the task. Then, when an exception is raised by a muscle, the surrounding

skeleton either handles the exception and produces a regular result, or raises the exception to the parent skeleton.

We applied this approach to the Skandium library. In order to produce understandable stack traces, the skeleton nesting is remembered during the interpretation of the skeleton program. First the programmer can now attach exception handlers to skeletons. Second, we intercepted the stack trace output so that the output trace produced in case of uncaught exception expresses the nesting of skeletons that raised the exception instead of a Java trace referencing code not written by the programmer and without any information on the skeleton nesting.

These works on algorithmic skeletons were a very nice opportunity to show the complementarity between formal methods, and language and middleware implementation. They allowed us to contribute to the algorithmic skeleton community with extensions that revealed both useful for the programmer, and formally specified and proved safe.

3.6 Behavioural specification and verification of GCM components

Safety of component-oriented and object-oriented applications

Distributed systems have by nature a complex behaviour. Even if the programming methodology entailed by active objects is way simpler that RMI-style of programming, bugs are still more frequent in any distributed applications than in sequential ones. Indeed, even if the programming models we presented in the previous sections of this document prevent data race-conditions, race-conditions between communications can still exist as in any distributed system and, except for functional active objects, deadlocks cannot be prevented in the general case. The complex interleaving of communications makes the reasoning on a distributed system difficult, even when the system is built from well separated components.

Our objective is to guarantee the safe execution of distributed applications, for this we provided tools to verify the behaviour of distributed active objects and GCM components. This work is complementary with the formalisation of the GCM presented in Section 3.4 or the formalisation of active objects presented in Chapter 2: while Section 3.4 presented a framework to prove property on the component model, and its behaviour at runtime, our behavioural specification allows the programmer to prove the correct behaviour of his/her applications, most probably relying on the properties proved on the component model. For example, the fact that futures in ASP can be returned at any time without changing the result of the computation allows us to generate a behavioural model where the future updates is delayed as much as possible in order to reduce the interleaving between future updates and the other actions, and thus to minimise the size of the system to verify.

Related work

The closest work to ours are the frameworks dedicated to the verification of behavioural properties on component applications, we focus below on a detailed comparison with two approaches: SOFA and Symbolic Transition Systems. A more exhaustive study of related works and positioning can be found in our journal paper [8] or in [Mad11].

The SOFA system [BHP06] is a development and verification framework for large-scale distributed software systems based on hierarchical components. It uses *behaviour protocols* [PV02] to specify interactions between components in terms of ordering of method invocation events. The behaviour compliance and consent relations are defined on behaviour protocols based on their trace semantics, allowing reasoning on substitutability and compositional compatibility. The *frame protocol* defines the behaviour of a component. In a composite component, the behaviour is constructed from frame protocols of its subcomponents, and checked for compliance with the composite frame protocol. For a primitive component, the Java implementation may be checked for compliance with a model checking tool [PP06].

Symbolic Transition Systems (STS) [PRS06, PR06], are structures akin to our pNets. In the STSLib toolset, there is a dedicated specification language (with abstract data types) for distributed components, that are modelled by STS, themselves mapped to LOTOS programs that can be model-checked with the CADP verification toolset [GLM02]. STS do not use the distinction between required and provided ports (or interfaces), whereas it is one of the main building blocks of our component systems. In fact, communication is not based on the classic notion of method calls, but on messages in which both parties (emitter and receiver) must agree in order to communicate. Although this adds expressivity to the language, it also has an impact on the asynchrony of the system. The protocols are expressed in terms of STS. On the implementation side, the two approaches are quite different: the implementation of STS simulates the synchronisation vectors that can be expressed in the specification, whereas in our approach, we write only the synchronisation vectors corresponding to the semantics of the ASP calculus (and of the ProActive library). Our

specification language is more independent from the middleware, it allows us to express complex synchronisations that cannot happen in ProActive. This allows us to reason on efficient, expressive, and proved communication mechanisms. Overall, even if pNets formalism is approximately at the same level of abstraction as STS, in our approach, the programmer is rather exposed to a higher-level composition framework, closer to his usual programming and composition concerns.

Objectives and contribution

In this work, we focused on the behavioural specification of active object and GCM applications in order to be able to verify their behaviour. The behavioural model we generate is expressed in the pNets formalism [8] that we designed and that is described in Section 3.6.1. pNets serves as a low level semantic framework for expressing the behaviour of various classes of distributed languages, and as a common internal format for our tools.

Our verification tool is called Vercors[3]. It is a platform that assists the programmer in the specification and the verification of his application. It provides tools for specifying an application behaviour: from the behaviour of each service method of the primitive components and a description of the application architecture, the platform is able to generate the behaviour of the whole application. We generate automatically the behaviour for asynchronous communication, queues, futures, and component composition based on the ADL.

Then, from the behavioural model of the application, we are able to verify its properties. The properties we aim at verifying range from absence of deadlocks, to reachability of some actions, and to "any" temporal property (safety, liveness) specific to the application. Our approach consists in specifying the property to be verified as regular μ-calculus formula [Koz85] or more recently as MCL (Model Checking Language) logics [MT08] formula. Currently, properties are verified using the CADP toolbox [GLM02], but other verification engines can be considered.

In the future, we would like to specify these properties at a higher-level, which would be subject to the same abstractions as the tools we provide to the programmer. This encompasses first push-button properties that can be verified automatically on each application like absence of dead-lock. Second, generic properties easy to generate from our tools like reachability of an event: the reception of a communication, the emission of a result, ... should also be considered. Also, we should provide support for the expression of more complex properties; they must be expressed with the same abstractions (names, range for parameters, ...) as in the specification tools, which is not yet the case.

[3] http://www-sop.inria.fr/oasis/index.php?page=vercors

The following of this section describes a simple version of the pNets formalism (Section 3.6.1), and then shows how we used it to express behaviour of active object-based (Section 3.6.2) and component-based (Section 3.6.3) applications. Then we include an article on the modelling of first-class futures (Section 3.6.4), consequently we will not focus on futures in the other sections. Finally, this section presents our recent work on a behavioural model for group communications and multicast interfaces in Section 3.6.5.

3.6.1 The pNets formalism

Our work on behavioural specification and verification relies on the pNet model as an intermediate specification language: pNets allow the specification of parameterised hierarchical labelled transition systems: classical labelled transition systems can be combined hierarchically, and parameterised by some variables. From such a specification, several verification techniques can be envisioned. In most cases, we chose a finite instantiation domain for the parameters of the pNets, and generated a flat finite labelled transition system on which we can prove the properties of the application by state-of-the art model-checkers. We describe below the principles and the semantics of pNets in a slightly simplified way compared to our previous descriptions [8].

Syntax and notations

In the following definitions, we extensively use indexed structures (maps) over some countable indexed sets. The indexes will usually be integers, bounded or not. Such an indexed family is denoted as follows: $a_i^{i\in I}$ is an indexed family of a_i. Such a family is equivalent to the mapping $(i \mapsto a_i)^{i\in I}$. To specify the set over which the structure is indexed, indexed structure are always denoted with an exponent of the form $i \in I$ (arithmetics only appear in the indices). For example $a^{i\in\{3\}}$ is the mapping with a single entry a at index 3; exceptionally, such mappings with only a few entries will also be denoted $(3 \mapsto a)$ in the following. When this is not ambiguous, we shall use abusive vocabulary and notations for sets, and typically write "indexed set over J" when formally we should speak of multi-sets, and still better write "$x \in A_i^{i\in I}$" to mean $\exists i \in I.\, x = A_i$. An empty family is denoted $[] = a_i^{i\in\emptyset}$.

Term algebra

Our models rely on the notion of parameterised actions. We leave unspecified the constructors of the algebra that will allow building actions and expressions used in our models, let us denote Σ the signature of those constructors. Let $\mathcal{T}_P$ be the term algebra of Σ over the set of variables P. We suppose that we are able to distinguish inside $\mathcal{T}_P$ a set of *action terms* (over variables of P) denoted $\mathcal{A}_P$ (*parameterised actions*), a set of *expression terms* (disjoint from actions) denoted $\mathcal{E}_P$, and, among expressions, a set of *boolean expressions* (guards) $\mathcal{B}_P$. For each term $t \in \mathcal{T}_P$ we define $fv(t)$ the set of free variables of t. For $\alpha \in \mathcal{A}_P$ we also suppose that there is a function $iv(\alpha)$ that returns a subset of fv which are the input variables of α, i.e. the variables newly defined by reception of their value during the action α.

We also allow countable indexed sets to depend upon variables, and denote $\mathcal{I}_P$ the set of indexed sets using variables of P. There must exist an inclusion relationship $\subseteq$ over the indexed sets of $\mathcal{I}_P$, with the natural guarantee that this operation ensure set inclusion when one replaces variables by values. In practice we will mostly use intervals for which the upper bound depends on the variables of P.

For example the actions of Milner's *Value-passing CCS* [Mil89] correspond to the following algebra: terms are τ, $a(x)$ for input actions, $\overline{a}(v)$ for output actions. Then $fv(a(x)) = iv(a(x)) = \{x\}$, whereas $iv(\overline{a}(x)) = \emptyset$.

The pNets model

A pLTS is a labelled transition system with variables; a pLTS can have guards and assignment of variables on transitions. Variables can be manipulated, defined, or accessed inside states, actions, guards, and assignments. A pLTS is formally defined as follows.

Definition 1 (pLTS) *A parameterised LTS is a tuple* $pLTS \triangleq \langle P, S, s_0, L, \rightarrow \rangle$ *where:*

- P *is a finite set of parameters, from which we construct the term algebra* $\mathcal{T}_P$*, with the parameterised actions* $\mathcal{A}_P$*, the parameterised expressions* $\mathcal{E}_P$ *and the* boolean expressions $\mathcal{B}_P$*.*
- S *is a set of states; each state* $s \in S$*. Variables of* s *are global to the pLTS.*
- $s_0 \in S$ *is the initial state,*
- L *is the set of labels of the form* $\langle \alpha,\ e_b,\ (x_j := e_j)^{j \in J} \rangle$*, where* $a \in \mathcal{A}_P$ *is a parameterised action,* $e_b \in \mathcal{B}_P$ *is a guard, and the variables* $x_j \in P$ *are assigned the expressions* $e_j \in \mathcal{E}_P$*. variables in* $iv(\alpha)$ *are assigned by the action, other variables can be assigned by the additional assignments.*
- $\rightarrow$ *is the transition relation* $\rightarrow \subseteq S \times L \times S$

pNets are constructors for hierarchical behavioural structures: a pNet is formed of other pNets, or pLTSs at the bottom of the hierarchy tree. Message queues can also appear in leaves of a pNet system. A composite pNet consists of a set of pNets exposing a set of actions, each of them triggering internal actions in each of the sub-pNets. The synchronisation between global actions and internal actions is given by a *synchronisation vector*: a synchronisation vector synchronises one or several internal actions, and exposes a single resulting global action.

Definition 2 (pNets) *A pNet is a hierarchical structure where leaves are pLTSs (or queues defined below):*

$$pNet \triangleq pLTS \mid Queue(M) \mid \langle P, L, pNet_i^{i \in I}, SV_k^{k \in K} \rangle$$

where

- P *is a finite set of parameters, from which we construct the term algebra* $\mathcal{T}_P$*, with parameterised actions* $\mathcal{A}_P$*.*
- $L \subseteq \mathcal{A}_P$ *is the set of labels of global actions of the pNet.*
- $I \in \mathcal{I}_P$ *is the set over which sub-pNets are indexed.*
- $SV_k^{k \in K}$ *is a set of synchronisation vectors* ($K \in \mathcal{I}_P$)*.*

$$\forall k \in K, SV_k = \alpha_j^{j \in J_k} \rightarrow \alpha'_k$$

Each synchronisation vector verifies: $\alpha'_k \in L$*,* $J_k \in \mathcal{I}_P$*,* $J_k \subseteq I$*, and* $\forall j \in J_k.\ \alpha_j \in \mathrm{Sort}(pNet_j)$*.*

For each pNet, we define a function sort ($Sort : pNet \rightarrow \mathcal{A}_P$*). The sort of a pNet is its signature: the set of actions a pNet can perform, that is to say the set of labels of its transitions, more formally:*

$$\mathrm{Sort}(\langle P, S, s_0, L, \rightarrow \rangle) = L \qquad \mathrm{Sort}(\langle P, L, pNet_i^{i \in I}, SV_k^{k \in K} \rangle) = L$$

$$\mathrm{Sort}(Queue(M)) = \{?Q\_M_i | M_i \in M\} \cup \{!Serve\_M_i | M_i \in M\}$$

Queues

We also define a particular pNet called $Queue(M)$; it models the behaviour of a FIFO queue. It can be considered as an infinite pLTS with a set of actions depending on the chosen term algebra and of the set of enqueue-able elements $M \subseteq \mathcal{T}_P$. We suppose then that the term algebra has two specific constructors $?Q$ and $!Serve$ such that for all set of variables P, $\forall M_i \in M.\ !Serve\_M_i \in \mathcal{A}_P \wedge ?Q\_M_i \in \mathcal{A}_P$. Then the queue pNet offers the following actions:

$$L = \{?Q\_M_i | M_i \in M\} \cup \{!Serve\_M_i | M_i \in M\}$$

Whenever pNets will be encoded by (ultimately finite) automata structures for model-checking, pNet Queues will

naturally be represented by finite automata. However, in order to be able to address more general approaches, and in particular specific model-checking algorithms for unbounded channels, we keep a high-level representation of queues. From our abstract queues, we will be able to generate both regular representation (for unbound queues), and finite representation (for explicit-state model-checking).

Families of pNets

We define a constructor for a pNet made of an indexed family of pNets. $\overleftrightarrow{PN_i^{i\in I}(P)}$ takes a family of pNets indexed over a set $I \in \mathcal{I}_P$ and produce a global pNet. The synchronisation vectors for this family will be expressed at the level above, consequently we "export" all the possible synchronisation vectors that the family could offer, only some of them will be used.

$$\overleftrightarrow{PN_i^{i\in I}(P)} \triangleq \langle P, SV, PN_i^{i\in I}, \{\alpha_j^{j\in J} \to \alpha_j^{j\in J} | \alpha_j^{j\in J} \in SV\}\rangle$$
$$\text{where } SV = \{\alpha_j^{j\in J} |\, J \subseteq I \wedge \forall j \in J.\, \alpha_j \in \mathrm{Sort}(PN_j)\}$$

This supposes that the elements of SV belong to the term algebra and more precisely are action terms.

In fact, what we show here is a version of pNets that is convenient for providing a concise formal definition of both pNets themselves and the component specification in terms of pNets. In practice it is not reasonable to define all the possible synchronisation vectors possible inside a family, and only the used ones are instantiated. In [8], we defined a version of pNets closer to what we use in practice where the families are flattened in the enclosing pNet. Though more efficient this notation was more complex, that is why a simpler definition is presented here. The two representations provide anyway the same expressive power.

An operational semantics for pNets

To give a semantics to pNets, we need a unique *valuation domain* $\mathcal{D}$. This domain could consist in a countable instantiation domain for each variable, but not necessarily, the only constraint we have on $\mathcal{D}$ is that it should be possible to decide whether a boolean expression in $\mathcal{D}$ is true, and to decide whether two expressions have the same value (e.g. when two action labels are the same). In fact we could remove the last hypotheses and instead check whether True is a possible value of a boolean expression in the rules below (or whether two expressions can have the same value) but this would make the semantics more complex. If we choose a finite domain for each variable and if each pLTS has a finite set of states, the semantics of the pNet will be a finite LTS that can be used in a model-checker.

We let $\phi = \{x_j \to V_j | j \in J\}$ be a valuation function where x_j range over variables of the considered pNet (each variable must be given a value), and $V_j \in \mathcal{D}$. Such a valuation acts as a store, maintaining a mapping from variables to values. For a term $t \in \mathcal{T}_P$, $t\phi \in \mathcal{D}$ is the value of the term replacing each variable by their values given by ϕ. A valuation can be applied to expressions, actions, or even indexed sets. In all cases, the variables are replaced by their value and the new expressions are evaluated. The set of valuation functions, Φ, allows the precise definition of the state-space to be considered for the behaviour of the system: only valuation functions such that $\phi \in \Phi$ are considered. We define an update operator $+$ on valuations, where $\phi_1 + \phi_2(x)$ is $\phi_2(x)$ if it is defined, or $\phi_1(x)$ else.

Note that variables are used locally to each pNet/pLTS, it is thus possible to use qualified names to avoid collision of variable names in the valuation, thus we consider that variable names are unique.

Consider a pNet *pNet*and an initial valuation $\phi_0 \in \Phi$ associating a value to each variable of the pNet. The semantics of *pNet* is given by a LTS (or possibly a pLTS if $\mathcal{D}$ contains variables) where:

- states are hierarchical composition of product states of the sub-pNets, more precisely states are $S(pNet)$ where:

$$S(\langle P, S, s_0, L, \to\rangle) = \{(s, \phi) | s \in S \wedge \ \phi \in \Phi\}$$
$$S(\langle P, L, pNet_i^{i\in I}, SV_k^{k\in K}\rangle) = \{\langle s_i\rangle^{i\in I\phi} | \forall i \in I.\, s_i \in S(pNet_i),\ \phi \in \Phi\}$$
$$S(Queue(M)) = \{(M_j\phi_j)^{j\in[1..n]} | n \in \mathbb{N} \wedge \forall j.\, (M_j \in M \wedge \phi_j \in \Phi)\}$$

- labels are $\{\alpha\phi | \alpha \in \mathrm{Sort}(pNet),\ \phi \in \Phi\}$;
- the initial state is the composition of initial states of sub-pNets, the initial state is $S_0(pNet)$ where:

$$S_0(\langle P, S, s_0, L, \to\rangle) = (s_0, \phi_0)$$
$$S_0(\langle P, L, pNet_i^{i\in I}, SV_k^{k\in K}\rangle) = \langle S_0(pNet_i)\rangle^{i\in I\phi_0}$$
$$S_0(Queue(M)) = []$$

- and transitions are $[\![pNet]\!]$ such that:

$$\frac{\begin{array}{c}\phi \in \Phi \qquad k \in K\phi \\ \alpha_j^{j\in J} \to \alpha \in SV_k \qquad \forall j \in J\phi.\, \phi_j \in \Phi \wedge s_j \xrightarrow{\alpha_j\phi_j} s'_j \in [\![pNet_j]\!] \\ \forall i \in I\phi \setminus J\phi.\, s'_i = s_i\end{array}}{\langle s_i^{i\in I\phi}\rangle \xrightarrow{\alpha\phi} \langle s_i'^{i\in I\phi}\rangle \in [\![\langle P, L, pNet_i^{i\in I}, SV_k^{k\in K}\rangle]\!]}$$

$$\frac{\begin{array}{c}\phi\in\Phi \qquad s \xrightarrow{\langle \alpha,\ e_b,\ (x_j := e_j)^{j\in J}\rangle} s' \in \to \qquad iv(\alpha) = \{x'_i |\, i \in K\} \\ \forall i \in K.\, V_i \in \mathcal{D} \qquad \phi' = \phi + \{x'_i \to V_i | i \in K\} \\ e_b\phi' = True \qquad \phi'' = \{x_j \to e_j\phi' | j \in J\}\end{array}}{(s, \phi) \xrightarrow{\alpha\phi'} (s', \phi' + \phi'') \in [\![\langle P, S, s_0, L, \to\rangle]\!]}$$

$$\frac{n \in \mathbb{N} \qquad \forall j \in [1..n+1].\, M_j \in M \wedge \phi_j \in \Phi}{(M_j\phi_j)^{j\in[1..n]} \xrightarrow{?Q\_M_{n+1}\phi_{n+1}} (M_j\phi_j)^{j\in[1..n+1]} \in [\![Queue(M)]\!]}$$

$$\frac{n \in \mathbb{N} \qquad \forall j \in [1..n].\, M_j \in M \wedge \phi_j \in \Phi}{(M_j\phi_j)^{j\in[1..n]} \xrightarrow{!Serve\_M_1\phi_1} (M_{j+1}\phi_{j+1})^{j\in[1..n-1]} \in [\![Queue(M)]\!]}$$

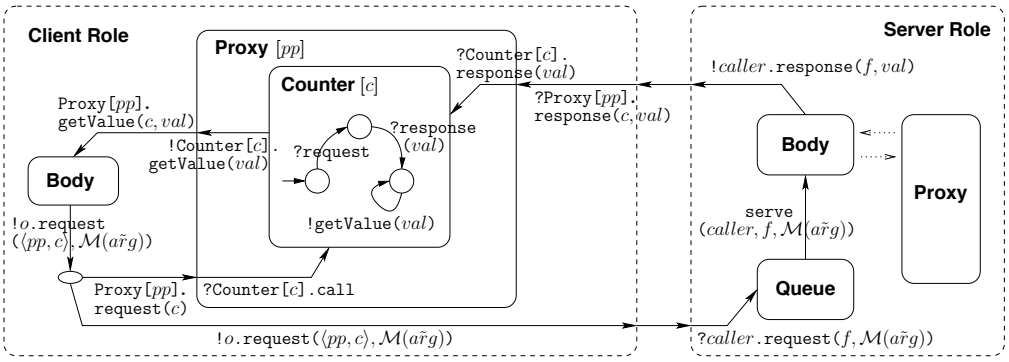


Figure 3.6: Communication between two active objects [8]

The most complicated part of the semantics is the way variables are dealt with in the pLTS: only input variables, and assigned variables are allowed to change value in the valuation function. This also applies (indirectly) to the queue where the valuation used in the action is in the *Serve* case constrained by the source state, and unconstrained in the case of a Q transition.

3.6.2 A behavioural model for active objects

Based on pNets, we first defined a behavioural model for active objects. This model is crucial because it gives a representation, on the form of pNets of most of the features of ASP:

- *Request queues*, using Queues of pNets, and then instantiated by queues of finite length. In our verified applications, a queue of small length (typically 1 or 2) is sufficient to verify the properties of interest; we also verify that the limit of the queue is never reached to ensure that all the behaviours are explored.
- *A body* serving requests one after the other, and calling an appropriate method.
- *Service methods* are supposed to be specified by the programmer, they define the business code of the active object. Of course, an abstract behaviour of the active object should be specified in order to limit the state-space to be explored; it should however be precise enough to allow the verification of all the properties of interest. In particular it should at least contain all the synchronisations between active objects, i.e. request calls and future accesses.
- *Futures* are encoded as proxies, and dedicated communications to return the future values are specified by the synchronisation vectors. In most of our developments, first-class futures were not considered but Section 3.6.4 details our investigations on the treatment of first-class futures in our tools.
  For dealing with futures, we create a family of proxies synchronised first on an initialisation phase upon method call, then with the return of the result at request completion, and finally with the access to the future value upon a wait-by-necessity.

Figure 3.6 illustrates the main pNets involved in the active object behavioural model. It also shows the synchronisation occurring between those different pNets (each synchronisation vector roughly corresponds to one arrow in the drawing). The oval shape is used to represent synchronisations involving more than two processes.

Except for service methods, we defined how to generate automatically and generically those pNets. We are in the process of formally specifying those pNet generators (in the context of components as described below), but also we have implemented a significant part of those generators. This way we are able to specify a pNets model for an application made of active objects *à la* ASP.

Then abstractions are applied on the data domains, yielding a finitary model. Finally the model is encoded using a combination of several input formalisms from the CADP toolset [GLMS11]: the Fiacre language [BBF+07] provides syntax for data types and expressions, definition of LTS, and a form of composition of processes by synchronisation on channels; the EXP and SVL languages [GLMS11] support the hierarchical encoding of our pNets, and the scripting of the various verification tasks. More details on the way we perform those generation and verification steps can be found in [42].

3.6.3 A behavioural model for GCM

In this section we describe how we build behavioural models for GCM components. First building models for hierarchical component systems like GCM allows us to adopt

a compositional approach, at least from the model specification point of view: we generate models hierarchically in each component. Even more, if the interface behaviour is further specified, that is to say if the context in which the component will be used is explicitly stated, the behaviour of each component can be generated and reduced, allowing us to envision the exhaustive verification of larger systems.

In the following we first explain how the structural informations provided by a component oriented approach help the construction of behavioural models. Then we briefly present the Vercors platform. Finally, we explain the principles of behavioural model generation for components based on an example of composite component.

Inference vs. generation of behavioural specification

We sketched above how to build the behaviour of an active object. But building such a model relies on several informations that could be either given by the programmer or inferred from the program. Those informations are the behaviour of service methods, the identification of active objects, future objects and future access points, and the definition of finitary domain for the instantiation of each variable. Three approaches can be envisioned for obtaining those informations: direct specification of the behaviour, static analysis of source code (Java in the case of GCM/ProActive), or writing code in a language dedicated to program specification from which both the program code and the behaviour can be generated.

In general, a specific static analysis or annotations are necessary to know which objects in a program are active, and where futures can be created or awaited. Deciding which objects are active objects or contain a future is in general not decidable. Finding wait-by-necessity instructions is also not decidable, but an over-approximation of the set of wait-by-necessity instructions is sufficient in our case, and for most analyses. Note however that this information is syntactically given in less transparent languages like Creol or JCoBox where asynchronous method calls and future access have a specific syntax.

GCM components provide a convenient abstraction level for behavioural specification. As GCM components abstract away distribution, asynchronous remote communication are necessarily statically identified: they follow component bindings. References to active objects are stored in client interfaces, are thus futures are created upon access to those client interfaces. We thus instantiate one queue and one body per component, and one family of future proxies per method of each client interface.

Consequently, in a component model, building a behavioural semantics only relies on finitary abstractions of each variable, and on the behaviour of service methods for each method of each service interface of a primitive component that includes future access specification. Note that, as shown in Section 3.4.1, the behaviour of composite components can be automatically generated: they only delegate requests to sub-components or to external components. They also return directly futures as results of the requests they serve.

Concerning the finite abstract domain of variables, in general we rely on the direct input of the programmer for those domains, even if we could provide tools to help inferring those domains, e.g. inferring the domain of a variable from the domain of other ones.

JDC: a language for the specification of component systems

We investigated the possibility to design a new programming language from which it would be possible to generate GCM components with a guaranteed behaviour. It is called JDC for Java Distributed Components specification language. It consists of a structural specification language close to an ADL, and of a behavioural specification language for writing the business code. Then pNets specifying the component behaviour can be generated from the JDC code, and then verified. On the other side, application skeletons can be generated from the behavioural specification. Those skeletons being then filled in with details of the application logic that have not been verified. The idea is that some of the details of the application logics have to be abstracted away for the verification, and provided the properties of interest are not dependent on those details, it is possible to guarantee the correct behaviour of the components. In general, the specification should feature the same communication patterns, and the same components and remote invocations, but the part of the data and control flow dependencies that are not related to the properties of interest can be abstracted away, and filled later by the (Java) programmer. Some dynamic integrity checks can also be generated to check that the Java code written by the programmer does not break the proven properties. More details on JDC can be found in [34].

The JDC language has not been implemented yet, mainly due to the difficulty of implementing a new language and because the Vercors team focused on the design and implementation of a graphical interface for specifying both the component structure and the service method behaviours. This interface also provides integration with the model-checking tools.

In practice: the Vercors platform

In practice, for specifying the behaviour of service methods, while the construction of pNets by static analysis has been studied in the past [Bou04], in our recent examples we specified by hand, as (p)LTSs, the behaviour of the service methods. However, the solution we promote is the specification in Vercors of the behaviour based on UML 2 State Machine diagrams, from which we will generate the adequate pNet. This method is currently being implemented and promoted in the Vercors platform, it also comes with a variant of UML component diagrams for specifying GCM component architectures. The graphical tool allowing to specify GCM components and their behaviour based on UML is called VCE.

Figure 3.7 shows the architecture of the Vercors platform, it illustrates the previous paragraphs and provides an overall view of the Vercors environment. From a VCE specification, our objective is to generate automatically pNets, and from the specification of the instantiation domain for each variable of the pNets we generate a finite model that can be model-checked. From the specification, we can also generate the component architecture that can be completed and executed in the GCM implementation. Even though all those generation phases are not entirely implemented, in our last case study [47], half of the behavioural model have been automatically generated, and we expect to be able to generate automatically whole component specifications very soon.

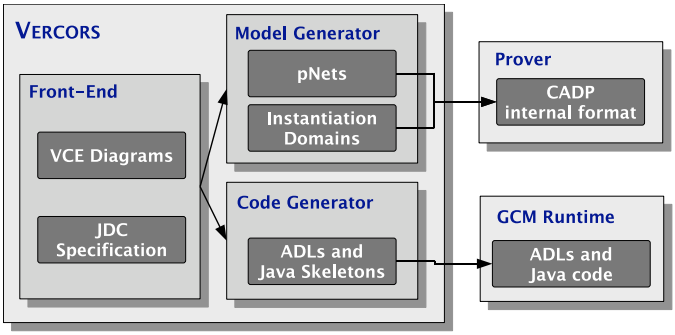


Figure 3.7: The VERCORS architecture

Behavioural specification of components: an illustrative example

Figure 3.8 shows a composite component as can be drawn in the Vercors platform. The component has two subcomponents, A and B, bound together and to the composite component. Type annotations are attached to each interface, they define the signature of each method of the interface.

Figure 3.9 shows the pNets structure corresponding to the composite component of Figure 3.8. It illustrates the structure of the pNets we generate for specifying the behaviour of a composite component. We use it here to illustrate how we are able to generate behavioural models for GCM components.

Two sub-pNets represent the behaviour of subcomponents A and B. A queue pNet receives **?Q_m0(f,arg)** requests where **f** is the future corresponding to the request and **arg** the value passed as argument. **Serve*** communications allow the body to retrieve those requests, which will then be treated by the **Deleg_m0** pNet, this pNet receives **Call** communications from the body and delegates the request to an inner component (here, A); during this process, a future proxy is created by the proxy manager **PMf(S)**, the proxy (**PF1_m0[q]**) is responsible for receiving the reply when **A** has finished the request treatment and for forwarding this result to the outside of the composite component: **R_m0(q,val)** that becomes **R_m0(f,val)**. Note that this proxy encodes some basic form of first class future: the future **q** corresponds to the same result as the future **f**. This situation also corresponds to the Figure 3.4 of Section 3.4.

Similarly, requests emitted by the inner components arrive in the queue[4], they are then delegated to the outside world by a similar mechanism: a **Deleg_m** pNet delegates the call, and creates a future proxy, which will be responsible for sending back the result to the appropriate inner component. Here again the proxy manages the fact that both the future **q** and the future **fa** (or **fb**) represent the same result.

Finally, note the proxy structure we adopt: there is one proxy manager **CPM*** for each method of each client interface (proxy managers are both indexed over interfaces and over methods). Then each of those managers itself manages a family of proxies **CProxy***. Performing model-checking on (the behaviour of) those structures then requires a precise definition and optimisation of the number and size of those families.

All the communications expressed above, but also the communication channels between the different inner components – requests **Q_m3** and the corresponding replies **R_m3** – can be automatically generated and correspond to synchronisation vectors of the pNet of the composite. Different boxes are expressed as pLTSs, except of course inner components that are pNets. Those pLTSs are not shown

[4] we drew two **Queue** boxes, but they represent the same element

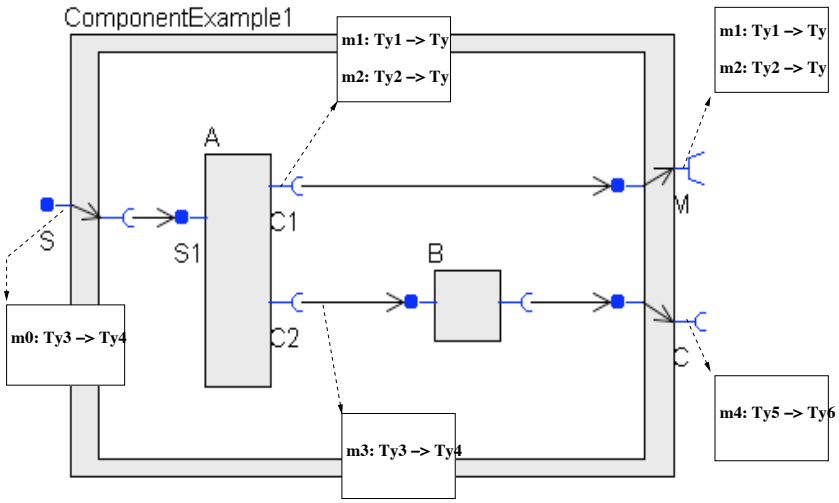


Figure 3.8: A simple composite component in Vercors

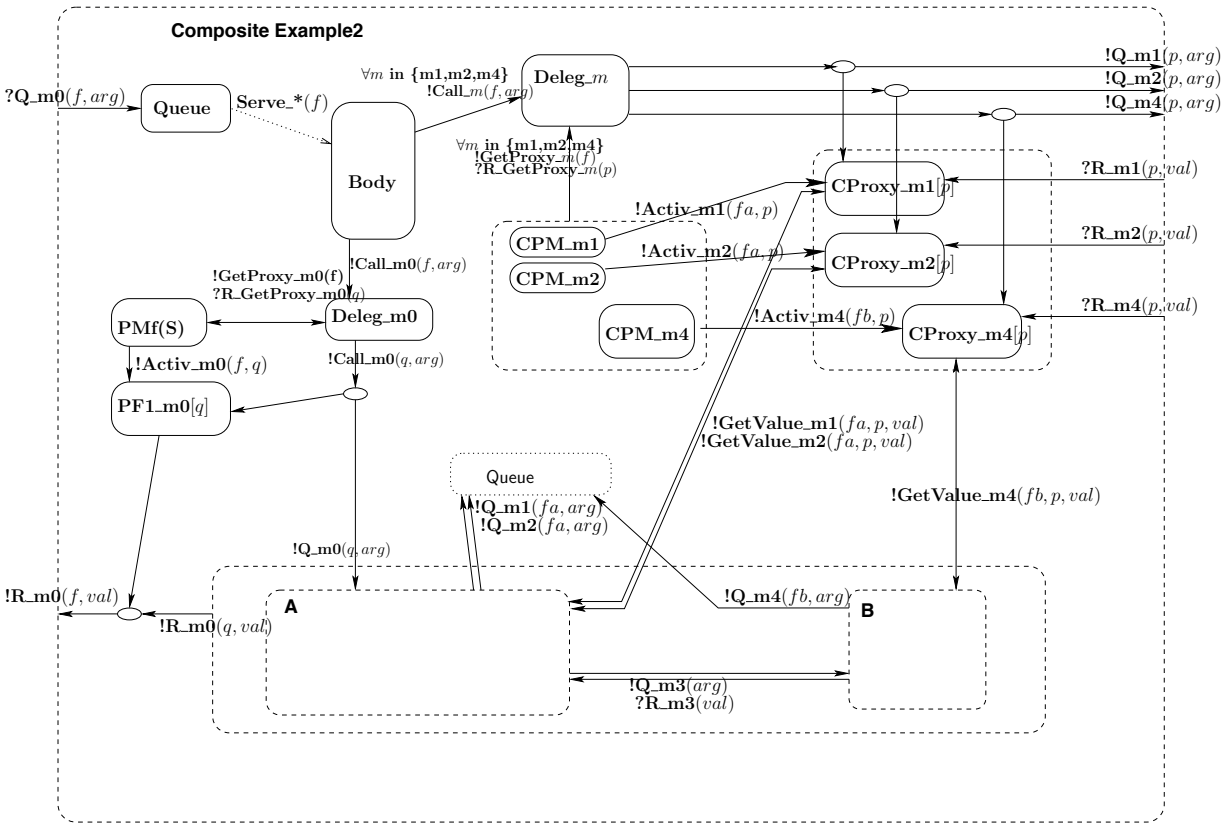


Figure 3.9: pNet for the composite component from Figure 3.8

here for conciseness of this section. Similarly, we are able to generate communication specific aspects, queues, bodies, and future proxies for GCM components.

Overall, from the specification of service methods and the description of the component architecture, we are able to generate a pNet specifying the behaviour of a component assembly. Then, from finite instantiation domains for future proxies, queue length, ... we are able to generate a finite behavioural model that can then be model-checked to verify the correct behaviour of a GCM application. more details on the size of the systems we are able to verify and the optimisation techniques we rely on can be found in [42]

3.6.4 Paper from FACS 2008

The paper presented in this section focuses on the representation of futures in our behavioural model. More precisely the contribution presented in this paper is twofold. First, we provide a static representation for futures, this consists mainly in an abstraction suitable for the static analysis of programs with futures. Second, we use our behavioural models to detect local deadlocks in a component system with first class futures. Additionally, this paper provides an extension to the interface definitions with annotations on the arguments that can contain a future; this enables the behavioural verification of components with first-class futures but also avoids deadlocks in some cases by enforcing additional synchronisations.

Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 260 (2010) 155–171
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Transparent First-class Futures and Distributed Components

Antonio Cansado, Ludovic Henrio, Eric Madelaine

INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis
2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex - France
Email :First.Last@sophia.inria.fr

Abstract

Futures are special kind of values that allow the synchronisation of different processes. Futures are in fact identifiers for promised results of function calls that are still awaited. When the result is necessary for the computation, the process is blocked until the result is returned. We are interested in this paper in *transparent first-class* futures, and their use within distributed components. We say that futures are *transparent* if the result is automatically and implicitly awaited upon the first access to the value; and that futures are *first-class* if they can be transmitted between components as usual objects. Thus, because of the difficulty to identify future objects, analysing the behaviour of components using first-class transparent futures is challenging. This paper contributes with first a static representation for futures, second a means to detect local deadlocks in a component system with first class futures, and finally extensions to interface definitions in order to avoid such deadlocks.

Keywords: Hierarchical components, distributed asynchronous components, formal verification, behavioural specification, model-checking, specification language.

1 Introduction

This paper provides a model for reasoning about futures in the context of distributed components. We define here a framework allowing us to find which components can be blocked on an access to a future, and extend the specification of component interfaces in order to avoid some of these blocked states.

Our approach consists of specifying statically a behavioural model for distributed systems with futures, in order to apply model-checking techniques on it. This approach can relate to [14,10], but in this paper we focus on the modeling of futures that were not taken into account in previous work in this domain. In our previous works [3,2] we gave behavioural models for active objects and asynchronous distributed components such as the GCM (Grid Component Model [4]), however, futures were local, i.e. they were not sent between activities (we call *activity* a unit of distribution). The behavioural models, as in this paper, are based on an inter-

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2009.12.036

mediate language that we call *Parameterized Networks of Synchronised Automata (pNets)* [2].

The remainder of this section reviews related works on futures, defines the context of our work, and our contribution. In Section 2 we provide an abstraction suitable for static analysis of programs with futures, as well as behavioural models for futures. In Section 3 we apply the behavioural model to an example and show some properties that can be checked. Section 4 builds on definitions useful for detecting blocked components. Finally, in Section 5 we suggest an extension for the definition of component interfaces that can prevent some deadlocks.

1.1 Futures

Futures [13,18,16] provide synchronisation for concurrent or parallel languages. A future is an "object" that can be filled or not. Accessing a future blocks if the future is not filled, and returns the encapsulated object otherwise. Explicit futures are typed: a (static) type "Future" exists and futures are statically identified as the objects with this type, they can only be accessed by a `getValue` method. In this paper we are rather interested in *transparent* futures. Transparent futures are accessed like the object they encapsulate, so futures and non-future objects are manipulated in the same way. Futures have some kind of proxy that automatically blocks the program when accessing the object if the future is not filled. This renders a data-driven synchronisation in a mechanism called *wait-by-necessity*.

The first question is: "when are futures created?". The answer depends on the programming language. Generally, futures are created by an explicit construct which delegates a thread for computing the future value, this construct is named `future` in Multilisp [13,11], and `thread` in [16]. In AmbientTalk [9], ProActive [5], and ASP [6], futures are created automatically upon a remote method invocation. In ProActive and ASP, there is no syntactic difference between a local method call (which may or may not create a future), and a remote method call (which creates a future). This means futures are implicit and their creation depend on the data-flow; synchronisation is also implicit and occurs on strict access to a future.

Another question is: "Which are the blocking operations on a future?". Whereas in a local setting most of the languages agree on the definition of a strict access to an object, in a distributed setting things get more complex. The question above boils down to: "Is transmitting a future to a remote object a strict operation?". In ASP and ProActive, we consider that futures can be transmitted between remote activities because we proved that this property had no influence on the possible execution paths, except that it can avoid some dead-locks [6]. We call such futures *first class futures* because they can be manipulated as first class objects.

As an interesting related work, [8] provides a language with futures that features "uniform multi-active objects": roughly, each method invocation is asynchronous because each object is active; each object has several current threads, but only one is active at each moment. However, their futures are explicit: a `get` operation retrieves their value. The authors also provide an invariant specification framework for proving properties on such multi-active objects with futures.

Our point of view in this paper is to build a behavioural model for futures *à la* ASP calculus, but more generally the proposed model applies to any kind of transparent first-class futures featuring wait-by-necessity mechanism.

1.2 Components and Futures

Components are software entities with interfaces (or ports); those interfaces are connected together by bindings. In GCM, there are two kinds of components: *primitive components* that implement some behaviour in Java, and *composite components* that compose other components. The composites are in fact dispatchers of services: the composite dispatches the received requests to the bound interfaces.

If communications occurring over the bindings are synchronous, then the interfaces can be accessed as usual objects, having methods with parameters and a return type. When components are connected asynchronously, one must find a way to create a channel for the objects returned by the components. Futures can be used as identifier of the asynchronous invocations over components. Indeed, futures provide some kind of transparent channels that correspond to the original bindings, but taken in the opposite direction: from the server to the client.

But components and futures get more related when considering static analysis. Indeed, in an asynchronous component model like the GCM, only invocations on a component create a future. Thus, the components allow the static identification of future creation points, and thus a finer static analysis.

1.3 Components as an Abstraction for Distribution

Components relieve us from a difficult analysis task: in a distributed object-oriented language with implicit futures, it is difficult to identify the communication and the creation points of futures. Indeed, asynchronous method calls are syntactically similar to synchronous ones, and distinguishing one from the other can only be the result of a static analysis step which is by nature imprecise, consequently identifying the points where futures are created is also difficult. In a distributed component model like the GCM, however, the only method invocations that are asynchronous are the ones performed on interfaces. The topology of distribution and communication is directly given by the component structure.

Unfortunately, although the component model provides a good abstraction for distribution and specifies which calls are asynchronous, the flow of futures is still hard to approximate. In other words, the component abstraction tells us where futures are created but not where they can go. The dynamic and transparent nature of futures implies that each result and each parameter of an invocation may contain a future; thus the only safe assumption for parameters and results is that any object received can be a future, and every field of this object can itself be a future. This leads to a very imprecise approximation of the synchronisation in the system; this over-approximation can always be improved by static analysis (when the system is closed), or by specification, as illustrated in Section 5.

1.4 Contribution

Transparent first-class futures provide a natural and efficient data-flow synchronisation where a result is awaited only when it is necessary. However, providing a model of programs using transparent first order futures is challenging. The contribution of this paper is first to give a static representation of transparent first-class futures, second to characterise how access to futures can block components indefinitely, third to use the previous results to identify local deadlocks, and finally extend the definition of interfaces to avoid some blocked states.

2 A Static Representation for Futures

The objective of this section is to give a behavioural model for transparent first-class futures, this model is intended at the static verification of the behaviour of components. We assume that the accesses to the component interfaces and the creation point of futures are given in the functional behaviour of the component (Body). We start this section by a brief definition of the pNets model, and of its (static) graphical representation on which we build our models.

2.1 Informal Description of pNets

The **pNets** model, formally defined in [2], is a generalisation of **Nets** [1]. It synchronises a (potentially infinite) number of processes by means of N-ary synchronisation vectors. The parameters set a symbolic representation of the system.

A pNets takes the form of a hierarchy of processes; each process encodes a particular Sort of the pNet. The Sort of a pNet can be filled with a parameterized LTS (pLTS) satisfying a Sort inclusion condition, or with another pNet.

In this paper we use a static subset of pNets in which synchronisation vectors don't change. A pNet is depicted by a set of boxes, and their synchronisations are given by arrows that express the synchronisation vectors; the direction of the arrow is an abstraction of the data-flow. For N-ary synchronisation, we use a synchronisation vector with an eclipse in the middle.

The pLTSs are represented by states (circles), and transitions (arrows). The transitions encode the actions that a process can perform in a given state.

A label with !*act* and ?*act* denote actions of emition and reception of *act* resp. An action *act* is a visible action without synchronisation; however, this action may be the result of synchronising other actions within inner pNets.

Moreover, within the behavioural model we adopt the following notation:

- `call(`f_{id}`,M(args))` is a method call `M`, with parameters `args`, which result should update the future f_{id}.
- `response(`f_{id}`,val)` is the result of a method call; it updates the future f_{id} with the value `val`.
- `forward(`f_{id}`,val)` is the message forwarding the value `val` of the future f_{id}.
- `getValue(`f_{id}`,val)` is the access to the future f_{id}; the body accesses the future

f_{id} and receives from the proxy the value `val`.

- `serve(M)` is the request to serve a method `M` from the queue.
- `pNets(C)` is the behavioural model of component `C`.
- `Proxy(`f_{id}`)` is the proxy dealing with the future f_{id}.
- `Body` is the (user) functional behaviour of the component.
- `Queue` is the request queue of the component.

2.2 Representing Futures

In the examples below, we will use a Java-like language *à la* ProActive, where creation of futures are method calls on some required interfaces (named `itf` here), i.e. all future creations are of the form `f=itf.foo()`, resulting in a future stored in the variable `f`. We call *future update* the operation consisting in replacing a reference to a future by the value that has been calculated for it.

The representation of a future must allow the contained object to be accessed, i.e. to synchronise futures. We call `waitFor` the primitive allowing the update of a future to be awaited (this primitive has also been named `touch` or `get` [11]). When futures are transparent, this waiting operation is automatically performed upon an access to the content of the future. We describe in this section what behavioural model can be created for this kind of futures. For the moment, we consider that futures cannot be passed between remote entities, and thus the future is necessarily accessed by the same entity that created it (at another point of the execution).

The objective of this part is already to be able to provide a model for the following piece of code:

```
f=itf.foo();          // creation of a future
if (bool) f.bar1();   // wait-by-necessity if bool is true
f.bar2();             // wait-by-necessity only if bool is false
```

In here, if `f.bar1()` is executed, then `f` must be filled; in this case `f.bar2()` will be necessarily non-blocking. Otherwise, `f.bar2()` may or may not be blocking depending if the future `f` is already filled by the time the call is performed. Note that it is much simpler when futures are explicit, i.e. if futures are typed.

In this work we formalise the abstract domain of a future. The previous example shows that futures are filled transparently at any time. Thus, whenever it is not statically decidable whether an object is a future or a value, it must be assumed as a future. This is an over-approximation that will, at least, include all possible synchronisations a variable may trigger. Therefore, static analysis of a program with futures requires the set of abstract values to be multiplied by two.

Indeed, statically each variable is either known to contain a value which *is not a future*, or, equivalently, a filled future, ranging in the domain of the usual static domain for values; or the variable *may be a future*, and when the future will be filled its value will range in the domain of the usual static domain for values. Note that an object that is not a future is semantically equivalent to a filled future. In abstract

interpretation [7] it would be easy to construct a lattice for this new abstract domain: suppose without futures, the abstract domain is a lattice $(D, \prec)$, then the new abstract domain taking futures into account is the *lattice* $D' = D \cup \{fut(a) | a \in D\}$ equipped with the order $\prec'$ built such that if $a \prec b$, then $a \prec' b$, $a \prec' fut(b)$, and $fut(a) \prec' fut(b)$. Indeed the abstract value a gives more information than $fut(a)$. To summarise, statically, the value for our objects are either "filled" or "potentially non-filled"; these abstract values are composed with the usual abstract values required for the analysis.

In the ProActive middleware, the example above creates a *proxy* in the first line, and all calls to the future stored in `f` would go through the proxy object, leading, if necessary, to a wait-by-necessity. For our model, the idea is the same: the variables have the "classic" static abstract domain, and the augmented lattice is taken into account by an additional automaton with the behaviour of a proxy. Initially, the proxy is in an empty state where the object can only be filled with a value, so any access to the variable will be blocking. In general, two instances of the same method call have two different futures, so the proxies are parameterized by the instance of method call. In Fig. 1 we show a first model on how two components communicate. The action `call(`f`,M(args))` puts the request in the server's queue, and initialises the local proxy. The call contains the *identifier* of the future to be updated, f. Once computed, the value of f is updated (`response(`f`,val)`).

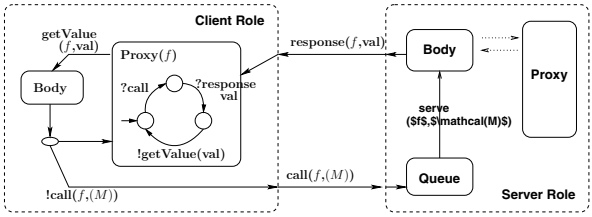


Fig. 1. Communication between two Active Objects

2.3 A Model for Transparent First Class Futures

We now extend the previous model to allow futures to be transmitted in a non-blocking manner. Futures can be transmitted in the parameters of a method call, or in the return value of a method call. Because of that, a future in an activity may have been created locally or by a third-party. In both cases, the activity is aware of the future *identifier*. The references of futures known by a component are local. Only when synchronising the components the references must match the data-flow. In pNets, synchronisation vectors allow us to synchronise different labels which we use to link the future references. This technique allows us to create the behavioural model for each component independently.

On the practical side, different future update strategies can be designed for propagating the values that should replace future objects. Despite having differences in performance, the update policies have equivalent behaviour, proved using ASP in [6]. This leaves freedom to choose any update policy. In this paper we use this result applied in the behavioural model.

Let $\sigma_{\mathtt{C}}$ be a valid execution on component `C`, `pNets(C)` the behavioural model of `C`, and f_{id} a future, then the model is built such that:

Property 1 *if* `getValue(`f_{id}`,val)` *in* σ_C*, then* `Proxy(`f_{id}`)` *is in* `pNets(C)`

As a consequence, the model has a proxy dealing with every future a component may receive. Due to imprecision of the abstraction, the component may even have proxies for futures that would never exist at run-time. However, any synchronisation is considered within the model.

Property 2 *if the value of* f_{id} *is computed, then all proxies of* f_{id} *are updated eventually*

Property 2 is true even for proxies that don't exist at run-time. The property is guaranteed by construction: (i) *the proxy that creates* f_{id} initially synchronises with the remote method call. The proxy then waits for the result (value of f_{id}). When the value of f_{id} is updated, the proxy forwards the value of f_{id} to all components to which the local component sent the reference f_{id}. (ii) *all other proxies of* f_{id} are initially in a state in which they are ready to receive the value of f_{id}; this guarantees they will also be able to be updated. When the proxy is updated, it forwards the value of f_{id} to all components to which the local component sent the reference f_{id}. In fact, a proxy forwards the value of a future to all components it has sent the reference to synchronously. Therefore, each proxy only needs one port "`forward`" for each future, independently of the number of components to which it sent the reference.

Property 3 *the update of proxies that do not exist at run-time has no influence in the behavioural model*

Depending on the data-flow, some components will receive the value of f_{id}, though the reference f_{id} was not transmitted. In this case, the reference f_{id} is also unknown to the given component, and thus the content of the future is innaccessible, i.e. the business part of the component is not affected.

Sending a future as a method call parameter

In Fig. 2, the `Client` performs a method call M_1 on `Server-A`, and creates a `Proxy(`f`)` for dealing with the result. Then the `Client` sends the future to a third activity (`Server-B`) in the parameter of the method $M_2(f)$. From `Server-B`'s point of view, there is no way of knowing if a parameter is (or contains) a future, so every parameter in a method call must be considered as a potential future. `Server-B` includes, therefore, a proxy for dealing with the parameter f of the method call M_2. For the sake of comprehension, however, in the figure the identifiers for futures already match the data-flow.

Retransmit a future received as a method call parameter

The previous example is extended such that `Server-B` transmits the future f to `Server-C`. This is partially depicted in Fig. 3. The proxy in `Server-B`, after receiving the value of the future (`?forward(val)`), forwards the value to the components

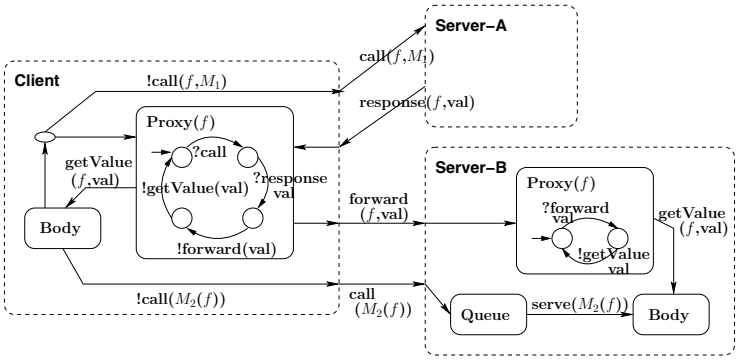


Fig. 2. Transmitting a future as method call parameter

it has sent the future reference.

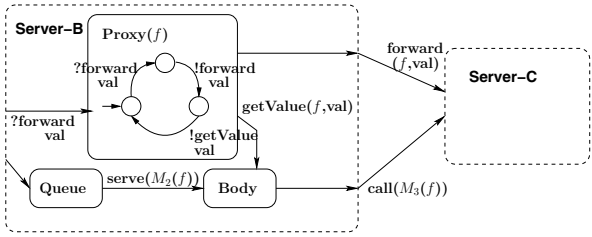


Fig. 3. Retransmitting a future as method call parameter

Returning a future

In Fig. 4 an activity (`Server-B`) creates a future f_2 and then transmits f_2 to the `Client` within the result of the method call $M_1(args)$. The behavioural model is slightly different to the one in ASP: instead of returning a future, there is a proxy on the server put in charge of forwarding the concrete value once it is known; no value or future is sent to the `Client` in the meanwhile. Using this mechanism, the behavioural model of the `Client` is the same no matter whether `Server-B` returns a value or a future. Moreover, `Client` remains as usual; the result of the method call $M_1(args)$ has a proxy `Proxy(`f_1`)` dealing with the result. It is up to the proxies of the `Client` and the `Server-B` to synchronise in order to match the expected behaviour. Concretely, the action with the response to the `Client` (`response(`f_1`,val)`) is synchronised with the forward action (`forward(`f_2`,val)`) of the `Proxy(`f_2`)`; it will then update the `Proxy(`f_1`)`. If the `Client` accesses the future, then it synchronises with its local proxy, `Proxy(`f_1`)`.

2.4 Summary: How to Build a Future Proxy?

We showed in this section that it is possible to specify the behaviour of proxies for futures providing a good approximation of the future flow is given. To summarise:

- Each proxy finishes by providing a `!getValue` transition allowing the access to the future value.

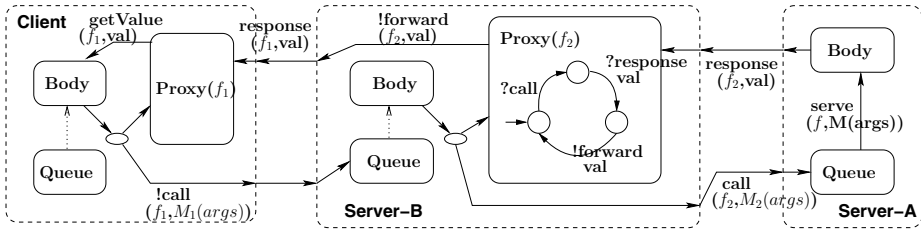


Fig. 4. Transmitting a future as a result for a method call

- At the future creation point (i.e. on the caller side of a remote method invocation), a proxy starts by two transitions `?call` for synchronising with the remote call, and `?response` for synchronising with the response.
- In the other activities that can *receive* the future, the proxy starts a single transition `?forward` for receiving the forwarded future value
- If the activity may send the future to another one, then the `!getValue` transition is preceded by a `!forward` one.
- Proxies that are used for transmitting a future reference as the value of another future are slightly different: they need no `!getValue` because they simply forward the value they receive as the value for *another* future. Assigning a future reference to another future is directly ensured at a higher-level, that is to say by the composition itself. This ensures that the behavioural model is still compositional as no name of an externally created future exist in the proxy.

3 Illustrative Example

Consider a component system like in Fig. 5. It contains a component `A` that requests some services of `B`, and stores the return value in a variable f. Component `A` does not access the return value f immediately; instead, it forwards f to the component `E`, and possibly forwards f to the component `F`. Finally, `A` accesses f. Component `B` is a composite component that wraps a primitive component `C`. `C`, when serving the method `foo()`, requests a service to `D` by means of its wrapper, `B`, and returns.

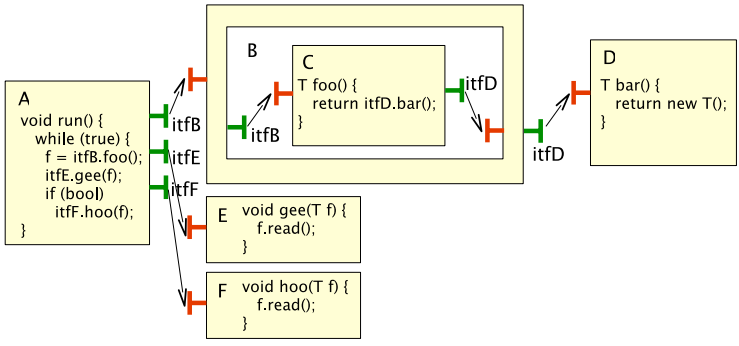


Fig. 5. Going through a composite thanks to first-class futures in ProActive

In GCM/ProActive, this would instantiate 6 active objects; one per primitive component (`A`, `C`, `D`, `E`, `F`), plus one per composite component (`B`). The active object

for B mediates services: requests coming from the composite's server interfaces are dispatched to a subcomponent, requests coming from its subcomponents client interfaces are dispatched towards an external component. For that it makes extensive use of first-class futures; it serves a request, performs a remote method call, creates a future for holding the result, and then sends back the future to the caller. In other words, B *delegates* the requests it receives to components C and D, returning the future corresponding to the delegated method call.

3.1 Behavioural Model

Fig. 6 shows the model created for the system above. Components E and F have similar behaviours. Components B and C are synthesised by the pNets model BC depicted in Fig. 7 (which is as well a model for a composite component). In this example, we index each future by the name of the component that created it.

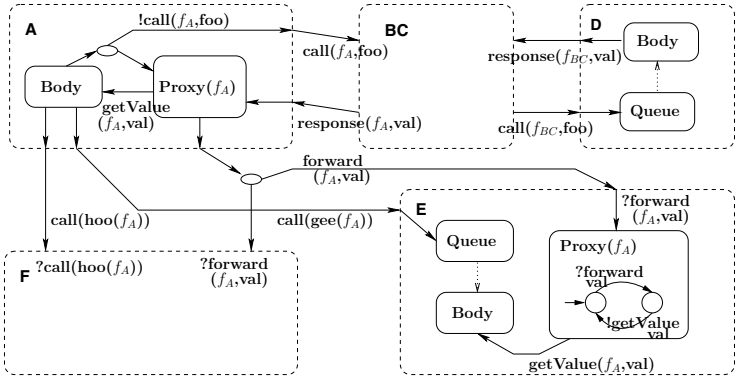


Fig. 6. pNets model of Fig. 5

In the pNets model of A, futures are forwarded to several activities; a future is sent as parameter of the method calls to E and F in call(gee(f_A)) and call(hoo(f_A)) resp. A proxy is created in each callee with the identifier (f_A) matching the proxy of the caller, i.e. Proxy(f_A). Proxy(f_A) in pNets(A), after receiving the concrete value, will forward the value to both activities E and F. This is seen as an action forward(f_A,val). As a remark, the update of Proxy(f_A) in F is done no matter whether the component is called or not, however if the call is never performed the proxy is unreachable (its identifier is unknown).

Fig. 7 shows the behaviour of components B and C. Component B creates the proxies Proxy(f_{B^1}) and Proxy(f_{B^2}) for the calls foo and bar resp. B does not access the proxies, so the responses of the calls are forwarded directly by the proxies. The same models applies for component C. It creates a proxy Proxy(f_C) when calling bar. C returns the future f_C, so Proxy(f_C) is the one forwarding the value it receives as a response to B.

3.2 Properties

In terms of behaviour, the value of f has no impact on the control flow, thus it is abstracted to a single abstract representative *dot*. It is the proxy that takes care

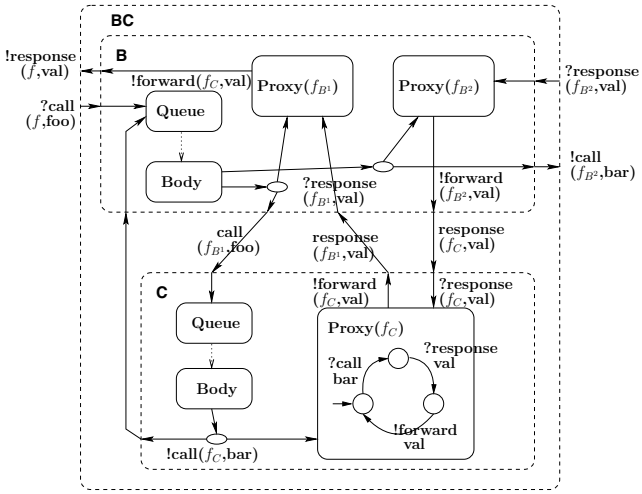


Fig. 7. pNets model of components B and C

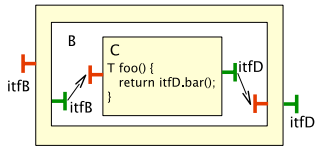


Fig. 8. Components B and C

of the abstract values *filled* and *non-filled*, meaning that we only care if the future has been filled or not and when it is accessed. We used the CADP [12] toolbox for generating the state-space and for the verification; the complete LTS for the system has: 12 labels, 575 states and 1451 transitions; when minimised using branching bisimulation 52 states and 83 transitions remain. Some properties can be found using alternation-free μ-calculus formulas [15]:

System is deadlock-free. As the program never terminates, we proved in CADP that, on the global-state space, every state has at least one successor.

All futures are necessarily updated. This is proved by stating that the call on `itfB.foo()` in component `A` will update all futures within a finite number of actions. In pNets, this is (see Fig. 9): starting in a state where `call(`f_A`,foo)` is performed, all leading traces will perform the future updates along the transmitting chain. More precisely, as no future is returned until a real value is known, when `D` computes the value, the components of the chain (`D`, `B`, and `C`) reply. Those `response` messages follow all the chain leading to `A`. Finally, `A` forwards the value to `E` and `F` (`forward(`f_A`,val)`).

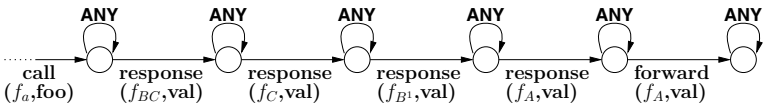


Fig. 9. Automaton representing the traces where futures are updated

System deadlocks if the composite does not support first-class futures. Suppose that the programming language does not support the transmission of futures, which implies that a method call must return a value (if any). Fig. 10 shows a modified version of the composite `B` with this behaviour. When the component `B` receives a request `?call(`f`,foo)`, the `Body` of `B` should: call the component `C` (action `!call(`f_{B^1}`,foo)`), access the return value (action `getValue(`f_{B^1}`,val)`), and then return the value of f_{B^1} (action `!response(`f`,val)`). The value of f_{B^1} is

computed by component `C` on a service that must go through component `B`. Therefore, this value will never get computed as component `B` is blocked synchronising on `getValue(`f_{B^1}`,val)`. Such a scenario systematically results in deadlocks.

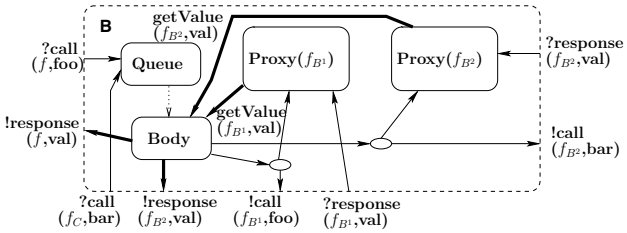


Fig. 10. pNets model of a composite without first-class futures

System deadlocks if `itfB.foo()` is synchronous The deadlock is similar to the previous one; if `foo()` is synchronous, then this call blocks component `B` until the result is known. What it means is that a synchronous call cannot trigger a flow that goes through a composite twice. This is a common pitfall for inexperienced programmers with GCM/ProActive that we can fortunately detect in our models.

4 Identifying Blocked Components

This section shows how to detect whether there are components blocked infinitely on a future access. We investigate definitions and properties adequate for this purpose based on ASP.

It is easier to start with the example of Fig. 11. A *Client* queries for some data. This data is properly formatted by the *QueryManager* and then forwarded to the *Database*. Once the *Client* creates the future d, it inserts a new entry into the table t with data from d; this is a method call performed directly towards the *Database*. The system may deadlock, though, due to a race condition on access to the *Database*. If the *Client* accesses the *Database* before the *QueryManager* does, the *Database* will access the future d – thus block –, but d will never be updated because the *Database* itself must update this future. The behavioural model of the previous section is enough to detect this problem.

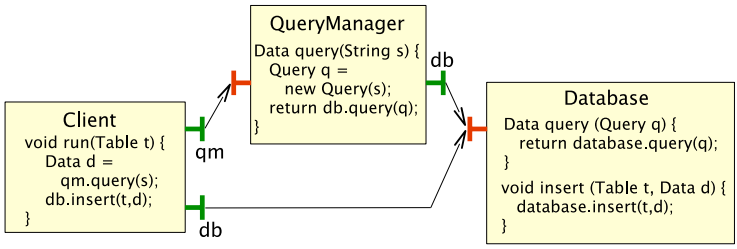
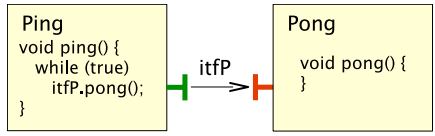


Fig. 11. Race condition in GCM / ProActive



Now, suppose the database example runs in parallel with two components that run continuously as the figure on the right. Using the same analysis over the complete system, no deadlock is found: indeed, some part of the system is constantly doing some work, i.e., in the global state-space every state has at least one transition. What we need is a finer grain definition of blocked component.

In ASP, synchronisations happen upon access to a future and when serving a request from the request queue. In the following we consider components that serve requests in a FIFO order, and thus no synchronisation on a request is made. Therefore, all deadlocks in a system must be related to access to a future. More precisely, there must be at least a future that is accessed and that is never updated. This gives us a starting point for defining what is a (non)-blocking future.

We refine the behavioural model in order to observe the accesses to futures in detail: first, there is a visible, non-synchronised action `waitFor(`f`)` signalling that a component wants to synchronise on the content of a future; and then a synchronised action `getValue(`f`,val)` where the component effectively retrieves the content of the future.

Unfortunately, due to an unfair scheduler, a subsystem (e.g. the Ping-Pong) could interact indefinitely while some components never progress. In this case, once the action `waitFor(`f`)` is performed, the action `getValue(`f`,val)` is reachable but not inevitable. Therefore, we impose some kind of fairness in traces. We use the definition of *fair reachability of predicates* as given by Queille and Sifakis [17].

Definition 4.1 A sequence is fair iff it does not infinitely often enable the reachability of a certain state without infinitely often reaching it.

Finally, we are able to define a *non-blocking future* and a *non-blocking component*.

Definition 4.2 A future f is *non-blocking* iff, under the fairness hypothesis, if each time the action `waitFor(`f`)` is performed, then the action `getValue(`f`,val)` is eventually reached.

Definition 4.3 A distributed component is *non-blocking* iff every future it accesses is *non-blocking*.

Moreover, if a distributed component system is *non-blocking*, and synchronisations are only due to future access, then the system is deadlock-free. In other words, if the system deadlocks and synchronisations are only due to access to futures, then there is at least one component blocked waiting for a future.

The main advantage of our approach is that it can be encoded in a model-checker, and thus we can ensure that every needed future reference is updated; in other words the program will have the expected behaviour: *all the object accesses of the program will occur*.

5 Extending the Interface Definition

By switching from an object-oriented to a component-oriented design, we make the application topology and dependencies explicit because: (i) every component contains a single thread; (ii) all method invocations are restricted to calls on client interfaces; and (iii) all future creation points are restricted to results of these method calls on client interfaces.

This removes some of the imprecision of the static analysis. Nevertheless, a source of ambiguity remains in open environments: a parameter (or any subfield) received in a method call may be either a future or a value due to transparency of first-class futures. This section suggests an extension to the Interface Description Language (IDL) to improve the precision of analysis and specification, we also explain how this extension prevents the occurrence of some deadlocks.

5.1 Principles

In order to be safe, the behavioural model must be an over-approximation of the implementation, including a proxy not only for futures, but also for variables or parameters which *may* be a future. Such imprecision is due to the undecidable nature of static analysis, and to the transparent nature of futures.

Moreover, for the database example of Fig. 11, one would like to offer means to correct the deadlock. For this, one can enforce further synchronisation on the *Client* side in order to guarantee that the *Database* always receives a value instead of a future. Up to now the only way to ensure such a synchronisation is to insert a call to the `waitFor()` primitive within the code of the *Client*. Nevertheless, from the server side, i.e., the *Database*, one does not know this information. Thus, the behavioural model for the *Database* still expects a future; the unneeded traces will only be pruned when synchronised with the environment.

The IDL used in the GCM specifies the interface signatures, but is insufficient to deal with transparent first-class futures. Based on the interface signature, one does not know whether method parameters are futures or not. Moreover, there is no way of controlling which parameters cannot be futures. Typing futures would solve the issue, however, we would lose all the good properties shown in Section 2.2. One way is to specify within the IDL which parameters (or fields) cannot be futures (i.e. marking them as *strict value*); the other parameters are allowed to be futures or not. Note that this is less restrictive than typing because some parameters can still be either a value or a future.

In an open system this information cannot be inferred by static analysis. It is a contract on futures that affects both client and server: client interfaces must ensure that method parameters match the interface specification; server interfaces assume – and may test – that method parameters agree with the interface specification. The contract also decreases the non-determinism in the server behaviour.

It is true that by the use of strict parameters there is less concurrency; components may enforce further synchronisations before performing remote invocations. On the other hand, behavioural models are more precise and closer to real executions; the programmer can specify parameters that are known to be non-futures.

5.2 Interface Specification

The difficulty is finding, statically, a proper abstraction for the parameter structure. In theory, every subfield of every parameter may be a future. Therefore a static representation of arbitrary types is impractical. Here we suggest a relatively precise approximation; marking a field as *strict value*, recursively, means that all its subfields (known at runtime during serialisation) are *strict values* as well. Similarly, not marking a field implicitly means that, recursively, all its subfields (except the marked ones) *may be futures*.

In the example of Fig. 11, an easy solution to the deadlock mentioned before is to force value-passing of d. Based on Java 1.5 Annotations the specification of the interface `DB` would look like:

```
interface DB {
  Data query(Query q);
  void insert(Table t,
      @StrictValue Data d);
}
```

On the practical side, if d is still a *non-filled* future by the time the method `insert(`t`,` d`)` is invoked, the invocation is halted until the future is updated. This way, the system is guaranteed to be *non-blocking*.

To implement this in ProActive, we would have to modify the *Meta-Object-Protocol* (MOP). The MOP will:

(i) *on the client side*: during serialisation any parameter marked as *strict value* will enforce an explicit synchronisation on the related object; the overhead is payed only for methods with annotated futures.

(ii) *on the server side*: during deserialisation any parameter marked as *strict value* can be checked not to be a future; to avoid overhead, one may assume that the sender respects the contract because it was previously checked during serialisation. Moreover, the affected parameters will never block because they are guaranteed to be concrete values.

6 Conclusion

Throughout this paper we studied how to model transparent first-class futures in distributed components, as well as some necessary properties in order to avoid deadlocks related to futures. To our knowledge, the only previous work providing static reasoning on futures is [8], and focused on invariant proofs for explicit futures. We provides here behavioural static models for transparent futures that can be detailed as follows:

A Model for Transparent First-Class Futures. We defined an abstraction and a model for futures and their behaviour (synchronisation, update). This model expresses the flow of future references and future values. It extends our previous works by giving behavioural models for transparent first-class futures, relying heavily on the properties proved in the ASP-calculus.

A Framework for Detecting Blocked Components. Thanks to our model we are able to detect components indefinitely blocked on future access using model-checking

techniques. This way, futures for which a value will never be computed can be identified. Our model greatly helps the programmer to find synchronisation issues in concurrent programs with futures.

Rich-Interfaces. Finally, we showed that the Interface Description Language of GCM can be improved in order to specify synchronisation on futures at the interface level. This lifts some synchronisation from the behaviour up to the interface level, which yields more precise behavioural models and avoids some deadlocks.

An alternative model for futures would consider global references to further optimise the state-space. The properties on confluence inherited from ASP allows us to update all references of a future synchronously without other impact than generating traces with less interleaving. This effectively avoids the propagation of values found in our model, however it requires inter-procedural static analysis, so it does not allow the model to be built independently.

References

[1] A. Arnold. *Finite transition systems. Semantics of communicating sytems.* Prentice-Hall, 1994.

[2] T. Barros, R. Boulifa, A. Cansado, L. Henrio, and E. Madelaine. Behavioural models for distributed Fractal components. *Annals of Telecommunications*, accepted for publication, 2008. also Research Report INRIA RR-6491.

[3] T. Barros, L. Henrio, and E. Madelaine. Verification of distributed hierarchical components. In *International Workshop on Formal Aspects of Component Software (FACS'05)*, Macao, Oct. 2005. ENTCS.

[4] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. Gcm: A grid extension to fractal for autonomous distributed components. *Annals of Telecommunications*, accepted for publication, 2008.

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

[6] D. Caromel and L. Henrio. *A Theory of Distributed Object.* Springer-Verlag, 2005.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Record of the Fourth Annual ACM SIGACT-SIGPLAN Symp. on Principles of Progr. Languages*, pages 238–252, Los Angeles, CA, 1977. ACM Press, New York.

[8] F. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *ESOP*, pages 316–330, 2007.

[9] J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt, and W. D. Meuter. Ambient-oriented programming in ambienttalk. In D. Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 230–254. Springer, 2006.

[10] F. Fernandes and J. Royer. The STSLIB project: Towards a formal component model based on STS. In *Proceedings of the Fourth International Workshop on Formal Aspects of Component Software (FACS'07)*, Sophia Antipolis, France, September 2007. To appear in ENTCS.

[11] C. Flanagan and M. Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.

[12] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology Newsletter*, 4:13–24, Aug. 2002.

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

[14] ISO: Information Processing Systems - Open Systems Interconection. LOTOS - a formal description technique based on the temporal ordering of observational behaviour. ISO 8807, Aug. 1989.

[15] R. Mateescu. Efficient diagnostic generation for boolean equation systems. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 251–265, 2000.

[16] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, Nov. 2006.

[17] J. Queille and J. Sifakis. Fairness and related properties in transition systems – a temporal logic to deal with fairness. *Acta Informatica*, 19(3):195–220, July 1983.

[18] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and programming in an object-oriented concurrent language ABCL/1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 55–89. MIT Press, Cambridge, Massachusets, 1987.

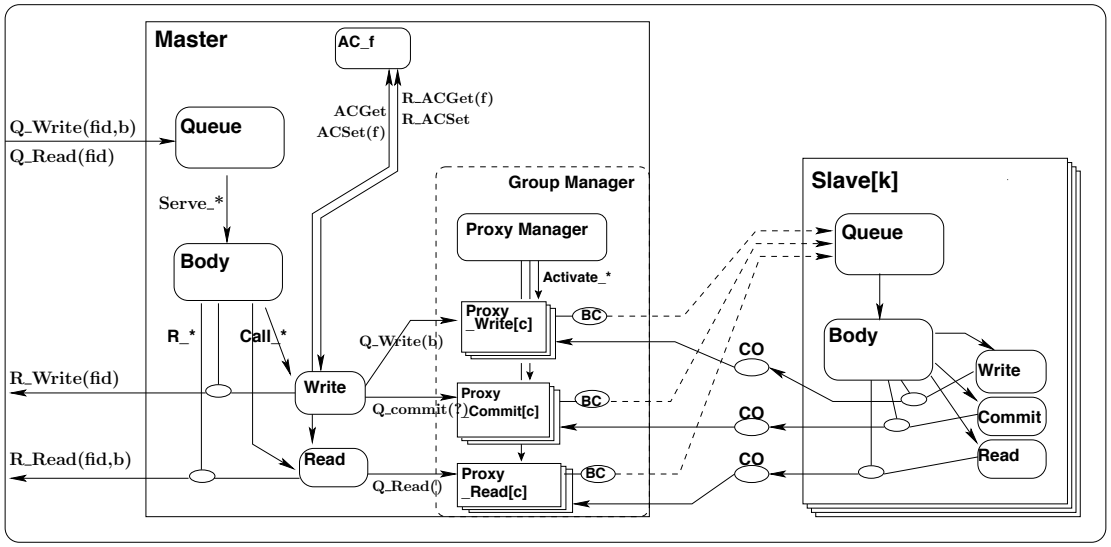


Figure 3.10: pNet Architecture for the fault-tolerant application

3.6.5 A model for one-to-many communications

The example presented in Section 3.6.3 was following a semantics *à la* GCM but had no example of architectural features specific to GCM. However, we recently focused on the handling of group communications [42] and multicast interfaces [47] in pNets.

In particular, in [47] we focused on the verification of a fault-tolerant application based on replication and consensus. To illustrate our approach, we choose a simple distributed application featuring fault-tolerance by replication. Though the fault-tolerance properties we address are not outstanding, we think this application is a good opportunity to investigate on the use of model-checking to ensure safety of fault-tolerant applications. This article provides a model for one-to-many communication, but also studies the modelling of faulty processes, and investigates the use of model-checking for verifying fault-tolerance from an application point of view. Our purpose is not to prove that a fault-tolerance protocol is correct but to understand whether it is possible to represent all the aspects of a complete component application communicating by request-replies, and at the same time reason about the fault-tolerance of this entire application. Our application consists of a Master component replicating data to be stored on several workers. The master updates the worker value, and gathers replies from workers to retrieve the stored value. If enough non-faulty workers are instantiated, and enough identical replies are returned to the master, the stored value can be retrieved.

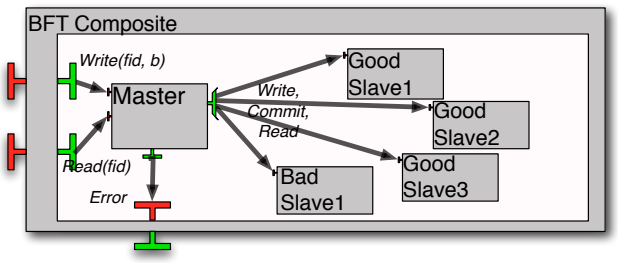


Figure 3.11: Our fault-tolerant application

From a behavioural modelling point of view, this article required us to provide a model for multicast interfaces where capabilities of pNets' synchronisation vectors were fully used and allowed one component to broadcast a request to several others, or one component to provide a reply that would reach the right index in a group of futures. We had to represent richer future proxies that were able to handle a list of results, and to provide a result as soon as some of them were resolved.

Figure 3.11 shows the architecture of our application as a GCM component system. Figure 3.10 shows the structure of the pNet we defined for representing this application, and for which we verified the behaviour by model-checking. Note that **Slave[k]** is a family of pNets representing all the slaves of the application, and which will be targeted by the multicast: the one-to-many synchronisation vector is symbolised by the circled **BC**. On the other side, the master asynchronously collects results from the different slaves, represented by the circled **CO** on the concerned synchronisation vectors.

Apart from the specification of a model for multicast interfaces with futures, this article was also the opportunity for us to investigate the use of distributed model-checking for verifying the properties of systems with a large number of states.

To summarise and to illustrate the kind of properties we are able to verify, we proved by model-checking that our application consisting of 1 master and 4 slaves (3 good ones and bad one) behaves correctly: 1) it answers to Read and Write requests: we proved both reachability and (fair) inevitability of termination of services, 2) the answers are correct in the sense that the read value is the value that has been written, 3) for this it relies on the slaves for storing the data (the master only performs a consensus), and 4) enough good slaves have been instantiated and the `NotBFT` error, signalling that a consensus could not be reached, cannot be raised.

In this work, we did not model reconfiguration and adaptation, but we design our specification in such a way that those aspects can easily be added to the model in the future: each element that can be modified by reconfiguration has a representation in our behavioural model.

3.7 Summary and conclusion

In this chapter we have mainly shown the efforts we have done in the design of a component model for large-scale computations. This model has been implemented, relying on the ProActive middleware, and provided a reference implementation for the GCM component model. This implementation has been a good starting point for experimenting and designing more advanced features of our component model: we proposed both a better structure for non-functional concerns, and improved reconfiguration capabilities, specialised for distributed components.

Our component model has been given an operational semantics inspired from the request/reply nature of ASP communications but much more general, as the behaviour of primitive components is highly parameterisable. This operational semantics was mechanised in the Isabelle/HOL theorem prover and allowed us to prove the correction of a strategy for updating futures.

Having such a framework to study the properties of the component model is a crucial point. It allows us to study the properties of our implementation and to prove the correctness of some design-choices. It also formalises and proves the properties the ProActive/GCM programmer can rely on. And finally, properties and formal specification justify the choices made when building the behavioural model of a component assembly. Those properties are thus also useful for ensuring the correctness of our behavioural verification approach.

Overall, the distinction is clear between theorem proving or paper proofs on one side that allow proving generic properties on the model and its implementation, and behavioural specification techniques on the other side that allow proving the correctness of a given GCM application.

One of the next steps we envision is the formalisation of the interplay between our formalisation of the component model, and the behavioural specification we build. Indeed, we plan to formalise the behavioural semantics of GCM components; this means formally specifying the semantics of pNets, and the translation of component systems into pNets. Then we will be able to prove formally the equivalence between the operational semantics of our component model, and its behavioural semantics. Finally this will prove the correctness of the behavioural semantics relatively to the GCM specification, but should also allow us to further and more precisely use theorem proving techniques to help the behavioural verification of application correctness.

Further future works, in particular dealing with component system descriptions and component system reconfigurations will be presented in Section 4.3.

Chapter 4

Current Works, Perspectives, and Conclusion

In this chapter I review a few of our current research topics that are the most promising. Section 4.1 presents our work on the design and formalisation of dissemination algorithms for CAN peer-to-peer networks, and Section 4.2 presents a programming model for multi-threaded active objects. Section 4.3 reviews some other research directions which are still at an earlier stage. Finally, Section 4.4 concludes this document.

4.1 Dissemination algorithms for CAN: design and formalisation

4.1.1 Context and objectives

In this work, we are interested in *Structured Overlays Networks* (SONs) that emerged to alleviate inherent problems of unstructured P2P architectures such as scalability, limited search guarantees,... In these systems, peers are organised in a well-defined topology where resources are stored in a deterministic location. The underlying geometric topology is used by the communication primitives and ensures their efficiency.

Our aim is to design an *efficient* (in terms of number of messages) and *correct* broadcast algorithm for the CAN overlay network. We want to use mechanical proofs to ensure the correctness of the studied protocols, with a much higher confidence than paper proofs which rely too often on "well known" properties or "obvious" steps that could reveal wrong or under-specified. We expect our framework to be general enough to study CAN networks by providing the formalisation of basic building blocks composing it. However, we are not interested in formalising the whole CAN protocol but rather we focus on the minimal set of abstractions needed to reason on communication protocols for CAN.

Proving properties on distributed algorithms could be done by specific formalisms for distributed systems, like TLA$^+$ [Ls03], however we chose a more general theorem prover to have better support for general reasoning. Indeed, reasoning on the structure of a CAN requires generic theorems that will be better supported by a general purpose theorem prover.

A *CAN* [RFH$^+$01] is a structured P2P network based on a d-dimensional Cartesian coordinate space labelled $\mathcal{D}$. This space is dynamically partitioned among all peers in the system such that each node is responsible for storing data, in the form of *(key, value)* pairs, in a sub-zone of $\mathcal{D}$. To store a (k, v) pair, the key k is deterministically mapped onto a point in $\mathcal{D}$ and the value v is stored by the node responsible for the zone comprising this point. The search for the value corresponding to a key k is achieved by applying the same deterministic function on k to find the node responsible for storing the corresponding value. These two mechanisms are performed by an iterative routing process starting at the query originator and which traverses its adjacent neighbours (a peer only knows its neighbours), and so on and so forth until it reaches the zone responsible for the key to store/retrieve.

We investigated on the existence and optimality of broadcast algorithm for CAN structured overlay networks. We look for a broadcast/multicast algorithm that is *efficient*, in the sense it minimises the number of messages exchanged between peers while still reaching every peer in the network. To our knowledge the main proposals for efficient broadcast in a CAN network, and the closest works to our, are M-CAN [RHKS01] and Meghdoot [GSAA04].

Another related problem is to perform a multicast instead of a broadcast, that is to say flood only some of the nodes. M-CAN reduces the problem of multicast to the one of a broadcast on another CAN network (interlinked with

the first one and containing only the nodes to be flooded). This ensures that an efficient broadcast protocol is sufficient. Our first goal is to design an efficient broadcast algorithm, then we can rely on M-CAN approach to flood only some of the nodes. However it might be interesting (as in Meghoot) to provide a multicast algorithm restrained to a zone of the *original* network, e.g. an hypercube, in that case no creation of an additional network is necessary, instead the broadcast is constrained by an hypercube that should be covered.

M-CAN [RHKS01] is an application-level multicast primitive which is almost efficient, but it does not eliminate all duplicates if the space is not perfectly partitioned and the dimension is greater than two. The authors measured 3% of duplicates on a realistic example. In a publish/subscribe context, Meghdoot [GSAA04], built atop CAN, also proposes a mechanism that totally avoids duplicates but require the dissemination to originate from one corner of the zone to be covered. Compared to those approach, our algorithm can originate from any node of the CAN and still remove all the duplicates.

Additionally, no existing dissemination algorithm atop CAN has been formally specified, and one of our main objective is this work is to design an algorithm *proven correct*.

Consequently, our work is also linked to the verification of DHT protocols. Borgström et al. [BNOG05] were interested in the verification of DHT protocols. As such, they formalised and verified a variant of Chord [SMK+01] in static settings (i.e. no churn) using CCS, a process algebra. In a subsequent work, Bakshi et al. [BG07] used π-calculus to prove the correctness properties of Chord in the pure-join model of the protocol. Zave, in her work [Zav09] proved the Chord protocol in its two models: the *pure-join* and *full*, using the Alloy analyser. She provides a rigorous correctness proof of the pure-join model and proved that the full model of the protocol is indeed not correct using lightweight verification methods. Pastry [RD01] was also the subject of a recent verification effort [LMW11], which focus was to ensure the correctness of Pastry's algorithm. The *join* and *lookup* protocols were specified using TLA$^+$ and the properties verified using the TLC model checker. After gaining confidence in their formalisation, the authors turned to TLAPS, a platform for the development and verifications of TLA$^+$ proofs and came up with a reduction theorem which reduces the global Pastry correctness properties to the invariants of the underlying Pastry's data structure.

To our knowledge, we are the first ones to formalise and prove some properties of an abstraction of the CAN overlay network using a theorem prover. This formalisation efforts should greatly increase the correctness and understanding of distributed algorithms for structured P2P networks, and the confidence one has in their correctness.

4.1.2 M-CAN: an almost-efficient algorithm

Let us first describe the dissemination algorithm proposed in M-CAN:

1. The source node, sends a message to all of its neighbours (in Figure 4.1, the grey node **I** is the source)
2. A node receiving a message from a neighbour along dimension i (in a given direction) will forward the message along dimensions 1...(i-1) and to the neighbour in the dimension i in the opposite direction from which it received the message. For instance, in Figure 4.1, node **B** will forward the message it received from **C** in dimension 1 in the +X direction; **C** forwards the message in three directions. Note that there are rules in order to avoid looping round the back of the space and the messages have sequence numbers so a node will not forward an already processed message.

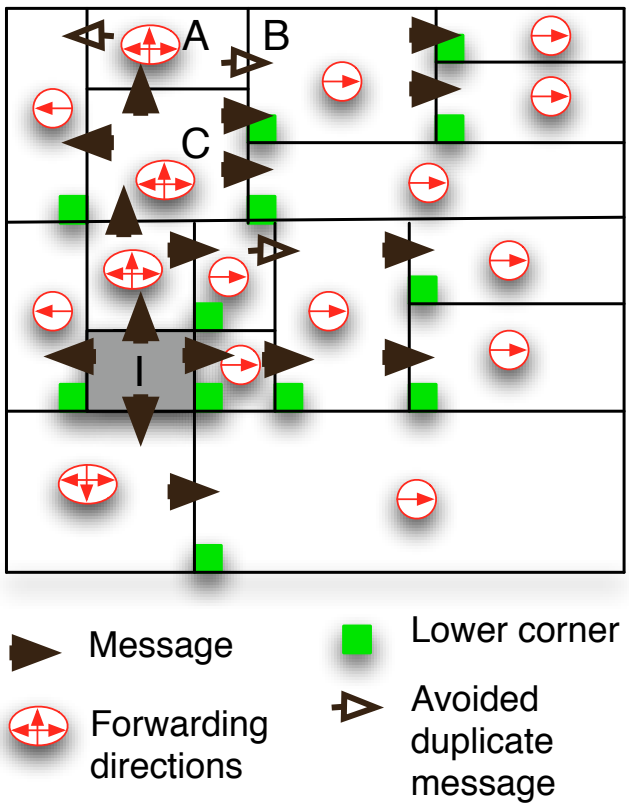


Figure 4.1: M-CAN - Message forwarding

3. For a perfectly partitioned space, where nodes have equal sized zones, the previously presented scheme avoid redundant messages. However, as you can see in Figure 4.1, nodes **A** and **C** should send a message to **B**, thus resulting in a duplicate. In order to remove certain of the duplicates, a deterministic rule can be used

between **A** and **C** since they known about each other and they are aware of each other's coordinates hence they can use this rule so only one node forwards the message to **E**. The idea is that a node only forwards the message if this node abuts the lowest corner of its neighbour (the lowest corner is the corner that touches the propagation direction and minimises all the other coordinates). In Figure 4.1, only node **C** forwards the message to **B** since it "touches" the lower leftmost corner of **E**. However, such a rule only removes duplicates arising from the first dimension and cannot be applied in higher dimensions. Indeed, because of the propagation rule, it is impossible to know if a neighbour is responsible for sending a message or not in a dimension greater than 1. The authors of M-CAN give the following example justifying why their approach is not valid in higher dimensions:

> "Consider a 3-dimensional CAN; if a node by the application of a deterministic rule decides not to forward to a neighbour along the second dimension, there is no guarantee that any node will eventually forward it up along the second dimension because the node that does satisfy the deterministic rule might receive the packet along the first dimension and hence will not forward the message along the second dimension." [RHKS01]

4.1.3 An optimal dissemination algorithm

In their paper, the authors of M-CAN did not manage to design an algorithm removing all the duplicates, even if they reached good performances with only 3% of the nodes receiving duplicates (those nodes generally receive 2 messages but can receive more). Our algorithm can be considered as an extension of M-CAN allowing to remove the duplicates arising in all the dimension, and reaching the optimum of 0 duplicated message. Our algorithm takes the idea of M-CAN but defines spatial constraints allowing us to remove duplicates appearing also in dimensions higher than one.

The idea is the following. We use the "corner criteria" of M-CAN for the first dimension on which everybody forwards. For preventing duplicates in the $d-1$ other dimensions, we constraint the algorithm to only send messages to nodes belonging to an hyperplane; each of the node of the hyperplane will then propagate the message on the first dimension. In the hyperplane, we can apply recursively our algorithm in a CAN of dimension $d-1$. Note that when the hyperplane becomes a line no duplicate can arise if we just follow the propagation direction (there is no more corner criteria).

Overall, if the propagation dimension is the dimension k, like in M-CAN, the message will be propagated in dimensions $1..k-1$ in all directions, and in one direction in dimension k: a node sends to the opposite direction from which it received. Then we have a spatial constraints on dimensions $1..k-1$, and we apply the corner criteria to the dimensions $k+1..d$ (we only send if the sender touches the minimal coordinates of the receiver in those dimensions). Except the propagation direction, each dimension is either constrained by the *spatial constraint* or by the *corner rule*.

In the example Figure 4.2, the *spatial constraint* is the higher bound of source node **I** on the X dimension, that is, X=10. **I** will only send a message along the vertical axis to neighbour(s) which intersect the line. For example, on the downward direction, **I** only sends a message to **D**; **E** will receive the message on the horizontal direction. Note here that the authors of M-CAN did not explain how such duplicates were avoided (there is no figure similar to Figure 4.2 in M-CAN article). Note here that in case the line is exactly the node border, we choose deterministically the side of the node to be taken. The *corner rule* is to send a message to a neighbour if the sender's lower bound on the other dimension is lower than our neighbour's. More formally, a node forwards a message if the following conditions are valid:

- when propagating on X:

$$Sender.LowerBound(Y) \leq Neighbor.LowerBound(Y) < Neighbor.UpperBound(Y)$$

- when propagating on Y:

$$Neighbor.LowerBound(X) \leq 10 < Neighbor.UpperBound(X)$$

Within the messages, the directions to be covered by the nodes are pictured by the red circled arrows. In the horizontal directions, the principle is similar to M-CAN, we simply added a spatial constraint preventing **E** from receiving duplicate messages.

As illustrated in Figure 4.3, this can be generalised to dimensions greater than 2 and thanks to our additional criteria, we still have no duplicate. In dimension 3 the initiator first sends messages to the nodes intersecting a plane. In this plane we reduce the problem to the example shown above (see Figure 4.2), in particular a line constraint is used and then a 2 dimensional corner rule is applied. Finally when propagating horizontally, a three dimensional corner constraint is applied as depicted in Figure 4.3.

We describe below the general algorithm in a less informal way. We give the data structure along with the

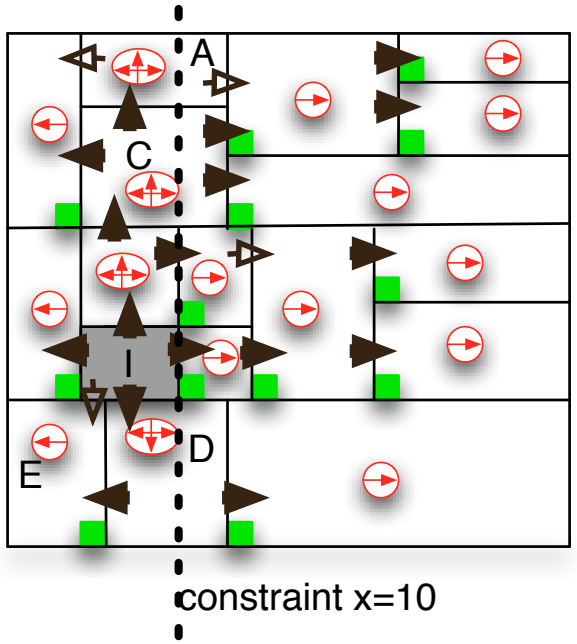


Figure 4.2: Principles of our algorithm in two dimensions

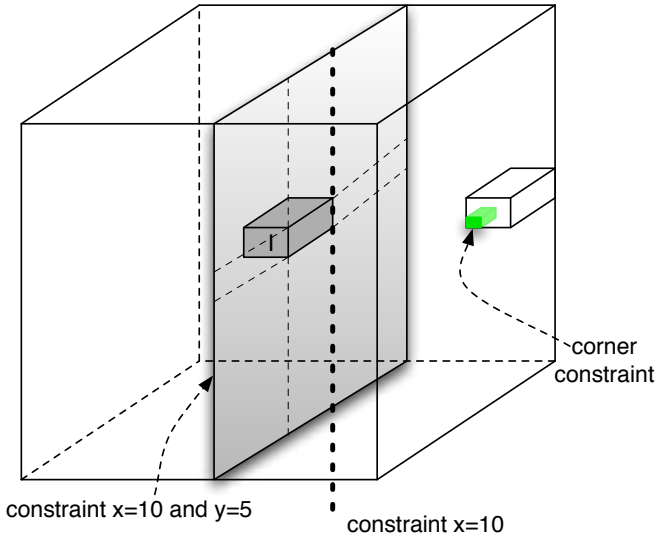


Figure 4.3: Our algorithm in higher dimensions

full algorithm below (Algorithm 4.1.1), which was implemented[1] using Java ProActive [CDdCL06]. We ran a series of preliminary micro-benchmarks on a static CAN (i.e. no peer churn) where we varied the number of peers (from 60 to 300) and the number of dimensions (from 2 up to 6). In every run, we found no duplicate message whatsoever confirming thus the efficiency of our algorithm. The data structures used in our algorithm are the following:

- **zone** $\stackrel{\text{def}}{=}$ (LowerBound, UpperBound); *the lower and upper bound of a peer's zone*
- **dimensions** $\stackrel{\text{def}}{=}$ [1 .. D]; *the dimensions each peer has*
- **side** $\stackrel{\text{def}}{=}$ inferior — superior; *the inferior/superior direction on a given dimension*
- **direction** $\stackrel{\text{def}}{=}$ (dimension, side); *the direction on which the message is received*
- **constraints** $\stackrel{\text{def}}{=}$ Array[D] of value; *An array of values representing the constraints used to decide to which neighbours the message is to be forwarded*
- **message** $\stackrel{\text{def}}{=}$ (direction, constraint, MessageValue); *The broadcast message during propagation*

Algorithm 4.1.1 Efficient broadcast algorithm

```
upon event ⟨message⟩ on node
 for each k≤message.direction.dimension do
  if k=D+1 then
   side← ∅
  else
   if k < message.direction.dimension then
    side← {inferior,superior}
   else
    side←message.size
   end if
  end if
  for each s in side do
   for each neighbor on dimension k and side s do
    for each i in 1 .. k − 1 do
     if not (neighbor.LowerBound[i] ≤ message.constraint[i]
< neighbor.UpperBound[i]) then
      skip neighbor
     end if
    end for each
    for each i in k + 1 .. D do
     if not(node.LowerBound[i] ≤ neighbor.LowerBound[i] <
node.UpperBound[i]) then
      skip neighbor
     end if
    end for each
     send ⟨constraint=message.constraint, direction=(k,s),
MessageValue=message.MessageValue ⟩ to neighbor
   end for each
  end for each
 end for each
end event
```

When receiving a message this algorithm forwards messages in all directions lower than the dimension on which the message has been received; in the dimension on which the message has been received, the message is only propagated on the same direction as the reception (not in both sides).

[1] https://bitbucket.org/lp/eventcloud-efficient-broadcast

As explained above, dimensions lower than the propagating direction are checked against the constraint, and dimensions higher than the propagating dimensions are checked against the lowest corner criteria. If the neighbor verifies all the criteria, the message is sent to it.

This algorithm is initiated by sending a broadcast message to an initiator node on a direction made of an artificial dimension $D+1$ and any side (this explains the first *if* condition of the algorithm). The message is associated with a constraint that should be the lowest corner of the initiator.

Did we loose robustness?

One can argue that having duplicated messages should increase the robustness of the algorithm in case a node fails, but we argue that there are much more efficient ways of duplicating the messages than the way imposed by the inefficiency of the algorithm. Indeed with the basic algorithm, some node receive the message once, while other can receive them an arbitrarily high number of times. For example, a much better way to ensure robustness would be to perform another broadcast of the same message with another source and reversing the dimensions of the CAN (considering the first one as the last one); this would ensure that each node receives each message *exactly twice* (instead of any number of times), most of the time coming from two different directions (instead of mostly coming from the same direction). One could also, based on the optimal algorithm, send an additional message to one neighbour, chosen randomly, which would already better distribute the duplicate messages that the basic flooding algorithm.

4.1.4 Formalisation of CAN and of dissemination algorithms

The following presents our mechanised formalisation efforts done in the context of dissemination algorithms over CAN. Our objective in this work is to provide a set of definitions and theorems to prove the properties of communication algorithms over CAN-like networks. This work will be illustrated by proving a set of properties for a class of broadcast algorithms. This formalisation has been realised in Isabelle/HOL[2].

A crucial question when formalising a complex structure like a CAN is which level of abstraction should be used, and which notions of Isabelle/HOL should represent basic notions of CAN networks. We represent a CAN by a set of nodes, a zone for each node, and a neighbouring relationship, stating whether one node is neighbour of another. More precisely, zones are represented by a function Z that matches each node to a *Zone*, where a zone is a *Tuple set* (a tuple is an array of integers). Note that we abstract away a few constraints of the CAN protocol:

- *Zones in a CAN are rectangular while in our formalisation a zone is a union of rectangles.* Thus our approach is more general than CAN. Also, in CAN, when a node leaves the network, a neighbour node will be responsible for several zones, in that case, a zone becomes a union of rectangles. Consequently our formalisation is better adapted to model a dynamic CAN with churn, i.e. the continuous joining/leaving of nodes.
- *We do not relate zones with the neighbouring notion.* Indeed, this aspect did not reveal to be useful in our proofs, and this approach alleviated us from geometric reasoning which would be difficult in Isabelle/HOL. Taking into account geometry will however be necessary in some cases and we could easily extend our formalisation with geometrical concerns. It seems reasonable to reason on geometry separately, and instead provide a set of properties of a CAN that are ensured because of geometrical constraints.

Overall our formalisation does not follow exactly the CAN protocol but our abstract notions of zones and neighbours are more general and provides more reasoning flexibility. In Isabelle, a CAN is defined as follows:

```
typedef CAN = {(nodes::nat set, Z::nat=>Zone, neighbours::(nat×nat) set).
finite nodes ∧ finite neighbours ∧
(∀ x y. (x,y)∈neighbours⟶(y,x)∈neighbours)∧ (*symmetric neigbhoring relationship*)
(∀ x. (x,x)∉neighbours) ∧  (*a neigbhor is unique*)
(∀tup. ∃n∈nodes. tup∈(Z n)) ∧(*every tuple is covered by a node*)
(∀ N∈nodes. ∀ N'∈nodes. N≠N'⟶¬intersects (Z N) (Z N')) ∧ (*no overlap*)
(∀ N∈nodes. Z N≠{})} (*a zone managed by a node is not empty*)
```

Additional constraints state that the set of nodes is finite and that the zones cover the whole space and are disjunct. We define three auxiliary functions *CAN_Nodes*, *CAN_Zones*, and *CAN_neighbours* returning each part of a *CAN*. We also define a function *intersects Z Z'* that checks whether zone Z intersects zone Z': it is true if Z and Z' have at least one point (tuple) in common. Then we define *NodesInZone C Z*, the set of nodes which zones intersect the zone Z, we say then that "the node N is in Z".

Then we define a notion of connectivity adapted to CAN zones. This notion is close to the geometrical notion of path connectivity but dedicated to the CAN networks. The idea is that a zone is connected if a message can go between one node in the zone to another node in the zone passing only through nodes in the zone. In the context of a communication protocol, a connected zone allows

[2]see: `www-sop.inria.fr/oasis/personnel/Ludovic.Henrio/misc`.

(indirect) communication between any two nodes of the zone. We state that a zone is connected if the nodes it intersects are all connected to one another (there is a path of neighbours between any two nodes intersecting the zone). The Isabelle definition of *Connected* is the following, it is a function that takes a *CAN* and a *Zone* and returns a *bool*. It states that if *n* and *n'* are two nodes in zone *Z*, then there is a list of nodes (all distinct) starting at *n*, ending at *n'*, only passing by nodes intersecting *Z*, and for which each node of the list is neighbour of the previous one.

```
definition Connected:: CAN⇒Zone ⇒bool where
Connected C Z≡∀ n∈NodesInZone C Z. ∀ n'∈NodesInZone C Z.
  ∃ node-list. node-list!0=n ∧ destination-NL node-list=n' ∧ dis-
 tinct node-list
       ∧ (∀ i<length node-list − 1. node-list!i∈CAN-Nodes C
            ∧ CAN-neighbour C (node-list!i) (node-list!(i+1))
            ∧ node-list!i∈NodesInZone C Z)
```

To reason on CAN structures, we provide several generic lemmas. They will be used in most further proofs. The following lemma is particularly useful to us as we will see that we will reason on paths of messages constrained inside a zone; it allows us to initiate a path inside a connected zone. It states that if the zone contains more than one node, then one can find two nodes, neighbour of one another, inside the zone:

```
lemma Connected-exists-neighbour:
 ⟦Connected C Z; ZoneSize C Z>1⟧
    ⟹∀ N∈NodesInZone C Z. ∃ N'∈NodesInZone C Z.
 CAN-neighbour C N N'
```

Let us conclude this section with an induction principle that allows one to prove a property related to a zone by induction on the size of the zone. A trivial induction lemma would express directly induction on the number of nodes in the zone on which the property is verified. More interestingly, one can prove a property by adding one by one the node belonging to the zone of interest; this allows some form of structural induction on a CAN zone:

```
theorem induct-node-zone-2:
⟦ P {};
  ⋀Z. (P Z⟶(∀ N∈CAN-Nodes C. N∉NodesInZone C Z⟶
        (∀ Z'.(NodesInZone C Z'={N}⟶ P (Z∪Z')))))  ⟧
  ⟹ P Z
```

This theorem states that, if (1) we prove that a property P is true for an empty zone, and (2) we prove that if P is true for a zone then it is true for a zone intersecting one more node; then the property is true for all zones.

Messages and message paths

This section describes the formalisation of messages and of the path followed by a message. A message is made of four parts: an identifier for the message (which could represent also its content), a source node, a destination node, and the zone to which it must be transmitted:

```
types Message = nat × nat × nat × Zone
```

We decided to rely on the notion of zone to be covered to define a broadcast algorithm, because it seems quite adapted to the CAN structure. This zone to be covered can have two purposes depending on the algorithm. First it allows the specification of multicast protocols where only the nodes in a given zone have to receive the message. Also, as we are looking for an efficient algorithm that minimises the number of messages necessary to broadcast the information, it seems reasonable to split efficiently the zone to be covered in order to avoid sending a message to the same node twice. *Message-zone*, *Message-dest*, and *Message-source* are functions accessing the first three fields. We also define an abbreviation $<m|x,y,Z>$ for defining a *Message*, this allows us to easily identify messages inside the definitions and lemmas.

In the context of CAN, it seems crucial to provide tools to reason on the path followed by a message. Indeed, communication inside CAN heavily relies on the notion of paths. For this, we define a path as a loop-free set of consecutive messages, and provide tools to reason inductively on those paths. Consider a message set *msgs*, a list of messages forms a *valid-path* if each message is sent from the destination node of the previous message. We only want to reason on paths of finite length, and also it seems reasonable to reason on the longest path in a zone. For those reasons, we consider only loop-free paths: all the elements of the list must be distinct.

```
definition valid-path:: Message set ⇒Message list ⇒ bool where
valid-path msgs ML ≡  ML≠[] ∧(∀ i<length ML. ML!i∈msgs) ∧
        (∀ i<length ML − 1. Message-dest (ML!i)=Message-source
 (ML!(Suc i)))
          ∧ distinct ML
```

The predicate *path-inside-zone* takes a CAN, a set of messages *msgs*, and a zone *Z*, and returns the set of valid message paths formed of messages that are entirely inside zone *Z*. For this, we check that the origin node of the path is in zone *Z*, and that the destination node of each message of the path is in zone *Z*.

```
definition path-inside-zone::CAN⇒Message set⇒Zone⇒Message
 list set where
path-inside-zone C msgs Z ≡
   {MsgL. valid-path msgs MsgL ∧ source MsgL∈NodesInZone C Z
 ∧
     (∀ i<length (MsgL). Message-dest (MsgL!i)∈ NodesInZone C
 Z)}
```

Specification of a class of broadcast algorithms

From this formalisation of a CAN network, our objective is to reason about broadcast algorithm. As a first step, we

formalised a class of algorithms based on the notion of *zone to be covered*. Defining a broadcast in a convincing way using Isabelle/HOL is not trivial; this is mainly due to the underlying functional language, similar to λ-calculus, which is probably not the best language for defining a broadcast algorithm. We will put an emphasis on the way messages are processed. Our formalisation is centred around the specification of messages which are the *consequences* of a given message and on the specification of the *set of messages* used to broadcast the original message. Then we define the way messages are broadcasted by an inductive definition, where messages are "treated" one message after the other sequentially. In our framework, a *Broadcast* is a triple made of a *CAN*, a message set and an initiator node constrained by several well-formedness rules as defined below:

```
typedef Broadcast = {(can,msgs,initiator).
(∀ x y m Z m' Z'. (<m|x,y,Z>∈msgs∧<m'|x,y,Z'>∈msgs)⟶
          (m=m' ∧ Z=Z')) ∧
(initiator∈CAN-Nodes can)∧
  (∀ m s d Z. <m|s,d,Z>∈msgs ⟶
   (s∈ CAN-Nodes can ∧ d∈ CAN-Nodes can ∧
   CAN-neighbour can s d ∧
    (s=initiator ∨
      (∃ MsgL. valid-path msgs MsgL∧destination MsgL=s∧source
  MsgL=initiator))
)) }
```

The constraints expressed in the above definition state that: (1) There is a single message between any 2 nodes. (2) The initiator is a node of the CAN. (3) All messages are exchanged between neighbour nodes of the CAN, and thus the broadcast pattern respects the CAN protocol. (4) All messages must be sent by a node that has been reached by a list of messages originating from the initiator: *valid-path msgs MsgL* $\wedge$ *destination MsgL* $=$ *s* $\wedge$ *source MsgL=initiator*. Requiring the existence of such a valid path ensures that a broadcast only relies on messages transmitted from nodes to nodes, and no message is spontaneously created (except for the origin of course). We denote $<C,M,n>$ a *Broadcast*, and define functions *BC-CAN*, *BC-msgs*, and *BC-initiator* to access its fields. We can then define a predicate checking whether a broadcast covers the whole CAN, or more precisely whether each node of the CAN is either the broadcast initiator or the destination of a message:

```
definition Coverage:: Broadcast⇒bool where
Coverage BC ≡ ∀ n∈CAN-Nodes (BC-CAN BC).
          (n=BC-initiator BC ∨ (∃ m s Z. <m|s,n,Z>∈BC-msgs
  BC))
```

From those definitions, we expect to prove coverage for some specific broadcast algorithm, but also study their optimality. We focus on broadcast algorithms that rely on zones to be covered by the consequence of a message. The

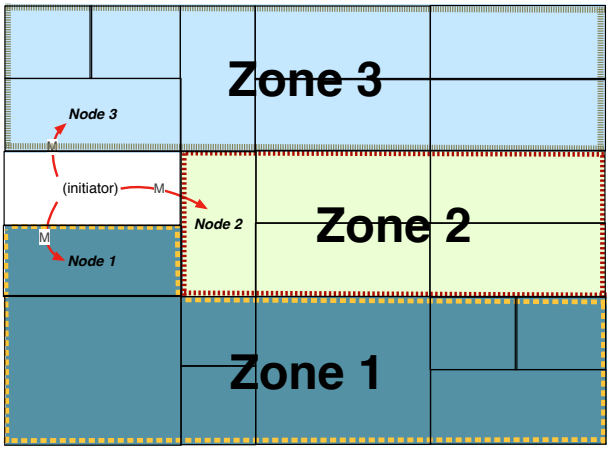


Figure 4.4: Zone node list (ZNL) definition

idea is that each message is given a zone and the messages that are triggered by this message must cover this zone, but should not pass by nodes outside this zone. Then the broadcast algorithm will be entirely characterised by a function that, given a node N receiving a message and the zone Z to be covered by the message returns a list of couples (*Zone*, *Node*), that we call ZNL (*Zone-Node list*). Each couple (Z_i, N_i) of the ZNL consists of Z_i a sub-zone of Z and N_i a neighbour of N belonging to Z_i: the message is forwarded to N_i that is now responsible for covering Z_i. The zone Z_i must be connected as defined in the previous section, else it would be impossible to cover it while staying inside Z_i. Given a node and zone, we define the set of ZNLs that ensure an optimal broadcast: $Set-of-Optimal-ZNL$. The definition (omitted here) guarantees optimality by the fact that no two nodes receive the same message and no node belongs to two different zones. Of course, the partition should also cover the whole space. For the moment our objective is not to show if such a partition of zones is easy to build, nor to prove its existence but we explain informally below how to build such a zone. Figure 4.4 illustrates the notion of ZNL for the first step of the broadcast: the message to be broadcasted is sent from the initiator. We split the zone to be covered into 3 zones, 3 messages will be sent to three nodes belonging to each zone and which are neighbours of the initiator. Each node will be responsible for broadcasting the message within its zone.

The following definition specifies the set of messages of a broadcast algorithm based on a *ZNLmap*, which is a function that given a node and zone returns an optimal ZNL. The inductive definition of the broadcast is of the form *BC-msgs C Mid init znlmap msgs ML* where *C* is the CAN network, *Mid* is the message identifier, *init* is the initiator node, and *znlmap* is the *ZNLmap* used by this instance of the algorithm. The inductive definition

works as follows. It takes one by one messages in *ML*, the list of message to be treated. Once treated, the message is put inside the set *msgs*. At the end, *ML* is the empty list, and *msgs* contains all the messages of the broadcast. Here processing a message consists in computing the messages that are consequences of this message thanks to the *znlmap* function, and putting them in the "message to be treated" list. Then the original message can be considered as treated and be put in the set of treated messages.

inductive
ZNL-BC-msgs::*CAN*⇒*nat*⇒*nat*⇒*ZNLMapping*⇒*Message set*⇒*Message list*⇒*bool*
for *C*:: *CAN* **and** *Mid*::*nat* **and** *init*:: *nat* **and** *znlmap*::*ZNLMapping* **where**
BC-step: ⟦*ZNL-BC-msgs C Mid init znlmap msgs* (*M*#*ML*); *M*=<*Mid'* | *s*,*d*,*Z*>;
ML'=*map* (λ*ZN*. *let Z'*=*fst ZN in let N'*=*snd ZN in* <*Mid'*|*d*,*N'*,*Z'*−*CAN-Zones C N'* >) (*znlmap Z d*)⟧ ⟹ *ZNL-BC-msgs C Mid init znlmap* (*insert M msgs*) (*ML*@*ML'*)

The rule above treats one message *M*=<*Mid'*|*s*,*d*,*Z*>; it computes the messages the node *d* has to send by producing a message for each member of the list *znlmap Z d*. The list of produced messages is pushed in the list of messages to be treated. There exists a similar rule (not shown here) for computing the messages sent by the initiator node. These rules are applied iteratively treating one message after the other. At the end, the list of messages to be treated is empty, and the fourth argument contains the list of messages of the broadcast. Those rules illustrate how we suggest to define the message propagation; note that even for a broadcast algorithm that would not rely on coverage zones, the inductive structure of the message set definition would be similar.

Here some design choices have been made in the way messages are computed. While treating messages one after the other seems adapted to Isabelle/HOL, this of course does not correspond to the parallelism that occur in a real system, but this is quite classical and has no consequence as long as we do not want to evaluate the time needed to broadcast the message. More important is the fact that the messages to be treated are totally ordered (they are represented as a list), this total ordering is artificial and one could improve the representation by defining an equivalence relation allowing messages to be re-ordered. However the list is a good structure to reason inductively on the messages and to allow rules to be applied iteratively. As total ordering was a too strong relation, the treated messages are defined as a set instead of a list. Note that a set is considered as sufficient as it is easy (e.g. thanks to a message cache) to prevent the same message to be sent twice between the same nodes.

We can prove that the set of messages generated by an optimal ZNL constitutes a valid broadcast, as stated by the theorem below.

theorem *ZNL-BC*:
⟦ *ZNL-BC-msgs C Mid init znlmap Finalmsgs* []; *init*∈*CAN-Nodes C*;
∀ *N Z*. *znlmap Z N* ∈ *Set-of-Optimal-ZNL C N Z*⟧⟹
(*C*, *Finalmsgs*, *init*)∈*Broadcast*

Note that we consider the last step of induction, when the list of messages to be treated is empty ([]). Also, we require that for each zone and node, *znlmap* verifies *set-of-Optimal-ZNL*. Finally we prove coverage for this broadcast.

theorem *coverage-ZNL*:
⟦ *ZNL-BC-msgs C Mid init znlmap Finalmsgs* []; *init*∈*CAN-Nodes C*;
∀ *N Z*. *znlmap Z N* ∈ *Set-of-Optimal-ZNL C N Z*⟧⟹
Coverage (<*C*, *Finalmsgs*, *init*>)

Discussion

The current specification and proofs consist of more than 2500 lines of Isabelle, for more than 100 lemmas and theorems. Most of the code is dedicated to proofs, but our framework requires a lot of different notions, and the definitions amount for more than 10% of the code.

Our specification provides a convenient level of abstraction for reasoning on communication algorithm while abstracting away most of the geometrical concerns. The structured network represented is slightly more general than a pure CAN: in a CAN zones are necessarily hypercubes, whereas ours could be in principle any tuple set, for example a union of hypercubes. We prefer relying on a less restrictive definition of the structure in order to see which properties of our algorithm are verified in those conditions and also to better adapt to node churn. Later additional requirements on the structure can be added to prove further properties, e.g. a broadcast algorithm may only be efficient if the zones are hypercubes.

It is important for us to have a formalism for expressing the CAN broadcast that is easy to understand. Although the specification we showed here is inductive and thus not in a classical form for a broadcast algorithm, we think it is clear enough to be convincing, and that it is easy to extract an algorithm from it. This way of expressing a broadcast algorithm is not as natural as one would expect because a form of event-based formulation of the algorithm ("when a message M is received, send messages M1, M2, and M3") would be more adapted. However, such an event-like formulation is not well supported in Isabelle/HOL. Providing new abbreviations for expressing message transmission more easily is outside the scope of our work for the moment.

The obvious next steps of this work, after polishing and finishing generic proofs of our framework will be to experiment our optimal algorithm on a large-scale basis, but also to mechanically prove its correctness and efficiency, based on our Isabelle/HOL framework.

4.2 Multi-active objects

As mentioned in the overview of active object languages, Section 2.2, existing frameworks for active objects suffer from some weaknesses, in particular:

- *Deadlocks can occur upon re-entrance of requests* or mutually recursive requests: ASP's active objects are almost systematically deadlocked in this case (except when first-class futures are sufficient to avoid deadlocks), while Creol and JCoBox rely on cooperative thread release (await statements) to prevent such deadlocks; but cooperative multi-threaded might introduce an interleaving of threads difficult to control.
- Active objects are in general inefficient on multi-core machines. Indeed, active objects unify the notion of thread, of location, and of objects, consequently there is a single thread manipulating the active object at each moment. Consequently, at deployment one has two choices, either there is one active object per machine, and thus a single active thread per machine which is inefficient, or there are several active objects per machine and thus several active threads on each machine, but then those objects do not share memory and thus communication between objects in the same machine is inefficient.

In fact, JCoBox proposes a shared immutable state that can be used efficiently on multi-core architectures but as the distributed implementation is still a prototype, it is difficult to study how an application mixing local concurrency and distribution would behave.

In order to overcome those limitations, we propose an active object language efficient on multi-core machines, on which deadlocks due to re-entrance can easily be avoided. Our programming model also provides a precise control over concurrency. We call this programming model *multi-active objects*. This language adapts concurrency annotations *à la* JAC [HL06] to an active object language *à la* ASP (see Section 2.2 for the description of the different frameworks). We explain below the principles of our framework, while illustrating it on the implementation of a CAN network based on active objects.

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

4.2.1 Assumptions and Design Choices

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

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

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

We assume here that the programmer defines groups and their compatibility relations inside a class in a safe manner. Of course dynamic checks or static analysis should be added to ensure, for example, that no race condition appear at runtime. However, we leave such an extension to the framework for future works, and decide to focus for the moment on the programming model itself.

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

To illustrate our approach, we describe below a possible design methodology for transforming an active object into a multi-active object. Without annotations, a multi-active object behaves similarly to an active object, no race condition is possible, but no local parallelism is possible. If some parallelism is desired, e.g. for efficiency reason or because dead-locks appeared, each remotely invocable method can be assigned to a group and two groups can be declared as compatible if no method of one group accesses the same variable as a method of another group. In that case, method of the two groups will be executed in parallel and without respecting the original order, meaning the groups can only be declared compatible if additionally the order of execution of method of one group relatively to the other is not significant. If more parallelism is still required, the programmer has two non-exclusive options: either he protects the access to some of the variables by a locking mechanism which will allow him to declare more groups as compatible, or he realises that, depending on runtime conditions (invocation parameters, object's state, ...) some groups might become compatible and he defines a compatibility function allowing him to decide at runtime which request executions are compatible.

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

4.2.2 Defining Groups

The programmer can use an annotation (`Group`) to define a group and can specify whether the group is `selfCompatible`,

```
@DefineGroups({
  @Group(name="join", selfCompatible=false)
  @Group(name="routing", selfCompatible=true)
})
public class Peer {
...
@MemberOf("join")
  public JoinResponse join(Peer other) { ...
      }

@MemberOf("routing")
  public void add(Key k, Serializable value)
      { ... }

@MemberOf("routing")
  public Serializable lookup(Key k) { ... }
}
```

Figure 4.5: The CAN Peer annotated for parallelism

i.e., two requests on methods of the group can run in parallel. The syntax for defining groups in the class header is:

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

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

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

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

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

Figure 4.5 shows how these annotations are used in the Java class implementing a CAN peer in which *add*s and *lookup*s can be performed in parallel – they belong to the same self-compatible group `routing`. Since there is no compatibility rule defined between the groups, methods of `join` and `routing` will not be served in parallel. To fully illustrate

our annotations, suppose that *monitoring* was a third concern independent from the others; declaring it would require to add the following lines:

```
...   @Group(name="monitoring",
    selfCompatible=true)
...
@DefineRules({
  @Compatible({"join", "monitoring"})
  @Compatible({"routing", "monitoring"})
})
```

We chose annotations as for defining parallel compatibility because these are strongly dependent on the application logic, and in our opinion, should be written at the same time as the application.

4.2.3 Scheduling Request Services

In ASP, AmbientTalk, etc. requests are served one by one, and if no particular service policy is specified, they are served in a first come first served order. In multi-active objects, even though we focus on increasing the parallelism inside active objects, we also provide guarantees on the order of execution.

A first policy, called *FIFO policy*, serves the requests in the order that they arrive, and provided that the first request in the queue is compatible with the currently served ones, it will be served in parallel with them. However, this solution does not always ensure maximum parallelism inside multi-active objects. Consider the following example. Inside a CAN peer there are two concurrent *add* operations running and there are two requests in the queue: *join* and *monitor* (member of the group `monitoring`). Using the FIFO logic presented above, we would not be able to start any more requests until the two *adds* finish. However, the *monitor* request is compatible with all the others, and it could be safely executed before *join*, concurrently with *adds*. This leads us to an *optimised policy* for scheduling the request services: a request can be served in parallel if *it is compatible with all running requests and all the requests preceding it in the queue*. This policy ensures maximum parallelism while maintaining the relative ordering of non-compatible requests.

4.2.4 Dynamic Compatibility

Sometimes it is desirable to decide the compatibility of some requests at run-time, depending on the state of the active object, or the parameters of the requests. For this reason we extend the groups with the notion of a *group-parameter*. This parameter is common to all methods belonging to the group (they might have other parameters as well). For instance, in Figure 4.6, all methods belonging to the *routing* group of the CAN application have the key they act on as a parameter. The group-parameter is identified by its type (`parameter="someType"`), and in case a method has several parameters of this type, the leftmost one is chosen. For example the routing group can be given a parameter of type `Key`.

```
@Group(name="routing", selfCompatible=true,
    parameter="Key"),
```

We considered that choosing a single parameter of the methods belonging to a group was the simplest and most convenient solution. Of course, if several parameters of a method are needed to decide compatibility, they can be wrapped in a single object.

Then, a condition can be added to a compatibility rule in the form of a *compatibility function*. A compatibility function takes as input the common parameters of the two compared groups, and returns *true* or *0* if the methods are compatible (*false* or any other integer otherwise). The syntax for a dynamic compatibility rule is:

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

Compatibility functions are resolved at runtime, based on how they were defined:

1. as a method of the parameter – if `SomeCondition` is of the form `someFunc`. In this case the comparator function call is of the form `param1.someFunc(param2)` where `param1` is the parameter of one request and `param2` is the parameter of the other.
2. in another object – if `SomeCondition` is of the form `[REF].someFunc`. The condition is a method that is invoked with both parameters as arguments (`someFunc(param1, param2)`). `[REF]` can be either `this` if the method belongs to the active object itself, or a class name if it is a static method.

Additionally the result of the comparator function can be negated using "!" as a prefix to `SomeCondition`, e.g. `condition="!equals"`.

If there is a single order of group parameters for which the compatibility function exists then this function is called. Otherwise, the order of parameters is unspecified and any function of the right name and signature is called, this is why, in that case, the compatibility method should be symmetrical. Sometimes the decision of compatibility may not depend only on the parameters, but also on the state of the active object. If the compatibility function is a method of the active object, it can then access its fields; in that

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

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

Figure 4.6: The CAN Peer annotated for parallelism with dynamic compatibility

case mutual exclusion with the currently executing threads should be ensured by the programmer. If one group have no common parameter, the compatibility function must have one less argument. In case both groups have no parameter the compatibility function has to be either a static method or a method of the active object, with no argument.

As an example, we show below how to better parallelise the execution of *join*s and routing operations in our CAN use-case. During a join operation, the peer which is already in the network splits its key-space and transfers some of the key-value pairs to the peer which is joining the network. During this operation, ownership is hard to define. Thus a *lookup* (or *add*) of a key belonging to one of the two peers cannot be answered during the transition period, as the result would be non-deterministic. Operations that target "external" keys, on the other hand, could be safely executed in parallel with a join. Figure 4.6 shows how to modify the peer based on this last remark, more precisely:

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

Adding a self-compatibility rule

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

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

4.2.5 Inheritance

It would be infeasible to re-declare compatibility information every time a class is extended. Therefore we designed our annotations to have an inheritance behaviour similar to Java classes: implicitly, parallel behaviour is inherited with the logic, but the programmer can add or override definitions in the subclass if necessary.

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

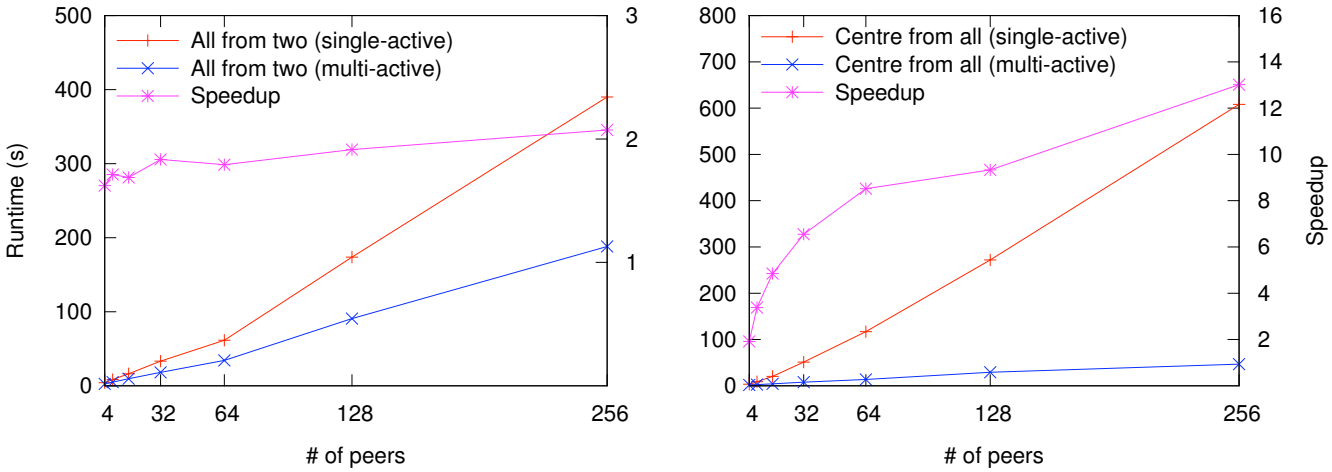


Figure 4.7: CAN experimental results

4.2.6 Experimental results

We extended the ProActive library with multi-active objects presented above.[3] Then we conducted several experiments including the parallelisation of CAN peers. We measured the benefits of `lookup` request parallelisation in the following situations:

- "All from two" (Figure 4.8) – In this scenario, we added an equal number of key-value pairs to all the peers in the network. We then used two peers, located at opposite corners of the CAN overlay to lookup all those values. Each corner sends `lookup` requests one after the other. However, the results are all awaited at the end thanks to the use of futures. This experiment gives an insight about the overall throughput of the overlay.

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

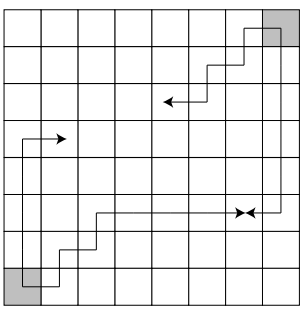


Figure 4.8: CAN routing from two corners

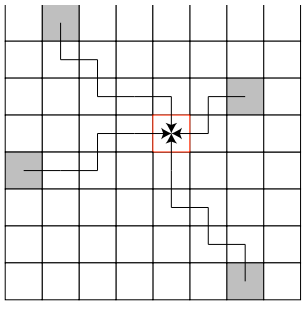


Figure 4.9: All nodes accessing centre

- "Centre from all" (Figure 4.9) – In this test case, all the peers lookup concurrently a key situated in a peer at the centre of the CAN. This experiment highlights the scalability of a peer under heavy load.

Figure 4.7 shows the execution times and speedup for several sizes for the two scenarios, both scenarios achieve significant speedup. However, the gain in the first scenario is smaller because the lookups are issued from the two corners in sequence; the sequential sending of the initial lookups limits the quantity of parallel lookups present at the same time in the network. On the contrary, in the second case, the active object version has a bottleneck because the centre peer can only reply to one request at a time whereas those requests can be highly parallelised with our model. These speedups are achieved by just adding a few simple annotations to the class declaration without changing any of the implemented logic.

4.2.7 Discussion

Active objects have been extended with annotations to allow and control multi-threaded execution inside them. The annotations can be written from a high-level point of view by declaring compatibility relations between the different concerns an active object manages. Relying on those annotations, request services can be scheduled such that parallelism is maximised while preventing two incompatible requests from being served in parallel. We showed that multi-active objects outperform simple active objects, and, in another experiment, we showed that multi-active objects

greatly reduce the number of lines dedicated to local concurrency. Overall we think multi-active object provide a good compromise between efficiency and easy writing of both concurrent and distributed objects. Multi-active objects provide the efficiency of local concurrency while benefiting from the easiness of distribution provided by active objects.

In the future, we plan on conducting further experiments on multi-active objects but also extending the programming model with better support for local concurrency. We wrote an operational semantics for multi-active objects, and proved on paper the first correctness properties of the model. Next steps could include the extension of the mechanised formalisation presented in Section 2.4 to model and prove properties of multi-active objects, but also the adaptation of determinism properties of ASP to multi-active objects, those new properties would necessarily be weaker but would allow a precise identification of the sources of non-determinism in our new programming model.

4.3 Future works on distributed components

The preceding sections presented current and future works, both in the domain of the active object programming model, and in a relatively new direction for me: the verification of distributed systems. Without the aim of giving an exhaustive list of research directions, I want to cite here a few of them related to distributed components.

Verification of reconfiguration procedures

Until now we almost never used our behavioural models for the verification of reconfiguration of component systems. Whereas most of the structure for performing such verifications are already available in our behavioural models, the state-space of a reconfigurable component system seems in general too big for allowing model-checking. However, our recent results provided us with better tools for harnessing this state-space explosion problem. Already the fact that we have been able to run distributed model-checking allowed us to verify bigger systems, also our recent works on verification of group of components required us to face systems with higher number of components. Thus it seems reasonable now to be able to specify and verify reconfiguration procedures on simple systems.

Also concerning component reconfiguration, our semantics for GCM (see Section 3.4) seems a good basis to study the properties of the reconfiguration of distributed components. Proving generic properties (either mechanically or on paper) of distributed reconfigurable components would be a useful tool in the design, implementation, and optimisation of our component middleware.

Also a promising research direction consists in considering higher-level reconfiguration primitives (like the replacement of a component by an equivalent one, or the duplication of a component) than the ones proposed by Fractal's API. Then properties of those primitives could be proved, probably based on the generic properties mentioned above. Those properties could be use to provide an optimised (i.e. smaller) behavioural model and then allow us to verify real-size applications.

Considering the fact component reconfiguration is a cornerstone of the autonomic framework developed in the OASIS team, being able to prove generic properties, but also application-related properties on such an evolving, autonomic application would be of particular interest to us.

parameterised ADLs

In GCM, its implementation, and our behavioural models, we deal with large-scale infrastructures in terms of groups of distributed components interconnected by collective interfaces. Communications between large groups of distributed components involve multicast and gathercast interfaces.

However, first the tools we have for defining families of similar components in the GCM (e.g. inside the ADL) are quite weak. More generally, we miss high-level abstractions for defining topologies of components (e.g. rings, matrices, etc.), and for their connections. These abstractions will have first to be be reflected at language and middleware level, typically in the component architecture description language (ADL) and in the middleware API: once generic topologies can be defined in the ADL, there must be a way to instantiate them at deployment time, depending on the quantity of machines available, or of the desired size for the system.

Similarly, the specification platform must also reflect those features, for example as new graphical constructs in the Vercors environment. Then the semantics of those constructs will be given in terms of parameterised model generation. For this the pNet model, by its parameterised nature is particularly well adapted.

One very appealing direction of this work could be the combination with the emerging research on component system reconfiguration. In the case of high-level group topologies, specific reconfiguration primitives should be defined that could bring more confidence in the safe behaviour of the application. Typically one could define what is required to safely insert a new component in a ring topology and

reconnect the bindings to guarantee a proper behaviour. Proving the correctness of such reconfiguration procedures will most probably rely on the semantics of GCM we defined in Section 3.4, and this work could be a good opportunity to tighten the links between our mechanised models and our behavioural models of GCM components.

4.4 Conclusion

This document presented most of my research achievements over the past six years. Most of those works have been realised in collaboration with researchers, PhD students, and interns of the OASIS team. Also some of them were the result of joint works with researcher of other universities. My research domain is mostly focused on distributed and concurrent computing, and more particularly on programming languages and their application to programming and execution environments. More recently, our works on peer-to-peer systems were more situated in the domain of distributed systems, and their theoretical foundations.

The overall objective of my work is to enforce the correct execution, but also to ease the programming of distributed applications. To reach this goal, I first presented my contributions in the domain of active object languages and the related execution environments. This gives to the programmer a language with strong guarantees, and also where writing programs is relatively easy. We also identified some weaknesses of this programming model and presented an improvement of the model called multi-active objects.

While active objects are adequate basic constructs for writing independent pieces of an application, some programming models are better adapted for the composition of those pieces of applications. Consequently, we came up with a composition model for distributed applications based on distributed software components. The GCM is a component model adapted to the composition of large-scale distributed applications, and its reference implementation, ProActive/GCM, illustrates well that those components are well-adapted to the composition of active objects.

With active objects, and even more with components, the programming model enforces a well-identified structure on the application that helped us in the specification and the verification of its correct behaviour. Finally through several works, in the domain of fault-tolerance and of peer-to-peer systems for example, I also contributed to the correctness of the execution environment for those applications. All those aspects contribute to my mind to an environment where distributed applications can be written easily and executed safely.

I think that my work illustrates well the interplay between theoretical foundations and practical system implementation. The models presented in this document have all been designed with the constraint to be as close as possible to a real language, API, or middleware. For this reason, those models were sometimes less nice and bigger than the ones usually found in theoretical developments. Also our results are sometimes not as strong as they would be if they were strictly focused on theory, but most of them are constrained by their direct applicability. They are also constrained by the compromise the programmer is willing to accept so that his programming and execution environment is both correct and efficient. Overall, most of the theoretical models presented in this document have been implemented, or have been written as a formalisation of an already existing implementation.

Our works on algorithmic skeletons illustrate well this interplay between theory and practice on a relatively small-scale whereas our achievements on active objects and components involve much bigger contributions. Consequently, in the domain of active objects and component oriented programming, the relation between theory and practice is more expressed by numerous links rather than by articles mixing theoretical results and their direct applications.

Chapter 5

List of Publications

Most of my publications are available at `http://www.inria.fr/oasis/Ludovic.Henrio`.
In almost all my papers, authors are ordered alphabetically, or alphabetically by institution.

Book:

[1] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects*. Springer-Verlag, 2005.

Book Chapters:

[2] Maciej Malawski, Marian Bubak, Françoise Baude, Denis Caromel, Ludovic Henrio, and Matthieu Morel. Interoperability of grid component models: GCM and CCA case study. In Thierry Priol and Marco Vanneschi, editors, *Towards Next Generation Grids: Proceedings of the CoreGrid Symposium*, pages 95–105. Springer US, 2007. 10.1007/978-0-387-72498-0_9.

[3] Francoise Baude, Denis Caromel, Ludovic Henrio, and Paul Naoumenko. A flexible model and implementation of component controllers. In *Making Grids Work – Post-Proceedings Selected Papers From The Coregrid Workshop On Grid Programming Model, Grid And P2p Systems Architecture, Grid Systems, Tools And Environments, June 2007*, pages 31–43. Springer US, 2008. 10.1007/978-0-387-78448-9_3.

[4] Maciej Malawski, Tomasz Gubala, Marek Kasztelnik, Tomasz Bartynski, Marian Bubak, Francoise Baude, and Ludovic Henrio. High-level scripting approach for building component-based applications on the grid. In *Making Grids Work – Post-Proceedings Selected Papers From The Coregrid Workshop On Grid Programming Model, Grid And P2p Systems Architecture, Grid Systems, Tools And Environments, June 2007*, pages 309–321. Springer US, 2008. 10.1007/978-0-387-78448-9_25.

[5] Antonio Cansado, Denis Caromel, Ludovic Henrio, Eric Madelaine, Marcela Rivera, and Emil Salageanu. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*, chapter A Specification Language for Distributed Components implemented in GCM/ProActive. Springer, 2008. `http://agrausch.informatik.uni-kl.de/CoCoME`.

Journals:

[6] Isabelle Attali, Denis Caromel, Carine Courbis, Ludovic Henrio, and Henrik Nilsson. An integrated development environment for Java Card. *Computer Networks*, 2001.

[7] Françoise Baude, Denis Caromel, Christian Delbé, and Ludovic Henrio. Un protocole de tolérance aux pannes pour objets actifs non préemptifs. *Technique et Science Informatiques*, 2006.

[8] Tomás Barros, Rabéa Ameur-Boulifa, Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Behavioural models for distributed fractal components. *Annales des Télécommunications*, 64(1-2):25–43, 2009.

[9] 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. *Annales des Télécommunications*, 64(1-2):5–24, 2009.

[10] Denis Caromel, Ludovic Henrio, and Bernard P. Serpette. Asynchronous sequential processes. *Inf. Comput.*, 207(4):459–495, 2009.

[11] Françoise Baude, Virginie Legrand, Ludovic Henrio, Paul Naoumenko, Heiko Pfeffer, Louay Bassbouss, and David Linner. Mixing Workflows and Components to Support Evolving Services. *International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS)*, 1(4):60–84, 2010.

[12] Ludovic Henrio, Florian Kammüller, and Bianca Lutz. ASPfun : A typed functional active object calculus. *Science of Computer Programming*, In Press, Corrected Proof:–, 2011.

Conferences and Workshops:

[13] Isabelle Attali, Denis Caromel, Carine Courbis, Ludovic Henrio, and Henrik Nilsson. Smart Tools for Java Cards. In Josep Domingo-Ferrer, David Chan, and Anthony Watson, editors, *Smart Card Research and Advanced Applications*. Kluwer Academic Publishers, September 2000. Proceedings of the IFIP Fourth Working Conference on Smart Card Research and Advanced Applications (CARDIS 2000), HP Labs, Bristol, UK.

[14] Denis Caromel, Ludovic Henrio, and Bernard Serpette. Context inference for static analysis of java card object sharing. In *Proceedings e-Smart 2001*. Springer-Verlag, 2001.

[15] Ludovic Henrio and Bernard Paul Serpette. A parameterized polyvariant Byte-Code verifier. In *Actes des journées JFLA*, Chamrousse, France, January 2003.

[16] Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*, pages 123–134. ACM Press, 2004.

[17] Ludovic Henrio, Bernard Paul Serpette, and Szabolcs Szentes. Algorithmes et complexités de la réduction statique minimale. In *Actes des journées JFLA*, Sainte-Marie-de-Ré, France, January 2004.

[18] Isabelle Attali, Denis Caromel, Ludovic Henrio, and Felipe Luna Del Aguila. Secured information flow for asynchronous sequential processes. In *3rd International Workshop on Security Issues in Concurrency (SecCo'05)*, Electronic Notes in Theoretical Computer Science, San Francisco, USA, August 2005. Elsevier.

[19] Laurent Baduel, Françoise Baude, Denis Caromel, Ludovic Henrio, Fabrice Huet, Stéphane Lanteri, and Matthieu Morel. Grid components techniques: Composing, gathering, and scattering. In *Coupled Problems 2005, Computational Methods for Coupled Problems in Science and Engineering, an ECCOMAS Thematic Conference*, Santorini, Greece, may 2005.

[20] Tomás Barros, Ludovic Henrio, and Eric Madelaine. Behavioural models for hierarchical components. In *Proceedings of SPIN'05*. Spinger Verlag, 2005.

[21] Tomás Barros, Ludovic Henrio, and Eric Madelaine. Verification of distributed hierarchical components. In *International Workshop on Formal Aspects of Component Software (FACS'05)*, Macao, October 2005. Electronic Notes in Theoretical Computer Science (ENTCS).

[22] Françoise Baude, Denis Caromel, Christian Delbé, and Ludovic Henrio. A Hybrid Message Logging-CIC Protocol for Constrained Checkpointability. In *Proc. of the 11th International Euro-Par Conference*, volume 3648 of *LNCS*. Springer-Verlag, 2005.

[23] Alessandro Basso, Alexander Bolotov, Artie Basukoski, Vladimir Getov, Ludovic Henrio, and Mariusz Urbanski. Specification and verification of reconfiguration protocols in grid component systems. In *Proceedings of the 3rd IEEE Conference On Intelligent Systems IS-2006*. IEEE Computer Society, 2006. long version published as a CoreGRID Technical Report, TR-0042.

[24] Sébastien Bezinne, Virginie Galtier, Stéphane Vialle, Françoise Baude, Mireille Bossy, Viet-Dung Doan, and Ludovic Henrio. A fault tolerant and multi-paradigm grid architecture for time constrained problems. application to financial option pricing. In *2nd IEEE International Conference on e-Science and Grid Computing*. IEEE, December 2006.

[25] Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Towards real case component model-checking. In *5th Fractal Workshop*, Nantes, France, July 2006.

[26] Denis Caromel and Ludovic Henrio. Asynchonous distributed components: Concurrency and determinacy. In *Proceedings of the IFIP International Conference on Theoretical Computer Science 2006 (IFIP TCS'06)*, Santiago, Chile,, August 2006. Springer Science. 19th IFIP World Computer Congress.

[27] Jeyarajan Thiyagalingam, Nikos Parlavantzas, Stavros Isaiadis, Ludovic Henrio, Denis Caromel, and Vladimir Getov. Proposal for a lightweight generic grid platform architecture. In *Proceedings of CompFrame 2006, Component and Framework Technology in High-Performance and Scientific Computing*, Paris, France, June 2006. IEEE.

[28] Françoise Baude, Denis Caromel, Ludovic Henrio, and Matthieu Morel. Collective interfaces for distributed components. In *CCGrid 2007: IEEE International Symposium on Cluster Computing and the Grid*, May 2007.

[29] Françoise Baude, Denis Caromel, Christian Delbé, and Ludovic Henrio. Promised messages: Recovering from inconsistent global states. In *ACM SIGOPS conference Principles and Practice of Parallel Programming (PPoPP). Poster*, 2007.

[30] Françoise Baude, Ludovic Henrio, and Paul Naoumenko. A Component Platform for Experimenting with Autonomic Composition. In *First International Conference on Autonomic Computing and Communication Systems (Autonomics 2007). Invited Paper*. ACM Digital Library, Oct 2007.

[31] Denis Caromel, Guillaume Chazarain, and Ludovic Henrio. Garbage collecting the grid: a complete dgc for activities. In *Proceedings of the 8th ACM/IFIP/USENIX International Middleware Conference*, Newport Beach, CA, November 2007.

[32] Ludovic Henrio and Florian Kammüller. A mechanized model of the theory of objects. In *9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, LNCS. Springer, June 2007.

[33] Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Transparent first-class futures and distributed component. In *International Workshop on Formal Aspects of Component Software (FACS'08)*, Malaga, Sept 2008. Electronic Notes in Theoretical Computer Science (ENTCS).

[34] Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Unifying architectural and behavioural specifications of distributed components. In *International Workshop on Formal Aspects of Component Software (FACS'08)*, Malaga, Sept 2008. Electronic Notes in Theoretical Computer Science (ENTCS).

[35] Denis Caromel, Ludovic Henrio, and Mario Leyton. Type safe algorithmic skeletons. In *Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and network-based Processing*, Toulouse, France, February 2008.

[36] Denis Caromel, Ludovic Henrio, and Eric Madelaine. Active objects and distributed components: Theory and implementation. In Frank de Boer and Marcello Bonsangue, editors, *FMCO 2007*, number 5382 in LNCS, pages 179–199, Berlin Heidelberg, 2008. Springer-Verlag.

[37] Ludovic Henrio and Marcela Rivera. Stopping safely hierarchical distributed components: application to gcm. In *CBHPC '08: Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance*, pages 1–11, New York, NY, USA, 2008. ACM.

[38] Boutheina Bannour, Ludovic Henrio, and Marcela Rivera. A reconfiguration framework for distributed components. In *SINTER Workshop Software Integration and Evolution @ Runtime*. ACM, 2009.

[39] Françoise Baude, Ludovic Henrio, and Paul Naoumenko. Structural reconfiguration: An autonomic strategy for gcm components. In *International Conference on Autonomic and Autonomous Systems*, pages 123–128, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[40] Ludovic Henrio and Florian Kammüller. Functional active objects: Typing and formalisation. In *Proceedings of the International Workshop on the Foundations of Coordination Languages and Software Architecture (FOCLASA)*. Elsevier, 2009.

[41] Ludovic Henrio, Florian Kammüller, and Marcela Rivera. An asynchronous distributed component model and its semantics. In Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelaine, editors, *Formal Methods for Components and Objects, 7th International Symposium, FMCO 2008, Sophia Antipolis, France, October 21-23, 2008, Revised Lectures*, volume 5751 of *Lecture Notes in Computer Science*, pages 159–179. Springer, 2009.

[42] Rabéa Ameur Boulifa, Ludovic Henrio, and Eric Madelaine. Behavioural models for group communications. In *WCSI-10: International Workshop on Component and Service Interoperability*, 2010.

[43] Ludovic Henrio, Florian Kammüller, and Muhammad Uzair Khan. A framework for reasoning on component composition. In *FMCO 2009*, Lecture Notes in Computer Science. Springer, 2010.

[44] Ludovic Henrio and Muhammad Uzair Khan. Asynchronous components with futures: Semantics and proofs in isabelle/hol. In *Proceedings of the Seventh International Workshop, FESCA 2010*. ENTCS, 2010.

[45] Ludovic Henrio, Muhammad Uzair Khan, Nadia Ranaldo, and Eugenio Zimeo. First class futures: Specification and implementation of update strategies. In *Post-Proceedings Selected Papers From The Coregrid Workshop On Grids, Clouds and P2P Computing August 31, 2010*, August 2010.

[46] Mario Leyton, Ludovic Henrio, and José M. Piquer. Exceptions for algorithmic skeletons. In *16th Int. European Conference on Parallel and Distributed Computing (Euro-Par 2010)*, 2010.

[47] Rabéa Ameur Boulifa, Raluca Halalai, Ludovic Henrio, and Eric Madelaine. Verifying safety of fault-tolerant distributed components. In *International Symposium on Formal Aspects of Component Software (FACS 2011)*, Lecture Notes in Computer Science, Oslo, Sept 2011. Springer.

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

Thesis:

[49] Ludovic Henrio. *Calcul d'Objets Asynchrones : Confluence et Déterminisme*. PhD thesis, Université de Nice Sophia-Antipolis, 2003. `http://www-sop.inria.fr/oasis/Ludovic.Henrio/these`.

Reports, Deliverables ...

[50] Ludovic Henrio. Analyses de partage pour applications javacard. Rapport de DEA, septembre 2000.

[51] Denis Caromel, Christian Delbé, Ludovic Henrio, and Romain Quilici. Brevet "dispositif et procédé asynchrones et automatiques de transmission de résultats entre objets communicants", Nov 2003. le 26 11 2003, No FR 03 138 76.

[52] Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous sequential processes. Research Report, INRIA Sophia Antipolis, 2003. RR-4753.

[53] Ludovic Henrio, Bernard Serpette, and Szabolcs Szentes. Implementation and complexity of the lowest static reduction. Research Report, INRIA Sophia Antipolis, 2003. RR-5034.

[54] Françoise Baude, Denis Caromel, Christian Delbé, and Ludovic Henrio. A fault tolerance protocol for ASP calculus : Design and proof. Research Report, INRIA Sophia Antipolis, June 2004. RR-5246.

[55] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for hierarchical components. Technical Report RR-5591, INRIA, June 2005.

[56] F. Baude, D. Caromel, L. Henrio, and M. Morel. Collective interfaces for a grid component model, proposed extensions to the fractal component model. Technical report, Internal technical report, CRE France Telecom R&D, Nov 2005.

[57] Rosa M. Badia, Olav Beckmann, Sofia Panagiotidi, Denis Caromel, Ludovic Henrio, Marian Bubak, Maciek Malawski, Vladimir Getov, Stavros Isaiadis, Jeyarajan Thiyagalingam, and Vladimir Lazarov. Lightweight grid platform: Design methodology. Technical report, Institute on Grid Systems, Tools and Environments, Jan 2006. CoreGRID Technical Report, TR-0020.

[58] Alessandro Basso, Alexander Bolotov, Artie Basukoski, Vladimir Getov, Ludovic Henrio, and Mariusz Urbanski. Specification and verification of reconfiguration protocols in grid component systems. Technical report, Institute on Programming Model (WP3), May 2006. CoreGRID Technical Report, TR-0042.

[59] F. Baude, V. D. Doan, L. Henrio, P. Naoumenko, and partners VTT, TUB, UBasel, CreateNet, SUN, Nokia of the BIONETS consortium. Specification of service life-cycle. Technical Report D.3.2.1, BIONETS IP Project Delivrable from the Requirement and Analysis workpackage (3.2), Dec. 2006.

[60] F. Baude, L. Henrio, and partners VTT, TUB, UNIHH of the BIONETS consortium. Service architecture requirement specification. Technical Report D.3.1.1, BIONETS IP Project Delivrable from the Requirement and Analysis workpackage (3.1), July 2006.

[61] Françoise Baude, Denis Caromel, Ludovic Henrio, Matthieu Morel, and Paul Naoumenko. Fractalising fractal controller for a componentisation of the non-functional aspects. 5th Fractal Workshop in conjunction with ECOOP'20 – poster, July 2006.

[62] D. Caromel, C. Delbé, and L. Henrio. Promised consistency for rollback recovery. Technical report, INRIA, Sophia Antipolis, 2006. Technical report n RR-5902.

[63] CoreGRID, Programming Model Institute. Basic features of the grid component model (assessed). Technical report, CoreGRID, Programming Model Virtual Institute, 2006. Deliverable D.PM.04, `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf`.

[64] CoreGRID Programming Model Virtual Institute. Programming models for the single gcm component: a survey, Sep. 2006. Deliverable D.PM.06.

[65] Ludovic Henrio and Florian Kammüller. A formalization of the theory of objects in Isabelle/HOL. Rapport de recherche, INRIA, 2006. RR-6079.

[66] OASIS team and other partners in the CoreGRID Programming Model Virtual Institute. Proposals for a grid component model. Technical Report D.PM.02, CoreGRID, Programming Model Virtual Institute, Feb 2006. Responsible for the delivrable.

[67] OASIS team and other partners in the CoreGRID Programming Model Virtual Institute. Survey of advanced component programming models. Technical report, CoreGRID, Programming Model Virtual Institute, Oct. 2006. Deliverable D.PM.05, CoreGRID, Programming Model Institute.

[68] Françoise Baude, Ludovic Henrio and partners TUB, UBASEL, CN, SUN, NOKIA, VTT, TI of the BIONETS consortium. Specification of service evolution. Technical report, BIONETS IP Project Delivrable from the Autonomic Service Life-Cycle And Service Ecosystems WP (3.2), Aug 2007. http://www.bionets.eu/docs/BIONETS_D3_2_2.pdf.

[69] Ludovic Henrio, Florian Kammüller, and Henry Sudhof. Aspfun: A functional and distributed object calculus semantics, type-system, and formalization. Research Report 6353, INRIA, 11 2007.

[70] OASIS team and other partners in the CoreGRID Programming Model Virtual Institute. Innovative features of gcm (with sample case studies): a technical survey. Technical report, CoreGRID, Programming Model Virtual Institute, Sep. 2007. Deliverable D.PM.07.

[71] Marco Aldinucci, Sonia Campa, Massimo Coppola, Marco Danelutto, G. Zoppi, Alessandro Basso, Alexander Bolotov, Francoise Baude, Hinde Bouziane, Denis Caromel, Ludovic Henrio, Christian Pérez, Jose Cunha, Classen Michael, Philipp Classen, Christian Lengauer, J. Cohen, S. Mc Gough, Natalia Currle-Linde, Patrizio Dazzi, Nicola Tonellotto, Jan Dünnwebber, Sergei Gorlatch, Peter Kilpatrick, Nadia Ranaldo, and Eugenio Zimeo. Proceedings of the programming model institute technical meeting 2008. Technical Report TR-0138, Institute of Programming Model, CoreGRID - Network of Excellence, May 2008.

[72] Françoise Baude, Ludovic Henrio, Paul Naoumenko, and Heiko Pfeffer. Graph-Based Service Individual specification: Creation and Representation. Technical report, BIONETS IP Project Delivrable from the Autonomic Service Life-Cycle And Service Ecosystems WP (3.2), Jan, revised June 2008. http://www.bionets.eu/docs/BIONETS_D3_2_3.pdf.

[73] Ludovic Henrio and Marcela Rivera. An algorithm for safely stopping a component system. Research Report RR-6444, INRIA, 2008.

[74] J. Lahti, Ludovic Henrio, K. Ville, F. Laura, Daniele Miorandi, David Linner, Heiko Pfeffer, and Françoise Baude. Advanced service life-cycle and integration. Technical Report D.3.2.4, BIONETS IP Project Delivrable from the Autonomic Service Life-Cycle And Service Ecosystems WP (3.2), June 2008.

[75] David Linner, Heiko Pfeffer, Stephan Steglich, Françoise Baude, Ludovic Henrio, and Paul Naoumenko. Service probes implementation and evaluation. Technical Report D.3.4.1, BIONETS IP Project Delivrable from the Service Probes WP (3.4), June 2008.

[76] Françoise Baude, Ludovic Henrio, Paul Naoumenko, Daniele Miorandi, and Janne Lathi and. Evaluating the fitness of service compositions. Technical Report D.3.2.5, BIONETS IP Project Delivrable from the Autonomic Service Life-Cycle And Service Ecosystems WP (3.2), September 2009.

[77] Muhammad Uzair Khan and Ludovic Henrio. First class futures: A study of update strategies. Technical report, INRIA a CCSD electronic archive server based on P.A.O.L [http://hal.inria.fr/oai/oai.php] (France), 2009. RR-7113.

[78] Heiko Pfeffer, Louay Bassbouss, Paul Naoumenko, Daniele Miorandi, David Lowe, Mihaela Ion, and Lahti Janne. Bio inspired service creation and evolution. Technical Report D.3.2.6, BIONETS IP Project Delivrable from the Autonomic Service Life-Cycle And Service Ecosystems WP (3.2), December 2009.

[79] Francesco Bongiovanni and Ludovic Henrio. Mechanical Support for Efficient Dissemination on the CAN Overlay Network. Research Report RR-7599, INRIA, April 2011. Also accepted at CFSE 2011.

Chapter 6

References

[ABF+08] Brian E. Aydemir, Aaron Bohannon, Nate Foster, Benjamin Pierce, Jeff Vaughan, Dimitris Vytiniotis, Geoff Washburn, Stephanie Weirich, Steve Zdancewic, Matthew Fairbairn, and Peter Sewell. The poplmark challenge. Web-site, 2008.

[ACP+08] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 3–15, New York, NY, USA, 2008. ACM.

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[AMST97] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[AT04] Gul Agha and Prasanna Thati. An algebraic theory of actors and its application to a simple object-based language. In *Essays in Memory of Ole-Johan Dahl*, volume 2635, pages 26–57, 2004.

[BBB+07] Michael Beisiegel, Henning Blohm, Dave Booz, Mike Edwards, and Oisin Hurley. SCA service component architecture, assembly model specification. Technical report, March 2007. `www.osoa.org/display/Main/Service+Component+Architecture+Specifications`.

[BBF+07] B. Berthomieu, J.P. Bodeveix, M. Filali, H. Garavel, F. La ng, F. Peres, R. Saad, J. Stoecker, and F. Vernadat. The syntax and semantics of Fiacre. In *Rapport LAAS 07264 Rapport de Contrat Projet OpenEmbeDD*, Mai 2007.

[BCJ07] Frank S. De Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.

[BCL+04] Eric Bruneton, Thierry Coupaye, M. Leclercp, V. Quema, and Jean Bernard Stefani. An open component model and its support in java. In *7th Int. Symp. on Component-Based Software Engineering (CBSE-7)*, LNCS 3054, may 2004.

[BCS84] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Proceedings of the Fourth International Symposium on Reliability in Distributed Software and Databases*, pages 207–215. Citeseer, 1984.

[BCS04] Eric Bruneton, Thierry Coupaye, and Jean Bernard Stefani. The Fractal Component Model. Technical report, ObjectWeb Consortium, February 2004. `http://fractal.objectweb.org/specification/index.html`.

[BG07] Rana Bakhshi and Dilian Gurov. Verification of peer-to-peer algorithms: A case study. *Electronic Notes in Theoretical Computer Science*, 181:35–47, June 2007.

[BHP06] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, pages 40–48. IEEE, 2006.

[BNOG05] J. Borgström, U. Nestmann, L. Onana, and D. Gurov. Verifying a structured peer-to-peer overlay network: The static case. In *Global Computing*, pages 250–265. Springer, 2005.

[Bou04] Rabéa Boulifa. *Génération de modèles comportementaux des applications réparties*. PhD thesis, University of Nice - Sophia Antipolis – UFR Sciences, December 2004.

[BP07] J. Bengtson and J. Parrow. Formalising the pi-Calculus using Nominal Logic. In *Proc. of the 10th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 4423 of *LNCS*, pages 63–77, 2007.

[CBBM+96] Charron-Bost, Bernadette, Mattern, Friedemann, and Gerard Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9:173–191, 1996. 10.1007/s004460050018.

[CDdCL06] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12(1):69–77, 2006.

[CF05] CCA-Forum. The Common Component Architecture (CCA) Forum home page, 2005. http://www.cca-forum.org/.

[CGS+05] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538. ACM, 2005.

[CLM07] A. Ciaffaglione, L. Liquori, and M. Miculan. Reasoning about object-based calculi in (co)inductive type theory and the theory of contexts. *JAR, Journal of Automated Reasoning*, 39:1–47, 2007.

[Col91] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[CS02] Xuejun Chen and Martin Simons. A component framework for dynamic reconfiguration of distributed systems. In *CD '02: Proceedings of the IFIP/ACM Working Conference on Component Deployment*, pages 82–96, London, UK, 2002. Springer-Verlag.

[DCMM06] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Ambient-oriented programming in ambienttalk. In *Proceedings of 20th European Conference on Object-oriented Programming (ECOOP)*. Springer, 2006.

[Del07] Christian Delbé. *Tolérance aux pannes pour objets actifs asynchrones - protocole, modèle et expérimentations*. PhD thesis, Université de Nice-Sophia Antipolis, January 2007.

[DL06] Pierre-Charles David and Thomas Ledoux. Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceeding of Fractal CBSE Workshop, ECOOP'06*, Nantes, France, 2006.

[DLLC09] Pierre-Charles David, Thomas Ledoux, Marc Léger, and Thierry Coupaye. Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures. *Annals of Telecommunications*, 64:45–63, 2009. 10.1007/s12243-008-0073-y.

[EAWJ02] E N Mootaz Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[FF95] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. pages 209–220, 1995.

[FF99] Cormac Flanagan and Matthias Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.

[GLM02] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002.

[GLMS11] H. Garavel, F. Lang, R. Mateescu, and W. Serve. Cadp 2010: A toolbox for the construction and analysis of distributed processes. In *TACAS'11*, volume 6605 of *LNCS*, Saarbrücken, Germany, 2011. Springer, Heidelberg.

[GSAA04] A. Gupta, O.D. Sahin, D. Agrawal, and A.E. Abbadi. Meghdoot: content-based publish/subscribe over P2P networks. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 254–273. Springer-Verlag New York, Inc., 2004.

[Hal85] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[HB07] Petr Hnětynka and Tomáš Bureš. Advanced features of hierarchical component models. In Alice Kelemenova, Dusan Kolar, Alexander Meduna, and Jaroslav Zendulka, editors, *Information Systems and Formal Methods*, pages 3–10, Opava, Czech Republic, 2007. Selesian University in Opava.

[Hil04] I. Hillman, J.; Warren. An open framework for dynamic reconfiguration. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 594–603, 23-28 May 2004.

[HL06] M. Haustein and K.P. Lohr. Jac: declarative java concurrency. *Concurrency and Computation: Practice and Experience*, 18(5):519–546, 2006.

[HMS01] Furio Honsell, Marino Miculan, and Ivan Scagnetto. pi-calculus in (co)inductive-type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.

[JO06] Einar Broch Johnsen and Olaf Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software & Systems Modeling*, 6(1):39–58, August 2006.

[JOY06] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: a type-safe object-oriented model for distributed concurrent systems. *Theoratical Computer Science*, 365(1):23–66, 2006.

[Koz85] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 40, 1985.

[Ley08] Mario Leyton. *Advanced Features for Algorithmic Skeleton Programming*. PhD thesis, Université de Nice - Sophia Antipolis – UFR Sciences, October 2008.

[LMW11] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. Towards Verification of the Pastry Protocol Using TLA$^+$. In *FMOODS/FORTE*, pages 244–258, 2011.

[LP04] H. Liu and M. Parashar. A component based programming framework for autonomic applications. In *First International Conference on Autonomic Computing (ICAC'04)*, 2004.

[LS96] R. Greg Lavender and Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. In *Pattern languages of program design 2*, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[Ls03] L. Lamport and Safari Tech Books Online (Online service). *Specifying systems: The TLA+ language and tools for hardware and software engineers*, volume 14. Addison-Wesley, 2003.

[LS11] Li Lu and Michael Scott. Toward a formal semantic framework for deterministic parallel programming. In *Proceedings of 25th Intl. Symp. on Distributed Computing (DISC)*, Rome, Italy, 2011. ACM.

[Mad11] Eric Madelaine. *Specification, Model Generation, and Verification of Distributed Applications*. PhD thesis, Université de Nice Sophia-Antipolis, September 2011. Habilitation à diriger des Recherches.

[MB05] V. Mencl and T. Bures. Microcomponent-based component controllers: A foundation for component aspects. In *APSEC*. IEEE Computer Society, Dec. 2005.

[MG05a] A. Mukhija and M. Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *Systems Aspects in Organic and Pervasive Computing - ARCS 2005*, pages 124–138. Springer Berlin / Heidelberg, 2005.

[MG05b] Arun Mukhija and Martin Glinz. The casa approach to autonomic applications. In *Proceedings of the 5th IEEE Workshop on Applications and Services in Wireless Networks, ASWN 2005*, pages 173–182. IEEE Computer Society, 2005.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989. ISBN 0-13-114984-9.

[MT08] R. Mateescu and D. Thivolle. A model checking language for concurrent value-passing systems. In K. Sere J. Cuellar, T. S. E. Maibaum, editor, *FM'08*, volume 5014 of *LNCS*. Springer, Heidelberg, 2008.

[NSS06] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, November 2006.

[NSSSS07] Joachim Niehren, David Sabel, Manfred Schmidt-Schauß, and Jan Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. In *23rd Conference on Mathematical Foundations of Programming Semantics*, ENTCS, New Orleans, April 2007. Accepted.

[Obj06] Object Management Group, Inc. (OMG). *CORBA Component Model Specification*, omg headquarters edition, April 2006. `http://www.omg.org/cgi-bin/apps/doc?formal/06-04-01.pdf`.

[Pit03] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[PP06] P. Parizek and F. Plasil. Model checking of software components: Combining java pathfinder and behavior protocol model checker. In *Proceedings of 30th IEEE/NASA Software Engineering Workshop (SEW-30)*. IEEE Computer Press, april 2006.

[PR06] P. Poizat and J.C. Royer. A Formal Architectural Description Language based on Transition Systems and Modal Logic. *Journal of Universal Computer Science*, 12(12), 2006.

[PRS06] P. Poizat, J.C. Royer, and G. Salaun. Bounded Analysis and Decomposition for Behavioural Descriptions of Components. In *FMOODS, LNCS 4037*, 2006.

[PV02] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), nov 2002.

[RBS11] C. Ruz, F. Baude, and B. Sauvan. Flexible Adaptation Loop for Component-based SOA applications. In *7th International Conference on Autonomic and Autonomous Systems (ICAS 2011)*. IEEE Explorer, 2011. Best paper awarded.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-To-Peer Systems. In *Int. Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[RFH+01] 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)*, pages 161–172. ACM, 2001.

[RH03] C. Roeckl and D. Hirschkoff. A fully adequate shallow embedding of the π-calculus in isabelle/hol with mechanized syntax analysis. *Journal of Functional Programming*, 13:415–451, 2003.

[RHKS01] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. *Networked Group Communication*, 2001.

[RP03] Andreas Rasche and Andreas Polze. Configuration and dynamic reconfiguration of component-based applications with microsoft .net. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, page 164, Washington, DC, USA, 2003. IEEE Computer Society.

[Ruz11] Cristian Ruz. *Autonomic Monitoring and Management of Component-Based Services*. PhD thesis, University of Nice-Sophia Antipolis, Jun 2011.

[SMK+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)*, pages 149–160, New York, NY, USA, 2001. ACM.

[SPC05] L. Seinturier, N. Pessemier, and T. Coupaye. AOKell: an Aspect-Oriented Implementation of the Fractal Specifications, 2005. `http://www.lifl.fr/~seinturi/aokell/javadoc/overview.html`.

[SPDC06] L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A component model engineered with components and aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, Lecture Notes in Computer Science. Springer, June 2006.

[SPH10] Jan Schafer and Arnd Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. *ECOOP 2010–Object-Oriented Programming*, pages 275–299, 2010.

[TBN+08] E. Tejedor, R. Badia, P. Naoumenko, M. Rivera, and C. Dalmasso. Orchestrating a safe functional suspension of gcm components. In *CoreGRID Integration Workshop 2008, Integrated Research in Grid Computing Hersonissos-Crete, Greece*, April 2008.

[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*, pages 275–292. American Mathematical Society, 1994.

[Uea06] C. Urban and et al. Nominal methods group, 2006. Project funded by the German Research Foundation (DFG) within the Emmy-Noether Programme.

[WJH05] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for java. *SIGPLAN Not.*, 40(10), 2005.

[Zav09] Pamela Zave. Lightweight verification of network protocols: The case of chord. Unpublished, `http://www2.research.att.com/~pamela/chord.pdf`, 2009.

Appendix A

Detailed CV

Education and Experience

since 2005 — Associate Scientist at CNRS – CR1 since 2009
OASIS Team - joint project INRIA/CNRS/I3S/Univ. of Nice Sophia Antipolis

2004 - 05
1 year — Research fellow at the University of Westminster:
Teaching at Harrow School of Computer Science
Research in the "Distributed and High Performance Computing" department.
Keywords: components, Grid, non-functional aspects, reconfiguration.

2003 - 04
1 year — Temporary teaching and Research assistant:
University of Nice Sophia-Antipolis at ESSI (computer science engineer school)
Research at INRIA Sophia-Antipolis - OASIS team (INRIA/CNRS/I3S/Univ. of Nice Sophia Antipolis)

2001 - 03
3 years — Ph.D in Computer Science: University of Nice Sophia-Antipolis
defended the 28th of November 2003 in Sophia Antipolis
Subject: *Asynchronous Object Calculus: Confluence and Determinacy*

Advisor:	Denis Caromel	UNSA, IUF
Co-advisor:	Bernard Paul Serpette	INRIA Sophia Antipolis
Reviewer:	Luca Cardelli	Microsoft research, Cambridge
	Ugo Montanari	Universita di Pisa
	Elie Najm	ENST, Paris
Jury members:	Gérard Boudol (president)	INRIA Sophia Antipolis
	Gilles Kahn	INRIA

Keywords: Parallelism, concurrency, object calculus, confluence, distribution.

2000
5 months — Research internship at INRIA Sophia Antipolis - OASIS project
Static analysis of Java Card Object Sharing.
keywords: Security, Java Card, static analysis, typing.

1999 - 00 — Master Degree in Computer Science: "*Programming* : semantics proofs and languages". University of Paris 7.

1999
4 months — Research internship at INRIA Sophia Antipolis - OASIS team
Interactive testing of Java Card Applications.
keywords: static analysis, Java Card, tests.

1996 - 99 — *Polytechnique School*

Students and Teaching

Advised PhD Students

- **Marcela Rivera**: *"Reconfiguration and Life-cycle of Distributed Components : Asynchrony, Coherence and Verification"* (Dec 2006 - Dec 2011).
  PhD advisors: Denis Caromel and Ludovic Henrio
  Scientific advisor: Ludovic Henrio
  Summary: For component programming, but even more specifically in distributed and Grid environments, components need to be highly adaptative. A great part of adaptativeness relies on dynamic reconfiguration of component systems. We introduce a new approach for reconfiguring distributed components with the main objective to facilitate the reconfiguration process and ensure the consistency and coherence of the system. First, before executing a reconfiguration it is necessary that the components is a coherent and quiescent state. This is done to avoid inconsistency in the reconfiguration process. To achieve this, we design an algorithm for stopping a component in a safe manner and reach this quiescent state. This was realized by implementing a tagging and interception mechanism that adds informations to the requests and manipulates their flow in order to decide which of them must be served before stopping the component. Additionally, for triggering the reconfiguration tasks, we extended the FScript language to give it the capability of executing distributed reconfiguration actions, by delegating some actions to specific components. To achieve this objective, we defined an additional controller inside the management part of the components. We tested our implementation over two GCM/ProActive based applications: the CoCoME example and the TurnTable example.

- **Muhammad-Uzair Khan**: *"A Study of First Class Futures: Specification, Formalisation, and Mechanised Proofs"* (Oct 2007 - Feb 2011)
  PhD advisors: Denis Caromel and Ludovic Henrio
  Scientific advisor: Ludovic Henrio
  Summary: Futures enable an efficient and easy to use programming paradigm for distributed applications. A future is a placeholder for result of concurrent execution. Futures can be "first class objects"; first class futures may be safely transmitted between the communicating processes. Consequently, futures spread everywhere. When the result of a concurrent execution is available, it is communicated to all processes which received the future. In this thesis, we study the mechanisms for transmitting the results of first class futures; the 'future update strategies'.
  We provide a detailed semi-formal specification of three main future update strategies adapted from ASP-Calculus ; we then use this specification for a real implementation in a distributed programming library. We study the efficiency of the three update strategies through experiments. Ensuring correctness of distributed protocols, like future update strategies is a challenging task. To show that our specification is correct, we formalise it together with a component model. Components abstract away the program structure and the details of the business logic; this paradigm thus facilitates reasoning on the protocol. We formalise in Isabelle/HOL, a component model comprising notions of hierarchical components, asynchronous communications, and futures. We present the basic constructs and corresponding lemmas related to structure of components. We present formal operational semantics of our components in presence of a future update strategy; proofs showing correctness of our future update strategy are presented. Our work can be considered as a formalisation of ProActive/GCM and shows the correctness of the middleware implementation.

- **Paul Naoumenko**: *"Designing non-functional aspects with components"* (Oct 2006 - Jul 2010)
  PhD advisors: Françoise Baude and Ludovic Henrio
  Scientific advisors: Françoise Baude and Ludovic Henrio
  Summary: In this thesis we considered programming models for large-scale and distributed applications that are deployed in dynamic ever-changing environments, like the Grid. To maintain their function with minimal involvement of human operators, those applications must provide self-adaptive capabilities. We ground our research on the autonomic computing paradigm, which proposes to design applications as compositions of autonomic elements. Those are software entities exposing two parts: a business part, and a management part, with managers in charge of supervising the business part by reacting to environmental changes.

Managers have the possibility to implement complex management strategies: additionally to the supervision of the business part, they can contact managers from other autonomic elements involved in the application, and collaborate with them in order to elaborate adequate reactions. Strategies of managers can be dynamically updated. We propose to design distributed autonomic applications using a component-oriented model: the GCM (Grid Component Model). GCM components are distributed by essence and the model features as a part of its specification separation of concerns (GCM components have a business part and a management part), hierarchical structure, and dynamic reconfiguration. Our contribution is twofold.First, we extend the management part of GCM components, giving the possibility to include managers that correspond to the vision of autonomic computing. Thanks to newly introduced architectural elements, the managers are able to supervise the business part of GCM components. They can also contact managers of other components and collaborate with them. A GCM component with self-adaptive capabilities should be easy to produce: we suggest a development process to design and implement the management part separately from the business part, and then integrate both parts inside one unified software entity. We modify the Architecture Description Language to statically describe GCM component assemblies according to the new development process. We included the previously presented extensions in the reference implementation of GCM.

- **Alessandro Basso**: *"Integrating formal reasoning into a component-based approach to reconfigurable distributed systems"*. Univ. of Westminster. (Feb 2006 - Mar 2010)
  PhD advisors: Alexander Bolotov, Vladimir Getov and Ludovic Henrio
  Scientific advisors: Alexander Bolotov
  Summary: Grid systems were born out of necessity, and had to grow quickly to meet requirements which evolved over time, becoming today's complex systems. Even the simplest distributed system nowadays is expected to have some basic functionalities, such as resources and execution management, security and optimisation features, data control, etc. The complexity of Grid applications is also accentuated by their distributed nature, making them some of the most elaborate systems to date. It is often too easy that these intricate systems happen to fall in some kind of failure, it being a software bug, or plain simple human error; and if such a failure occurs, it is not always the case that the system can recover from it, possibly meaning hours of wasted computational power.
  The difficulty of Grid systems to deal with unforeseen and unexpected circumstances resulting from dynamic reconfiguration is related to the fact that Grid applications are large, distributed and prone to resource failures. This research has produced a methodology for the solution of this problem by analysing the structure of distributed systems and their reliance on the environment which they sit upon. It is concluded that the way that Grid applications interact with the infrastructure is not sufficiently addressed and a novel approach is developed in which formal verification methods are integrated with distributed applications development and deployment in a way that includes the environment. This approach allows for reconfiguration scenarios in distributed applications to proceed in a safe and controlled way, as demonstrated by the development of a prototype application.

Additionally to the PhD students mentioned above, I have been unofficially but strongly involved in the scientific advisement of the following PhD students, all those collaborations resulted in published papers:

- Christian Delbé (2003-2007): "Tolérance aux pannes pour objets actifs asynchrones - protocole, modèle et expérimentations"
- Mario Leyton (2005-2008): "Advanced Features for Algorithmic Skeleton Programming"
- Francesco Bongiovanni (2008-2012): Design, Formalisation and Implementation of Overlay Networks; Application to RDF Data Storage".

Internship Students

- Master 2: Paul Naoumenko : "A component-oriented approach for adaptive and autonomic computing: application to situated autonomic communications" (2006)
- Master 2: Muhammad Uzair Khan : "A Fault-tolerance Mechanism for Future Updates" (2007)
- Master 2 + Enseirb: Boutheina Bannour: "Langage de Reconfiguration pour Composants Distribués" (2008)
- Master 1: Sona Djohou: "Outils pour la preuve formelle de propriétés ASP" (2008)

PhD Committees

- Yann Hodique - Univ. des Sciences et Technologies de Lille (jury member): "Sûreté et optimisation par les systèmes de types en contexte ouvert et contraint" (2007).

Teaching

- Java Card Programming (48h - Master 2),
- Java Card Security (8h - Master 2),
- System programming (54h - ESSI 2nd year),
- C language (42h - ESSI 3rd year)
- Introduction to Programming – C++ (66h - Univ. of Westminster - first year)
- Object Oriented Software Development – Java (44h - Univ. of Westminster - first year)
- Semantics of Distributed and Embedded Systems (21h, Master 1 - 2009-2011)
- Distributed Systems: an algorithmic approach (20h, Master 2 - 2009-2011)

Contracts and Collaborations

I have been significantly involved and took responsibilities in the following projects:

NoE CoreGrid

Type: European Network of excellence FP6
Title: The European Research Network of Excellence on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies
Dates: 2005-2009
Personal responsibility: Coordination of deliverables, local responsible for a work-package (programming models)
Partners: ERCIM (France). CETIC (Belgium), IPP-BAS (Bulgaria), CNR-ISTI (Italy), CNRS (France), TUD (The Netherlands), EPFL (Switzerland), FhG (Germany), FZJ (Germany), USTUTT (Germany), ICS-FORTH (Greece), INFN (Italy), INRIA (France), KTH (Sweden), MU (Czech R.), PSNC (Poland), STFC (UK), SICS (Sweden), SZTAKI (Hungary), QUB (UK), WWU Muenster (Germany), UNICAL (Italy), UWC (UK), UCHILE (Chili), UCO (Portugal), UCY (Cyprus), Univ. Dortmund (Germany), UCL (Belgium), Univ. of Manchester (UK), UNCL (UK), Univ. Passau (Germany), Univ. Pisa(Italy), HES-SO (Switzerland), Univ. of Westminster (UK), UPC (Spain), VUA (The Netherland), ZIB (Germany), CYFRONET (Poland), Univ. of Innsbruck (Austria)

Summary The CoreGrid Network of Excellence (NoE) aims at strengthening and advancing scientific and technological excellence in the area of Grid and Peer-to-Peer technologies. To achieve this objective, the Network brings together a critical mass of well-established researchers (161 permanent researchers and 164 PhD students) from forty-one institutions who have constructed an ambitious joint programme of activities. This joint programme of activity is structured around six complementary research areas that have been selected on the basis of their strategic importance, their research challenges and the recognised European expertise to develop next generation Grid middleware, namely:

- knowledge and data management;
- programming models;
- architectural issues: scalability, dependability, adaptability;
- Grid information, resource and workflow monitoring services;
- resource management and scheduling;
- Grid systems, tools and environments.

IP BIONETS

Type: European IP FP6
Title: Bio-inspired Networks and Services.
Personal responsibility: Coordination of deliverables, local responsible for a workpackage
Dates: 2006-2011
Partners: CREATE-NET (Italy), University of Basel (Switzerland), TUB (Germany), University of Passau (Germany), Budapest University of Thechnologie and Economics (Hungary), Nokia Corporation, VTT (Finland), INRIA (France), National and Kapodistrian University of Athens (Greese), Telecom Italia. London School of Economics and Political Science (UK). Sun Microsystems Spain.

Summary The motivation for BIONETS comes from emerging trends towards pervasive computing and communication environments, where myriads of networked devices with very different features will enhance our communication and tool manipulation capabilities. Traditional communication approaches are ineffective in this context, since they fail to address several new features: a huge number of nodes including low-cost sensing/identifying devices, a wide heterogeneity in node capabilities, high node mobility, the management complexity, the possibility of exploiting spare node resources. Nature and society exhibit many instances of systems in which large populations are able to reach efficient equilibrium states and to develop effective collaboration and survival strategies, able to work in the absence of central control and to exploit local interactions. We seek inspiration from these systems to provide a fully integrated network and service environment that scales to large amounts of heterogeneous devices, and that is able to adapt and evolve in an autonomic way. BIONETS overcomes device heterogeneity and achieves scalability via an autonomic and localised peer-to-peer communication paradigm. Services in BIONETS are also autonomic, and evolve to adapt to the surrounding environment, like living organisms evolve by natural selection. Biologically-inspired concepts permeate the network and its services, blending them together, so that the network moulds itself to the services it runs, and services, in turn, become a mirror image of the social networks of users they serve. This new paradigm breaks the barrier between service providers and users, and sets up the opportunity for "mushrooming" of spontaneous services, therefore paving the way to a service-centric ICT revolution.

FUI CloudForce

Type: Programme d'Investissements d'Avenir - FUI
Dates: 2012-2014
Personal responsibility: Task coordinator
Partners: France Télécom, ActiveEon, Armines, Bull, eNovance, eXoINPT/IRIT, INRIA, OW2, peergreen, PetalsLink, Télécom Paris Tech, Télécom Saint Etienne, Thalès Communication, Thalès Services, Univ. Joseph Fourier/LIG, Univ. de Savoie/LISTIC, UShareSoft.

Summary CloudForce project will provide a software engineering platform for developing, deploying, and administrating collaborative cloud applications. It targets especially infrastructures with multiple IaaS. The project will also provide a PaaS platform compatible with multiple Iaas for deploying, orchestrating, benchmarking, self-managing, and provisioning applications.

Associate Team SCADA

Type: INRIA - Associate team
Dates: 2012-2014
Personal responsibility: Project coordinator
Partners: OASIS, NIC-Labs (Chile).

Summary Besides a formal collaboration between NIC Labs and OASIS team, the aim of the project is to contribute to programming models and languages for programming, running and debugging parallel and distributed applications. For this we will contribute both from at theoretical and practical perspectives to the design of languages, and their implementation and formalisation. In this project

we will focus on composition models allowing to put together individual sequential code into complex applications featuring parallelism and distribution. More precisely we focus on two such composition models: algorithmic skeletons and software components.

I also participated to the following projects: GCPMF (ANR - 2006-2008), GridCOMP (EU FP6-Strep - 2006-2009), Reseco (Stic-Amsud - 2006-2009), MCorePHP (ANR blanc international - 2010-2012).

Other activities

• Program committee: *FMOODS/DAIS 2003 Student Workshop*, *FOCLASA* 2009 to 2012, *FESCA* 2009 to 2012, Sophia Antipolis Formal analysis local workshops.

• Reviews for many other conferences, and for the following journals: SCP, TCS, MSCS, TOPLAS, ComSIS

Formal Models for Programming and Composing Correct Distributed Systems

Abstract

My research focuses on distributed programming models, more precisely using objects and components. In this area, I provided tools easing the programming of large-scale distributed applications and verifying their correct behaviour. To facilitate the programming of distributed applications, I contributed to the design and the development of languages with a high level of abstraction: active objects, algorithmic skeletons, components. To verify correction of the behaviour of an application, I have contributed to the creation of tools for specifying and verifying behavioural distributed applications. My work aims to provide a strong model of programming languages, libraries, and runtime environments provided to the developer, and to guarantee the safe behaviour of distributed applications.

During my thesis, I developed the ASP calculus for modelling the behaviour of active objects and futures. Since, we created a functional version of this calculus and formalised it in Isabelle/HOL. I also strongly contributed to the definition of a distributed component model - the GCM (Grid Component Model) -, to its formalisation, and to its use for programming adaptive or autonomous components. Finally, I contributed to the specification and behavioural verification of programs based on active objects and components, in order to ensure their safe execution. Currently we are working both on a multi-threaded extension of the active object model, better suited for multi-core machine, and on the use of formal methods to design and prove the correction of an algorithm for broadcast on CAN-like peer-to-peer networks (Content Addressable Network). This manuscript provides an overview of all these works.

Modèles Formels pour la Programmation et la Composition de Systèmes Distribués Corrects

Résumé

Mes travaux de recherche portent sur les modèles de programmation distribuée, principalement par objets et composants. Dans ce domaine, j'ai travaillé à fournir des outils facilitant la programmation d'applications distribuées à large échelle et vérifiant la correction de leur comportement. Pour faciliter la programmation d'applications distribuées je me suis intéressé à la mise au point de langages avec un fort niveau d'abstraction: objets actifs, squelettes algorithmiques, composants. Afin de vérifier la correction du comportement d'une application j'ai collaboré à la mise au point d'outils de spécification et de vérification comportementales d'applications distribuées. Mes travaux ont pour but de fournir un modèle formel des langages de programmations, des bibliothèques, et des environnements d'exécution fournies au programmeur afin de garantir un comportement sûr des applications distribuées.

Ma thèse m'a permis de mettre au point le calcul ASP modélisant le comportement des objets actifs et des futurs. Depuis, nous avons créé une version fonctionnelle de ce calcul que nous avons modélisé en Isabelle/HOL. Aussi j'ai fortement contribué à la définition d'un modèle à composants distribués – le GCM (Grid Component model)–, à sa formalisation et à son utilisation pour programmer des composants adaptables ou autonomes. Enfin, j'ai contribué à la spécification et la vérification comportementale des programmes utilisant des objets actifs et des composants afin de garantir la sûreté de leur exécution. Actuellement, nous travaillons à la fois à une extension multi-threadée du modèle à objets actifs mieux adaptée aux machines multi-cœurs, et à l'utilisation de méthodes formelles pour mettre au point et prouver la correction d'un algorithme de diffusion pour réseau pair-à-pair de type CAN (Content Adressable Network). Ce manuscrit fournit une vue d'ensemble de tous ces travaux.