Project No. FP6-004265

CoreGRID

European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies

Network of Excellence

GRID-based Systems for solving complex problems

Deliverable D.PM.04 – Basic Features of the Grid Component Model (assessed)

Due date of deliverable: September 25, 2006
Actual submission date: March 7, 2007

Start date of project: 1 September 2004 Duration: 48 months

RESPONSIBLE PARTNER: INRIA

Revision: *Final*

Project co-funded by the European Commission within the Sixth Framework Programme (2002–2006)		
Dissemination level		
PU	Public	PU

Keyword list: programming model, components, Grid, high performance, scalability

Contents

Executive Summary

Abstract

This document describes a Grid component model (called GCM). It defines the main features to be included in the GCM, as currently assessed in the Programming Model Institute. By defining the GCM, the Institute aims at the precise specification of an effective GCM. Fractal has been chosen by the CoreGrid community as the basis for the definitions of the GCM: The GCM features is roughly defined as an extension to the Fractal specification in order to better target Grid infrastructure.

The Institute actually expects several different implementations of the GCM, supporting all or most of the features presented in this deliverable; such implementations do not necessarily need to rely on existing Fractal implementations.

GridCOMP is an European project that will provide a reference implementation for the GCM. As planned in the GridCOMP description of work, this definition of the GCM specification will be used as the initial component model for the reference implementation that should be provided by the GridCOMP project.

Though this document is assessed by all the partners of the Programming Model Institute, further versions of the GCM specification are expected, possibly including improvements.

The GCM abstract view

In this Section, we outline the high level features supported by the GCM component model. The section is meant to provide an abstract view of the *logical* features of the component model, right before dealing with the GCM *specification* that may be directly used to provide implementations of the GCM. By the way, we collect here a description of those properties and features demonstrated and supported by GCM that actually *differentiate* GCM from the other component models that have been developed so far. For a further analysis of the features that should be provided by a Grid component model and a more comprehensive study of these features, please refer to a previous deliverable of the Programming Model institute that is intended as a rationale for the GCM [18].

Let us first start by the definition of what is a component: A component is a software module, with a standardized description of what it needs and provides, that can be be manipulated by tools for composition and deployment.

First of all, the GCM is a **hierarchical** component model. This means users of GCM have the possibility to program GCM components as compositions of existing GCM components. The new, composite components programmed this way are first class components, and they can be used in every context where elementary components can be used. Users (programmers) do not need to perceive the fact that some components are composite, unless they explicitly want to explore their content. This property is already present in existing component models. In particular, the Fractal component model [25, 11] assumes components can be hierarchically composed, and this is one of the reasons that contributed to take the Fractal component model as the reference model in the Programming model Institute and to consider Fractal as the model to be extended in order to define GCM.

The GCM allows component interactions to take place with several distinct mechanisms. In addition to classical "RPC-like" use/provide (or client/server) ports, GCM allows **data**, **stream** and **event ports** to be used in component interaction. Furthermore, **collective interaction patterns** (communication mechanisms) are also supported. Data ports allow data sharing mechanisms to be implemented. Using data ports components can be used to encapsulate shared data while preserving the ability to properly perform ad hoc optimization of the interaction among components sharing data. Stream ports allow one way, data flow communications to be implemented. Despite stream ports can be easily emulated by classical use/provide ports, the explicit usage of stream ports allows much more effective optimizations to be performed in the component run time support. Event ports may be used to provide asynchronous interaction capabilities to the component framework. Events can be subscribed and generated. Furthermore events can be used just to synchronize components as well as to synchronize *and* to exchange data while the synchronization takes place. This is much in the sense of what actually is already supported in the CCM component model.

Concerning collective interaction patterns, GCM supports several kind of collective ports, including those allowing to implement structured interaction between a single use port and multiple provide ports (multicast collective) and between multiple use ports and a single provide port (gathercast collective). The two parametric (and therefore customizable) interaction mechanisms allow to implement all the interesting collective interaction patterns deriving from the usage of composite components. The current definition of GCM does not exclude the possibility to have further collective interaction patterns in the future, in case the ones included in the current definition turn out to be insufficient to support commonly used Grid component patterns.

GCM is aimed to be used in Grid contexts, which are characterized by highly dynamic, heterogeneous and networked target architectures. GCM is therefore assumed to provide several levels of **autonomic managers** in components, that take care of the **non functional** features of the component programs. GCM components have thus two kind of interfaces: a functional and a non functional one. The functional interface hosts all those ports contributing to implement the functional features of the component, i.e. those feature directly contributing to the computation of the result expected by the component. The non functional interface hosts all those ports needed to support the component manager activity in the implementation of the non functional features, i.e. all those features contributing to the efficiency of the component in the achievement of the expected (functional) results but not directly involved actual result computation. Each GCM component therefore contains one or more managers, interacting with other managers in other components via their non functional interfaces. Such managers interact with both the other managers and the internal components of the same component using the proper mechanism provided by the GCM component implementation. Managers are assumed to be present in each component that manage all those aspects related to grid that contribute to the efficient execution of the component on the Grid target architecture.

GCM component architecture is described using an ADL (Architecture Description Language) that describes the component system using composition and binding of sub-components. ADL actually decouples functional program development from the actual tasks needed to deploy, run and control the components on the component framework. In GCM, the ADL is mostly inherited from the Fractal ADL.

Last but not least, GCM component model supports **interoperability** at several levels. First, interoperability is guaranteed in terms of the ability to support several grid middleware environments as possible platforms used to implement GCM and, in particular, to host the GCM framework. Second, interoperability is guaranteed by the possibility of wrapping GCM components into standard Web Services, in such a way that the WS framework can benefit from the “services” provided by the GCM framework. Third, naturally, GCM components are allowed to invoke standard Web Service services during their execution Current definition of GCM does not prevent the extension of the interoperability features to other frameworks in the future.

GCM supports the features mentioned above according to several compliance levels, in order to allow smooth transition to GCM from other existing component frameworks. Lower compliance levels accommodate components that still do not support all the features required by the GCM model, but at least can be identified as GCM components with limited support for the GCM features. High compliance levels host full featured GCM components.

The above mentioned features characterize GCM with respect to the other component models currently available. As an example, GCM can be characterized as CCA (Common component architecture) plus hierarchical composition, advanced communication patterns and autonomic control, whereas it can be characterized against Fractal as only supporting autonomic control and advanced communication patterns in addition to the features already supported by Fractal. In this document, a "specification" of GCM is discussed that further details how the features mentioned above are accommodated in the GCM framework. The specification is closer to GCM implementation, however, than the abstract view of the model we presented in this section. These abstract view should be intended as a "qualitative" only description of the model assumed and designed by GCM.

It is important to note that this component model must be suitable both for implementing Grid applications and Grid platforms themselves, with both of them benefiting from having the above features. For example, adaptativity is a key issue for programming Grid application that can be deployed on heterogeneous environments, but this also means that Grid platforms will themselves be deployed and have to manage heterogeneous systems, consequently such platforms would necessitate an even stronger support for adaptativity than the applications themselves.

Main Technical Contributions of the GCM Specification

The proposal for the specification of the GCM include the following aspects:

- *Fractal as the basic component architecture:* Fractal defines a highly extensible component model which enforces separation of concerns, and separation between interfaces and implementation. Fractal is not particularly intended at distribution, and Grid specificities need to be taken into account in the definition of the GCM.

- *An Architecture for the GCM:* this document presents the basis for defining an *abstract view* of the GCM. The final version of the GCM should include standard definitions for this abstract view. Such a high-level view should allow all the partners to define a common view of what should be in a component model for the Grid, thus allowing interoperability. The following architecture has been proposed for concretely defining the GCM:

    1. Component Specification as an XML schema or DTD
    2. Run-Time API defined in several languages
    3. Packaging described as an XML schema

- Communication Semantics Standards: GCM components should include various communication semantics, however we chose to particularly support asynchronous method calls as the default case. It is however important to notice that the definition of GCM interfaces should allow for any kind of communications (e.g., streaming, file transfer) either synchronous or asynchronous.

- *Deployment of components relying on Virtual Nodes:* Virtual Nodes are abstractions allowing a clear separation between design infrastructure and physical infrastructure. Virtual Nodes are used in the code or in the ADL and abstract names and creation and connection protocols to physical resources, from which applications remain independent. Virtual Nodes are optional.

- *Multicast and Gathercast Interfaces:* 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.

- *Component Controllers*: To provide dynamic behavior of the component control, we propose to make it possible to consider a controller as a sub-component, which can then be added, plugged or unplugged dynamically.

  This approach gives a better adaptivity, both with respect to the platform, and with respect to the controlled components.

- *Autonomic Components:* 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: They correspond to Fractal *controllers*.

Concerning the status of the GCM specification, feedback from the implementation is still necessary before agreeing on a final specification. However, we consider that the specification is precise enough to allow for first implementations of the GCM to be realized and used. We hope further works inside the CoreGrid programming model institute but also inside the GridCOMP project, and any other implementation of the GCM to contribute to the finalization of the specification.

1 Introduction

This document is an assessed definition for a standard Grid component model that is being defined by the Programming Model Institute. Its first aim is to target the specificities of Grid computing, and the kind of components we want to address. In order to fulfill those requirements, we first describe an abstract definition of the GCM and then give a specification of the GCM.

The general strategy adopted by the Institute is to rely on the Fractal specification as much as possible, because the Fractal specification provides us with a terminology and abstract API relative to components. We expect the GCM to be an extension of the Fractal specification, if necessary specifying some choices and extensions of Fractal that we consider as necessary in the context of Grid components. As for the Fractal model, we expect the GCM to be very extensible and generic, that is why we only aim at providing very generic APIs allowing different implementations of the GCM to be realized, and different components implementing the specification to communicate. Thus, we expect different GCM conformance levels to be defined, and different implementations of GCM components to be characterized by their conformance level. Because of the extensible and generic aspect of the model, we also expect extensions of the GCM, and specific refinements to be defined in the future, not necessarily inside the Institute.

The existence of different levels of conformance, and thus different versions of the GCM is a crucial feature here. This results from discussions inside the Institute, and is similar to what exists in Fractal. It should allow every CoreGrid partner, and more generally, every user of the GCM to provide different implementations of the component model, still allowing all those implementations to interoperate and to be comparable.

This proposal is concluded by an overview of the features that we identified as requirements for a Grid component model, and the way we propose to fulfill the requirements in the final specification.

1.1 A Programming Model for the Grid

The Institute on programming models aims to deliver a definition of a component programming model that can be usefully exploited to design, implement and run high performance, efficient Grid applications. The same component model should also be exploited in the design of tools supporting Grid programming, such as in the development of PSEs (problem solving environments) or in the development of tools supporting resource management or system architecture related activities.

It is assumed that the component based programming model's main aim is to address the new characteristic challenges of Grid computing - heterogeneity and dynamicity - in terms of programmability, interoperability, code reuse and efficiency. Grid programmability, in particular, represents the biggest challenge. Grid programs cannot be constructed using traditional programming models and tools (such as those based on explicit message passing or on remote procedure call/web service abstraction, for instance), unless the programmer is prepared to pay a high price in terms of programming, program debugging and program tuning efforts.

The objective of this deliverable is to list the key features to be included in the GCM, as currently assessed in the Programming Model Institute.

1.2 Challenges and Requirements of the GCM

The Grid poses new challenges in terms of programmability, interoperability, code reuse and efficiency. These challenges mainly arise from the features that are peculiar to Grid, namely heterogeneity and dynamicity.

New programming models are required that exploit a layered approach to GRID programming which will offer user friendly ways of developing efficient, high performance applications. This is particularly true in cases where the applications are complex and multidisciplinary. Within CoreGRID, the challenge is to design a component based programming model that overcomes the major problems arising when programming Grids.

The challenge, per se, requires that a full set of sub challenges will be addressed:

- A suitable programming model (that is user friendly and efficient) to program individual components is needed
- Component definition, usage and composition must be organized according to standards that allow interoperability to be achieved.
- Component composition must be defined precisely in such a way that complex, multidisciplinary applications can be constructed by the composition of building block components, possibly obtained by suitably wrapping existing code. Component composition must support and, in addition, guarantee scalability.
- Semantic must be defined, precisely modeling both the single component semantics and the semantics of composition, in such a way that provably correct transformation/improvement techniques can be developed.
- Performance/cost models must be defined, to allow the development of tools for reasoning about components and component composition programs

All of these sub-challenges must be dealt with taking into account that improvements in hardware and software technology require new Grid systems to be transparent, easy to use and to program, person centric rather that middleware, software or system-centric, easy to configure and manage, scalable, and suitable to be used in pervasive and ubiquitous contexts.

The essential characteristics proposed by the GCM include: Support for *reflection*, *Hierarchical structure*, Model *Extensibility*, Support for *adaptivity*, and *Interoperability*. Additionally the model should allow for *lightweight implementations*, in order to support the design of compact and portable implementations.

We also require that the GCM has a *well defined semantics*. Moreover, the GCM should take into account the necessity for its implementations to ensure high performance.

1.3 The Fractal Component Model

In this document, we will describe the GCM as an extension of the Fractal specification, and we will introduce the new features using a Fractal compliant terminology. In that sense, Fractal is considered as a common terminology to ground the definition and comparison of Grid extensions to be included in the GCM. Fractal [6, 11] has been chosen as the reference model for designing the GCM, because it is well-defined, hierarchical, and extensible. We define the GCM as a well-designed and sizable extension of Fractal. Indeed, we rely on Fractal concepts and specification for the design of the hierarchical component structure and clear separation between functional – i.e., content – and non-functional – i.e., controllers – aspects.

A brief summary of the Fractal specification is given in Section 4, but the reader should refer to the Fractal specification [11] for a complete description of the Fractal model. Similar to most component models, Fractal uses an object-oriented terminology [26, 16]. We also adopt this terminology for the GCM; however, it is possible to implement GCM components on top of any kind of language (provided the designer of the component framework knows the Object-Oriented terminology).

Fractal is first an abstract component model; it has a formal specification, which can be instantiated in different languages, like, for example, Java or C. In practice, Fractal actually has several different implementations in several languages. The GCM is based on this formal specification and proposes an extension adapted to Grid computing. It is not tied to the reference implementation (Julia [6]), which is not targeted at distributed architectures. Fractal does not constrain the way(s) the GCM will be implemented, but it provides a basis for its formal specification, allowing us to focus on the Grid specificities.

Fractal is a multi-level specification, where depending on the level some of the specified features are optional. Section 4 will recall those levels, called conformance levels in the Fractal specification.

1.4 Principles and Organisation of the GCM Definition

A component model definition can be divided into three aspects:

- The aspect of the *abstract model*: Section 2 describes the abstract GCM. A component framework is stated to be *compliant to the abstract GCM* if it is a component model that provides, in some way, all the abilities defined by the abstract GCM

- The aspect of the *specification*: Giving a specification of the GCM is the objective of Sections 3 to 10, this specification is not final but precise enough to allow for prototype implementations of the GCM. Most importantly, this specification is intended to be *extensible.* A framework that would fulfill the specification described in those sections is stated to be *compliant to the GCM specification.*

- The aspect of the (reference) implementation that is clearly out of the scope of the CoreGrid institute on Programming Models. For instance, the GridCOMP project aims at providing such a reference implementation.

The remaining of this document is organized as follows. The GCM component model relies on the Fractal specification [11]. We first give an abstract view of the component model we propose (Section 3), then we recall briefly the Fractal component model specification in Section 4, please refer to [11] for the complete specification of the Fractal reference model.

Section 5 defines a (set of) communication semantics adapted to the GCM. Virtual Nodes, a convenient way of specifying a Grid application distribution, are defined in Section 6. Section 7 define multicast and gathercast interfaces that allow to specify one-to-many and many-to-one communications. Concerning non-functional aspects Section 8 proposes a standard for adding dynamic controllers into the component model, i.e. dynamically reconfigurable controllers which are crucial in an heterogeneous and dynamically evolving environment such as Grid. For specific non-functional features, a Grid application would highly benefit from autonomicity: the ability to dynamically and autonomously reconfigure and adapt an application to its environment and required quality of service; these aspects are defined in Section 9.

Section 10 introduces the specification of behavioural protocol into the ADL in order to allow the dynamic verification of component composition.

Finally, Section 11 summarizes the required features for a well-designed Grid component model, together with the solution proposed by the GCM to implement those features.

Except from the Fractal specification section (Section 4), all the features presented in this section are not part of Fractal and should be part of the GCM extension to Fractal. However, as explained in the following, some of the extensions proposed here have already been suggested inside the Fractal community, but the GCM is to our knowledge the first proposal to standardize those concepts.

2 A Comprehensive Grid Component Model

The GCM is first defined as an abstract model. This abstract model is defined by the set of abilities that must be provided by a component model specification to meet the requirements expressed in Section 1.2. We first define a component as follows:

> Each component can be viewed as a black box, for which the specification of its interface to the external components is sufficient. More precisely, each component defines a set of ports (also called interfaces) that define the type of the component. Ports can be connected (bound) together provided they are compatible. The compatibility relation must be defined at the level of the component model and must be verifiable both at composition time but also at runtime.

The usual example of ports (including kinds and compatibility) is the following: each port is either a client one (also called required interface) or a server one (also called provided interface); and Client ports can be connected (bound) to server ports provided they are of compatible type[1].

A component framework is stated to be *compliant to the abstract GCM* if it allows to instantiate components fitting to the preceding definition, and moreover, it provides the following abilities:

- *Hierarchical composition*: allowing a GCM component to be built from several components. A component that is built from others (called composite component) exposes its content.
- *Structured communications*: support for many-to-one and one-to-many communications but also different communication semantics including asynchronous remote method invocation, events based, and streaming based communication.
- *Autonomicity*: support for autonomic behaviour of GCM components, that is self-configuration, self-optimization, and self protection.
- *Functional and non-functional adaptivity*: support for functional and component control dynamic adaptation, meaning reconfiguring both functional and control aspects. This entails the ability to add, replace and remove dynamically both functional and non-functional entities, but also dynamically changing the (functional and non-functional) bindings between components.
- *Deployment*: support for some description of deployment for components, and for the deployment of any component architecture on the Grid.
- *Behavioural protocol specification*: support for behavioural protocol specification, that can be associated to components or generated, and verification of the correctness of the behaviour and the component composition.

[1]Provided a type system is implemented by the component model.

Those different functionalities are further specified in the next sections.

Most of these aspects can be defined as conformance levels, and thus different implementations of the GCM abstract model can implement those abilities more or less completely.

3 Architecture

First, we present the basis for defining an *abstract view* of the GCM. The final version of the GCM should include standard definitions for this abstract view (DTD or XML schema, complete API, ...). Such a high-level view should allow all the partners to define a common view of what should be in a component model for the Grid. This abstract view is at the level of defining what a primitive component is, what a hierarchical composition is, and what the various kinds of ports and interfaces are.

The following architecture has been proposed for concretely defining the GCM:

1. Component Specification as an XML document, conforming to a defined schema or DTD

2. Run-Time API defined in several languages

3. Packaging described as an XML document, conforming to a defined schema or DTD

The first part, using a schema to precisely define a component description language, a kind of ADL in XML, is the basic mechanism for defining inter-operable component descriptions. The second aspect, run-time API allows the manipulation of components at execution in a uniform manner. Finally, the packaging schema authorizes the development of common deployment tools. Details of the elements comprising each part of the specification are given below.

3.1 Component Specification as an XML Document

The first element in a component specification is the notion of *primitive components*. One must be able to define, from a given piece of code, the attributes of the component being constructed. A piece of standard code, or a module, is promoted to the status of a component. Then, provided we are targeting a hierarchical model to tackle the complexity and large scale nature of Grids, one must be able to compose the primitives to build *hierarchical* entities.

Of course, the specification of components, both primitive and composite, requires defining interfaces (various kind of ports), and the binding between those ports.

With respect to language interoperability, a key aspect of the proposed specification is reliance on external references (Java Interface, C++ .h, Corba IDL, WSDL) to specify the nature of ports. In this way, one can define components specific to a given platform, one can also specify a component that is exported in several interfaces (e.g. Java and C), or even exported with portable ports such as a WSDL definition.

Finally, the Grid aspects are covered through the specification of specific elements such as distribution, virtual Nodes, QoS, etc.

Figure 1 summarizes the structure and the key aspects of the component specification. An important feature of such a specification is the extensible nature of an XML Schema.

3.2 Run-Time API Defined in Several Languages

We propose to define a common run-time API for manipulating components at execution. The basic functions of the API will address:

- Life cycle management

- Introspection

- Definition of Primitive Components
- Definition of Composite Components (composition)
- Definition of Interfaces (ports)
  - Server, Client, event, stream, etc.
- Including external references to various specifications:
  - Java Interface, C++ .h, Corba IDL, WSDL, etc.
- Specification of Grid aspects:
  - Parallelism, Distribution, Virtual Nodes,
  - Performance Needs, QoS, etc.

Figure 1: Content of the component specification

- Basic Control (Monitoring, Reconfiguration, ...)
- Optimization

The languages and formats in which we would like to define related APIs for manipulation of Grid components at runtime include Java, C++, C#, Fortran.

The API will facilitate the portability and interoperability, and standard implementations of component infrastructures and containers will be possible; making such components interoperable within a given language. We believe it would not be realistic to attempt a direct multi-language infrastructure, with inter-language interoperability. However, one can define the WSDL specification of part of the run-time API. It would allow implementation and deployment of Web Services that provide portable run-time manipulation of components (management of life cycle, etc.). In any case, for the sake of efficiency, and expressive power, language specific implementations are needed.

3.3 Packaging described as an XML Document

Together with the component specification defined earlier, there is also a need to provide more practical information about Grid components. For instance, one must specify the dependencies between codes, where to find the appropriate bundles, for what hardware platform, etc. Here is an incomplete list of such information:

- Requirements on the hardware platform
- Location of the code needed to instantiate the component on a platform
- Dependencies between versions
- Metadata driving reconfiguration (cf. Section 8.2), like cost functions or descriptions of the component's functional behaviour at different levels of abstraction.

With such information, the components may be deployed in various contexts. Defined by an XML document, conforming a predefined schema, this extensible specification will provide for generic tools to help solve a complex problem: large scale deployment on the Grid.

- Level 0: GCM ADL
- Level 1: GCM API (per language)
- Level 2.a: GCM with collective communications: Multicast and Gathercast
- Level 2.b: GCM with support for Event and Streaming Ports
- Level 2.c: GCM with Dynamic Component Controllers
- Level 2.d: GCM with Autonomic Controllers
- Level 2 = Level 2.a + 2.b + 2.c + 2.d
- Level 3: GCM API with Web Service Interoperability

Figure 2: Compliance levels for GCM Implementation and Interoperability

3.4 General Conformance Levels

Level 0 corresponds to the very minimal interoperability and integration (e.g. in CoreGrid) you get with being GCM: the capacity to view and edit GCM components with common tools.

Level 1: This is per language, so one can have an implementation that is, for example, GCM Level 1 Java Compliant, GCM Level 1 C Compliant, GCM Level 1 Corba Compliant, or GCM Level 1 .Net Compliant. It gives *programmer interoperability*, and program interoperability. One programmer being used to do GCM in Java can switch implementation. This level is strongly connected to Fractal conformance levels, and correspond to an adaptation of Fractal API (called GCM API) and specified in appendix C. This conformance level is further detailed by the Fractal conformance levels given in Table 1

Level 2 gives an implementation with full capability. Detailed conformance levels will be given in Section 3.5:

- Table 2 further details level 2.a: collective communication conformance level.
- Table 4 further details level 2.c: dynamic controllers conformance level.
- Table 3 further details level 2.d: autonomic controllers conformance level.

Level 3 goes further than the functional web-service interoperability that should be implemented in the above levels. It is intended to provide full interoperability of the frameworks by the exportation of the complete GCM API as web-services. As such, GCM component management (e.g. binding, life-cycle) can be achieved across several languages.

It is important to note that those levels are independent and somehow parameterizable. Indeed, for example implementing level 0 is not necessary in order to implement level 1. A much more detailed specification of conformance levels will be given in Section 3.5.

3.5 GCM Detailed Conformance Levels: Summary

To summarize we define below the different conformance levels that will be defined in this deliverable. These levels relate strongly to the abstract level defined in Section 3.4 but are much more detailed, these levels can only be clearly understood after the detailed presentation given in this deliverable, but we put them here for better comparison with the abstract conformance levels.

A component model is said to conform the GCM specification up to a conformance level, this conformance level is in fact a tuple of conformance levels according to the different aspects of the GCM specification. “GCM Levels” are the level defined in Figure 2; each of the table below details one of these levels.

Fractal conformance level (GCM Level 1)

Fractal conformance level	Requirement
0	at this level nothing is mandatory. Fractal components are like simple objects
0.1	same as level 0, with the additional requirements that all components with configurable attributes must provide the AttributeController interface, that all components with client interfaces must provide the BindingController interface, that all components that expose their content must provide the ContentController interface, and that all components that expose their life cycle must provide the LifeCycleController interface. Of course, these requirements do not prevent components from providing additional control interfaces, including extensions and alternatives of the previous interfaces.
1	same as level 0, with the additional requirement that all components must provide, at least, the Component interface.
1.1	same as level 1, with the same additional requirements as for level 0.1, concerning the control interfaces
2	same as level 1, with the additional requirement that all component interface references must be castable to Interface.
2.1	same as level 2, with the same additional requirements as for level 0.1, concerning the control interfaces
3	same as level 2, with the additional requirement that all the components must use (an extension of) the type system defined in the Fractal specification.
3.1	same as level 3, with the same additional requirements as for levels 0.1, 1.1 and 2.1, concerning the control interfaces.
3.2	same as level 3.1, with the additional requirement that a bootstrap component must be accessible from a “well-known” name. This bootstrap component must provide a GenericFactory and a TypeFactory interface. Moreover, the GenericFactory interface must be able to create components with any control interfaces in the set of control interfaces defined in section 4 (and, in particular, primitive and composite components). Finally, this interface must also be able to create (3.2 level) primitive components encapsulating 0.1 level components.
3.3	same as level 3.2, with the additional requirement that the GenericFactory interface of the bootstrap component must be able to create primitive and composite template components.

Table 1: Fractal conformance level

Collective Communication conformance level (GCM Level 2.a)

Collective communication conformance level	Requirement
0	no collective communications
1	one-to-many communication using multicast interfaces
2	level 1 + gathercast interfaces (many-to-one communication and synchronization)
3.a	Level 2 + extended typing and dynamic reconfiguration of data dispatch and aggregation (as specified in Section 7.3.1)
3.b	Level 2 + optimization of MxN synchronization and communication (Section 7.4)
3	Both levels 3.a and 3.b

Table 2: Collective communication conformance level

Autonomicity conformance level (GCM Level 2.d)

Autonomicity conformance level	Requirement	Fractal conformance level required
1	no autonomic control	0
2	low-level, passive autonomicity: components include one or more `AutonomicController` interface (named `XXX-autonomic-controller`)	1.1, 2.1, 3.1 and above
3	high-level, active autonomic control: same as level 2,but each implemented autonomic aspect should provide: • A (server) operation to submit a QoS contract to the component; • A (client) operation able to signal contract violations.	1.1, 2.1, 3.1 and above

Table 3: Autonomicity conformance level

Adaptivity Conformance Level (GCM Level 2.c)

Dynamic controller conformance level	Requirement	Fractal conformance level required
0	full functional adaptativity, no component controller	?.1
1	full functional adaptativity, membrane can be specified on the form of component controller *at instantiation time*	?.1
2	component controllers exist at runtime and can be reconfigured, but non-functional interface are not pluggable	?.1
3	same as 2, where client and server non functional interface exist and can be handled as functional ones (allowing binding between non-functional interfaces)	?.1
3.1	same as 3, with the additional requirement that each component allows addition and removal of non-functional ports to an existing component	2.1
4	same as 3, with the additional requirement that Fractal's type system is extended to take into account non functional interfaces	3.1
4.1	same as 4, except that the type system is extended into a dynamic type system; like in 3.1 each component allows addition and removal of non-functional ports to an existing component, but binding operations on non-functional interfaces are subject to dynamic type-checking	3.1

Table 4: Dynamic controller conformance level

Adaptivity is partitioned into two aspects.

Functional adaptivity that is specified by Fractal specification: all levels supporting binding and content controller are adaptive at the functional level (Fractal conformance levels 0.1, 1.1, 2.1, 3.1 and above – denoted ?.1 in the following).

Non-functional adaptivity is provided in the GCM specification by dynamic controllers as detailed in Table 4.

Note that, even at the highest conformance level we do not require that Fractal's controllers are implemented as components themselves: the only requirement is to be able to instantiate user-defined controllers as components. In other words, the only requirement is for the component model to *accept user defined controllers defined by components*. Note that this neither requires that user-defined controllers are given on the form of components, nor that predefined controllers can be handled on the form of components.

Finally, recall that though assessed by all partners of the programming model institute and detailed enough for allowing first implementations of the GCM to be performed, this specification will possibly evolve in the next months or years.

4 Fractal Specification

The GCM relies on the Fractal component model; globally, we refer to the Fractal specification as a basis for the specification of the GCM, and consider the GCM as an extension to the Fractal specification. We present here a brief summary of the crucial features of the Fractal component model taken from the Fractal specification [11].

Fractal defines a highly extensible component model which enforces separation of concerns, and separation between interfaces and implementation.

Fractal is based on the following definitions:

- *Content:* one of the two parts of a component, the other one being its controller. A content is an abstract entity controlled by a controller. The content of a component is (recursively) made of sub components and bindings.
- *Controller* or *membrane*: one of the two parts of a component, the other one being its content. A controller is an abstract entity that embodies the control behavior associated with a particular component. A controller can exercise an arbitrary control over the content of the component it is part of (intercept incoming and outgoing operation invocations for instance).
- *Server interface:* a component interface that receives operation invocations (e.g. (a,I) in Figure 3).
- *Client interface:* a component interface that emits operation invocations (e.g. (b,J) in Figure 3).
- *Functional interface:* a component interface that corresponds to a provided or required functionality of a component, as opposed to a control interface.
- *Control interface:* a component interface that manages a "non functional aspect" of a component, such as introspection, configuration or reconfiguration, and so on.

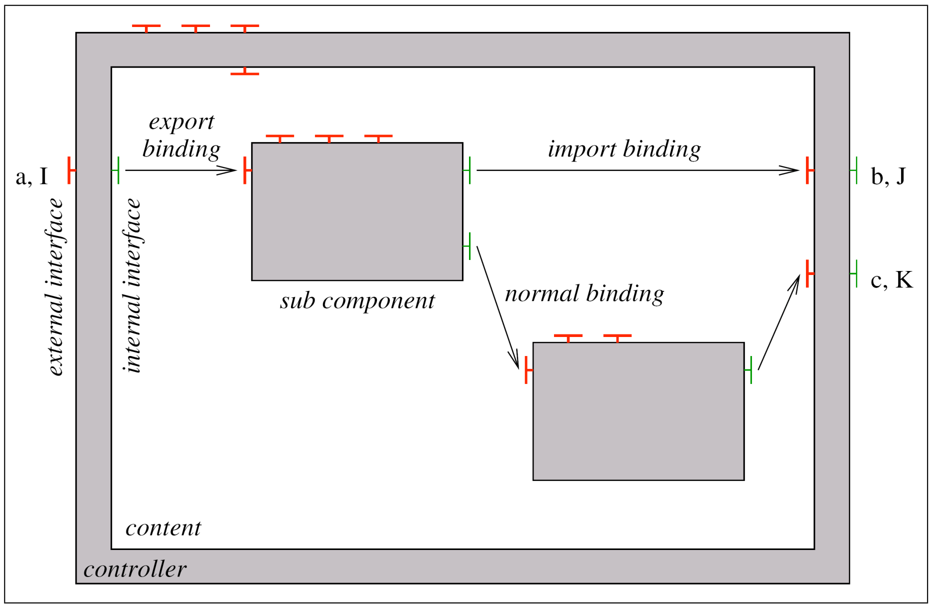

Figure 3: A composite component as defined in Fractal Specification

Figure 3 shows a composite component, its content (2 sub-components), and its membrane containing the controllers.

Fractal defines several conformance levels, mainly depending on the level of control exercised over the components. Different implementations of the GCM can rely on different conformance levels of Fractal, and thus provide more or less features among the ones defined in the Fractal specification.

The definition of the conformance level can be found in the Fractal specification. We recall those levels below:

- level 0: at this level nothing is mandatory. Fractal components are like simple objects. A Java object, a Java Bean, or an Enterprise Java Bean, for example, are conform to the Fractal component model of level 0.
  - level 0.1: same as level 0, with the additional requirements that all components with configurable attributes must provide the AttributeController interface, that all components with client interfaces must provide the BindingController interface, that all components that expose their content must provide the ContentController interface, and that all components that expose their life cycle must provide the LifeCycleController interface. Of course, these requirements do not prevent components from providing additional control interfaces, including extensions and alternatives of the previous interfaces.
- level 1: same as level 0, with the additional requirement that all components must provide, at least, the Component interface.
  - level 1.1: same as level 1, with the same additional requirements as for level 0.1, concerning the control interfaces.
- level 2: same as level 1, with the additional requirement that all component interface references must be castable to Interface.
  - level 2.1: same as level 2, with the same additional requirements as for levels 0.1 and 1.1, concerning the control interfaces.
- level 3: same as level 2, with the additional requirement that all the components must use (an extension of) the type system defined in the Fractal specification.
  - level 3.1: same as level 3, with the same additional requirements as for levels 0.1, 1.1 and 2.1, concerning the control interfaces.
  - level 3.2: same as level 3.1, with the additional requirement that a bootstrap component must be accessible from a “well-known” name. This bootstrap component must provide a GenericFactory and a TypeFactory interface. Moreover, the GenericFactory interface must be able to create components with any control interfaces in the set of control interfaces defined in section 4 (and, in particular, primitive and composite components). Finally, this interface must also be able to create (3.2 level) primitive components encapsulating 0.1 level components.
  - level 3.3: same as level 3.2, with the additional requirement that the GenericFactory interface of the bootstrap component must be able to create primitive and composite template components.

In the same way, we can imagine specifying in the GCM several conformance levels that clearly identify the different extensions of the GCM. Moreover a GCM component system can be composed of components with different conformance levels both in the sense of the Fractal specification and for the GCM.

Appendix C gives the GCM API, this API is an adaptation of the Fractal API as defined in the Fractal specification.

The DTD for the GCM ADL is presented in Appendix D, this DTD is inspired from the standard Fractal ADL.

5 Communication Semantics

As highlighted in [18], we aim to support several kinds of communication in the GCM. The kind of communication implemented/accepted by each interface of a component should be specified in the interface type (asynchronous method invocations being the default value), with the natural additional requirement that only interfaces with *compatible* communication paradigms can be bound together. The definition of the compatibility relation could, initially, be simply equality, meaning that only a client and a server interface which communicate in the same way can be plugged together; more generally, some kind of sub-typing can be defined between the communication types.

Fractal's composite bindings can be implemented in order to be able to plug together interfaces implementing different communication paradigms.

However, we consider the default semantics for communication to be **asynchronous method invocations**. Tags can be added in the ADL to specify interfaces with a different semantics. The way in which communication is implemented is beyond the scope of the GCM (e.g., nothing prevents asynchronous method call from relying on synchronous communication for transmitting the reified method calls).

Now let us focus on one of the most important alternative kinds of communication on the Grid: communication via streams. We could add four different tags in the ADL *Streaming push client*, *Streaming push server*, *Streaming pull client*, and *Streaming pull server*, in order to specify streaming communication, with some additional requirements for binding compatibility: a streaming push client interface (that sends data) can only be bound with a streaming push server interface receiving the same type of data.

The Fractal ADL could be extended in order to accept the following interface definition:

```
<interface name="r" role="stream push server" signature="dataType"/>
```

where dataType is the IDL or any type specification for the data that transits along the binding. The reflexive role is "stream push client".

For reference, here is an example of an interface definition in the current Fractal ADL:

```
<interface name="r" role="server" signature="java.lang.Runnable"/>)
```

It is obviously outside the scope of the GCM to specify how streaming is implemented. However, as an example of a possible implementation, we detail below how a pure object interface based specification of streaming communication would look like.

Example: Implementing Streaming with Object Interfaces Let us consider an implementation of the GCM that would only define GCM interface as object ones (each component interface would correspond to an object interface of a given object). Of course, an efficient implementation of streaming would probably implement GCM streaming interfaces with a dedicated mechanism instead of the object interfaces shown here. However, every implementation should be able to check typing compatibility between streaming interfaces.

In such an object interface setting, we have the following two additional requirements

- *Streaming push* requiring that each method of an interface implementing such a service has no return value.
- *Streaming pull* requiring that each method of an interface implementing such a service takes no argument.

For instance, in Java a "stream push" object interface can look like:

```
interface StreamPush {
 void put(Object);
}
```

The component that sends data performs invocation on the method `put` of its streaming push client interface. This triggers an invocation of the method `put` on the server interface of the destination component, and thus transmits data.

Stream pull interface can be specified similarly with a client interface that looks like:

```
interface StreamPull {
 Object get();
}
```

The component that requires data performs invocation on the method `get` of its streaming pull client interface; which results in an invocation on the server interface of the destination component, the data is returned to the client component via the result returned by the method.

6 Virtual Nodes

6.1 Definitions

Virtual Nodes are abstractions allowing a clear separation between design infrastructure and physical infrastructure. Virtual Nodes are used in the code or in the ADL and abstract away names and creation and connection protocols to physical resources, from which applications remain independent. Virtual Nodes are optional.

The *design infrastructure* - or *virtual infrastructure* - corresponds to the targeted deployment infrastructure of components. The virtual infrastructure may define specific constraints, such as the *cardinality* of the virtual node, which can be single, or multiple. A *single virtual node* requires a unique physical node at deployment time, whereas a *multiple virtual node* requires several physical nodes.

The *physical infrastructure* is the infrastructure which is available at deployment time. Deployment on a physical infrastructure usually implies the use of connection or creation protocols and the naming of existing resources (which may be hidden by a resource broker framework, in which case the physical infrastructure in our terminology refers to the resources requested from this resource broker).

The definition of virtual nodes together with a few items of information use the following syntax. The syntax could include a constraint XML file, for instance as in:

```
<virtualNodesDefinition>
  <virtualNode name="Dispatcher" property="unique_singleAO"/>
  <virtualNode name="Renderer" property="Multiple"
               constraintFile="RendererConstraints.xml" />
</virtualNodesDefinition>
```

In the example above, the constraint file "`RendererConstraints.xml`" allows one to specify properties that should be ensured by the allocated nodes. This specification permits a program to generate automatically a deployment plan, i.e. find the appropriate nodes on which processes should be launched.

In the future, we envisage the adjunction of more sophisticated descriptions of the application needs with respect to the execution platform, for instance topology of nodes, including point-to-point QoS, hardware or OS constraints, interconnect preferences. It can also include specification of the application behavior.

In the future, it might also be necessary to introduce constraints at the level of the component itself. These constraints will allow allow the specification of constraints between Virtual Nodes, generally concerning the topology of Virtual Nodes. For example one could specify to co-allocate two virtual nodes on the same machine or cluster.

6.2 Virtual Nodes and Components

Different ADLs usually use distinct virtual node names, and it may be adequate to rename some virtual nodes, particularly for collocation purposes. This renaming is done through the exportation and the composition of virtual nodes. The exportation of virtual nodes defines which virtual nodes may be renamed and is specified in the ADL. Only exported virtual nodes can be renamed. The exportation is actually a renaming, and it is possible to export already exported virtual nodes: this is called composing virtual nodes. Exportation preserves cardinality: a single virtual node is exported with a single cardinality, a multiple virtual node with a multiple cardinality.

An Example
For clarity, the following examples focus on virtual nodes in the ADL.

Suppose that a component named *client* uses a virtual node *myNode*, of cardinality single, and exports it as *client-vn*:

```
<exportedVirtualNodes>
  <exportedVirtualNode name="VN1">
    <composedFrom>
      <composingVirtualNode component="this" name="myNode"/>
    </composedFrom>
  </exportedVirtualNode>
</exportedVirtualNodes>
...
<virtual-node name="myNode" cardinality="single"/>
```

If *myNode* had a *multiple* cardinality, the exported virtual node VN1 would also be of multiple cardinality.

Suppose there is another component, named *server*, which exports the node *server-vn*, of cardinality single.

An application using the client and server component may decide to keep *client* and *server* in distinct locations, in which case it may export these nodes as *VN1* and *VN2* (for instance):

```
<exportedVirtualNodes>
  <exportedVirtualNode name="VN1">
    <composedFrom>
      <composingVirtualNode component="client" name="client-vn"
          />
    </composedFrom>
  </exportedVirtualNode>
  <exportedVirtualNode name="VN2">
```

```
    <composedFrom>
      <composingVirtualNode component="server" name="server-vn"
          />
    </composedFrom>
  </exportedVirtualNode>
</exportedVirtualNodes>
...
```

If, on the contrary, the client and server components should be collocated, say on VN1, then the ADL would specify:

```
<exportedVirtualNodes>
  <exportedVirtualNode name="VN1">
    <composedFrom>
      <composingVirtualNode component="client" name="client-vn"
          />
      <composingVirtualNode component="server" name="server-vn"
          />
    </composedFrom>
  </exportedVirtualNode>
</exportedVirtualNodes>
```

As a result, the client and server component will be collocated / deployed on the same virtual node. This can be profitable if there is a lot of communications between these two components.

Although this example is simplistic, one can foresee a situation where the components would be prepackaged components, where their ADL description could not be modified ; the exportation and composition of virtual nodes allow the adaptation of the deployment of the system depending on the available infrastructure. Collocation as well as separation can be specified in the enclosing component definition.

6.3 Virtual Nodes Deployment

Appendix E provides the specification of a schema for deploying onto many schedulers, or using de facto Grid standards protocols. The following protocols are currently specified: ssh, gsissh, rsh, rlogin, LSF, PBS, Sun grid engine, Oar, Prun, Globus, Unicore, glite EGEE, Arc Nordugrid.

It also specifies a way to register a Virtual Node into a given naming service, using a specific protocol (e.g. RMI registry, JINI, HTTP registry, IBIS/RMI Registry Service), to acquire a given Virtual Node using a lookup protocol (e.g. RMI registry or a P2PService), and to use standards Files Transfers protocols (scp, rcp, Unicore, Arc Nordugrid).

For Grid programmers, this permits to develop components that can be deployed onto various infrastructure in a portable manner, and to integrate them into Service Oriented Architectures (SOA).

6.4 Summary: Virtual Nodes in the GCM

Virtual Nodes are abstractions capturing information about how a given component can be deployed on a Grid. They may be extended to include various kinds of constraints or preferences which provide information for either the deployer, or the automatic Grid computing tools such as schedulers and allocators. One can envisage more sophisticated information such as, for instance, topology information, QoS requirements between the nodes, etc. Having a standard definition for such information within the GCM will make possible the interoperability of tools within the CoreGrid community and beyond.

The final version of the GCM specification will precisely define the composition of virtual nodes.

7 Multicast and Gathercast Interfaces

GCM components do not *a priori* restrict the kind of code they may embed. Parallel codes, and especially SPMD codes like MPI-like based code, are very common in high performance computing, GCM components have to support such kinds of code.

We propose integrating the notion of collective interfaces into the component model, so that it will be possible to expose a specific collective behavior at the level of the interfaces. Collective interfaces correspond to new kinds of cardinalities for interfaces: *multicast*, and *gathercast*. Each of these cardinalities provides facilities for collective communications, and their behavior is customizable. To initially configure and later dynamically customize their behavior, a collective interface controller is associated with each component defining such collective interfaces.

Collective interfaces allow one to have collective operations without relying on intermediate components (composite bindings in Fractal specification). The objective is to simplify the design of component-based applications, ensure type compatibility in a direct manner. Of course, the model still allows for the use of explicit binding components for non-collective interfaces, in case of specific requirements for inter-component communications, for instance when binding interfaces of incompatible types.

The multicast and gathercast interfaces are configurable, which allows these interfaces to suit various situations. Each collective interface is associated to a *collective interface controller* for configuration purposes[2].

Solutions to the problem of data distribution have been proposed within PaCO++ / GridCCM [8]; these solutions can be seen as complementary to the basic distribution policy specified here. Extensions of multicast and gathercast specification should include the possibility to express such distribution policies.

Preliminary Remarks

- The examples provided in this section use the Java language, but the proposal is not tied to this language.
- 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*.

7.1 Multicast Interfaces

Multicast interfaces provide abstractions for one-to-many communications.

7.1.1 Definitions

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

On one hand, a multicast client interface distributes invocations to connected server interfaces. On the other hand, a multicast server interface explicitly exposes a multicast behavior, and forwards a single invocation either to a complementary

[2]For clarity reasons, those collective interface controllers are not shown on the figures

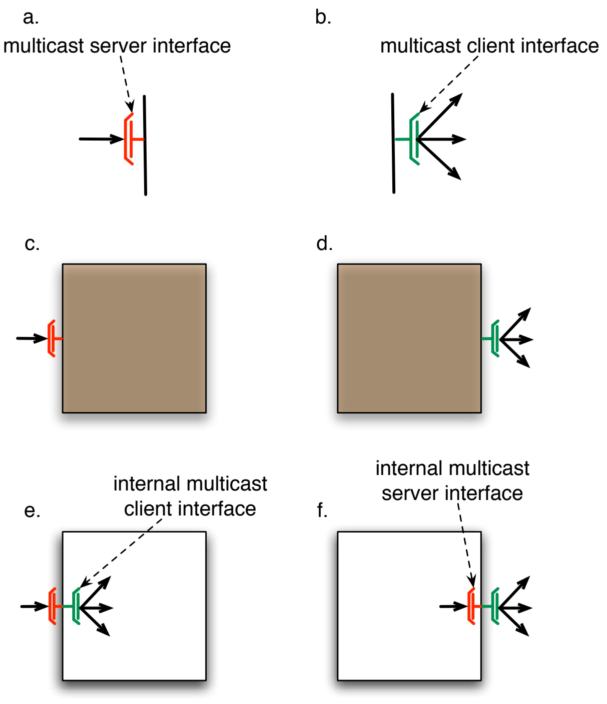
a.
multicast server interface
b.
multicast client interface
c.
d.
e.
internal multicast client interface
f.
internal multicast server interface

Figure 4: Multicast interfaces for primitive and composite components

multicast client interface in the case of a composite component (Fig. 4.d), or to a contractually defined implementation code in the case of a primitive component (Fig. 4.c).

> A *multicast server interface* transforms each single invocation into a set of invocations that are forwarded either to implementation code of a primitive component (Fig. 4.c), or to bound server interfaces of internal components (Fig. 4.d).

> A *multicast client interface* transforms each single invocation coming from either implementation code of a primitive component (Fig. 4.e) or from an internal component (Fig. 4.f) into a set of invocations to bound server interfaces of external components.

When a single invocation is transformed into a set of invocations, these invocations are forwarded to a set of connected server interfaces. A multicast interface is unique and exists at runtime (it is not lazily created). The semantics of the propagation of the invocation and of the distribution of the invocation parameters are customizable, and the result of an invocation on a multicast interface - if there is a result - is always a list of results. Invocations forwarded to 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.*

This specification does not make any assumption about the communication paradigm used to implement the multicast invocations ([15, 21]).

7.1.2 Signatures of methods

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

The bindings between a multicast interface and server interfaces can be regarded as *composite bindings*, because there is a typing conversion, from return type `T` in a method of the server interface, to return type `list of T` in the corresponding method of the multicast interface. The framework must transparently handle the type conversion between return types, which simply is an aggregation of elements of type T into a structure of type list of 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();
}
```

Section 7.3.1 defines a generalized aggregation mechanism that returns any kind of reduced value(s) instead of systematically returning a list. Then the return type of a multicast interface could be any type, a list is not mandatory any more.

7.1.3 Distribution of Invocation Parameters

If some of the parameters are actually lists of values, these values can be distributed in various ways through method invocations to the server interfaces connected to a multicast interface. The default behavior - named *broadcast* - is to send the same parameters to each of the connected server interfaces. However, similar to what SPMD programming offers, it may be adequate to strip some of the parameters so that the bound components will work on different data - named *scatter*.

Which Parameters to Distribute? The first question is where to specify which parameters are to be stripped and distributed. We propose to specify the configuration in a dedicated controller, named CollectiveInterfacesController, and we also need an extension of the type system of Fractal interfaces:

```
interface InterfaceType extends Type {
   String getFcItfName ();
   String getFcItfSignature ();
   boolean isFcClientItf ();
   boolean isFcOptionalItf ();
   boolean isFcCollectionItf ();
   String getFcItfCardinality ();
}
```

The type of an interface 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 cardinality parameter:

```
interface TypeFactory {
   InterfaceType createFcItfType (
      String name,
      String signature,
      boolean isClient,
      boolean isOptional,
      boolean isCollection,
      String cardinality
      ) ...
}
```

The type of a multicast client interface of signature `I`, named `multicastItf`, is defined as follows:

```
InterfaceType itfType = typeFactory.createFcItfType(
        "multicastItf",
        I.class.getName(),
        TypeFactory.CLIENT,
        TypeFactory.MANDATORY,
        TypeFactory.MULTICAST
    )
```

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 Distribute Parameters? The second question is how to specify the distribution of the parameters into the invocations that are generated and forwarded. In the broadcast mode, all parameters are sent without transformation to each receiver. We suppose here that, in the case of the scatter mode, the scattered parameter is of type `list of T` on the server side, and of type `T` on the client side of the multicast interface. 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. We propose to define, as part of the distribution policy, the multiset[3] F where each element $f_j \in F$ is such that, $f_j \subseteq [1..k_1] \times [1..k_2] \times .. \times [1..k_n]$ of the combination of parameters, where F is the (multi)set of the combinations of parameters, 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.

Depending on the type of the client side, we have some restriction on F: if on the server side, the parameter number i is of type $List < T_i >$, and on the client side, the parameter number i is of type T_i, then for each $f_j \in F$, the projection of f_j on the i^{th} dimension must be a singleton.

F may depend on the number of bound components. This multiset allows the expression of all the possible distributions of scattered parameters, including broadcast, 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.

7.1.4 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 d, which knowing s the number of server interfaces bound to the multicast interface, 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

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

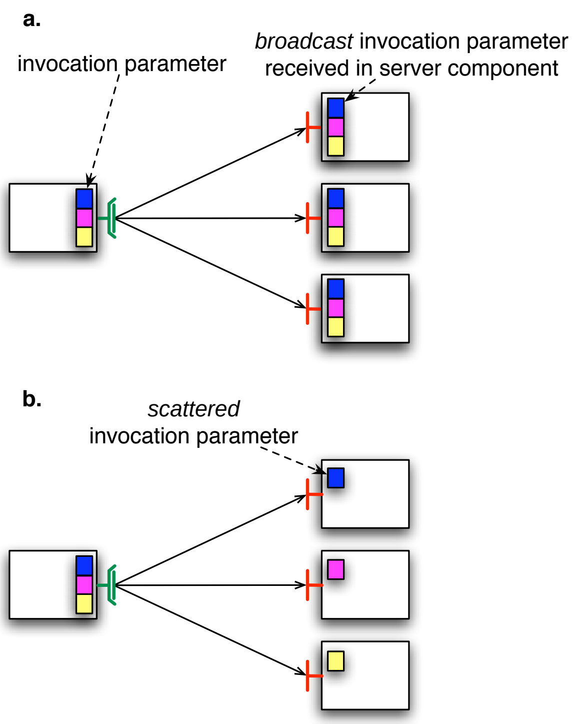
a.
invocation parameter
broadcast invocation parameter received in server component
b.
scattered invocation parameter

Figure 5: Broadcast and scatter of invocations parameters

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.

7.2 Gathercast interfaces

Gathercast interfaces provide abstractions for many-to-one communications. Gathercast and multicast interface definitions and behaviors are symmetrical [3].

7.2.1 Definition

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

A gathercast interface coordinates incoming invocations before continuing the invocation flow: it may define synchronization barriers and may gather incoming data. Return values are redistributed to the invoking components.

Gathercast server interfaces gather invocations from multiple client interfaces (Fig. 6), but client interfaces can also have a gathercast cardinality. A gathercast client interface transforms gathercast invocations (gathering and synchronization operations) into a single invocation which is transfered to the bound server interface. A client gathercast interface also indicates that invocations coming from this client interface contain gathered parameters (lists). In primitive components, the purpose of a gathercast client interface is solely to expose the gathercast nature of this interface. These considerations are summed-up in the following definitions, which characterize external interfaces:

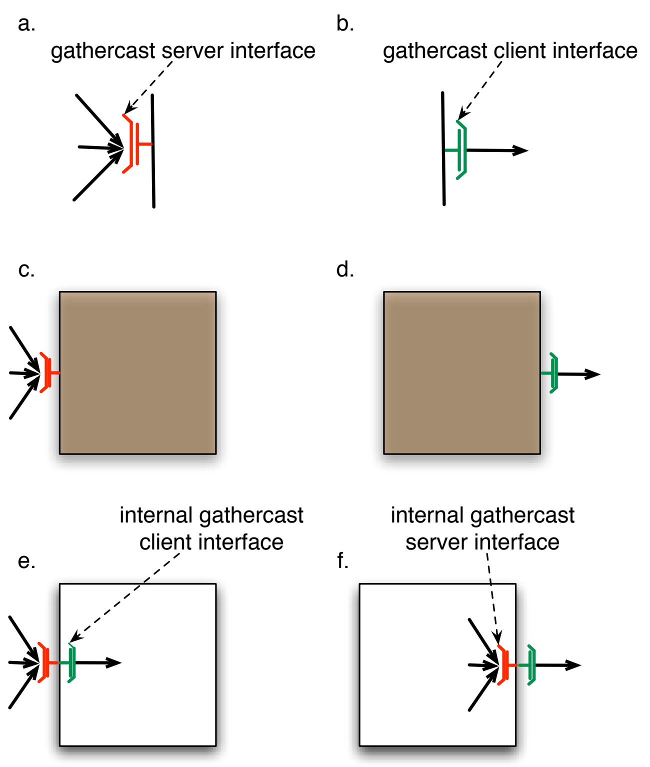
a.
gathercast server interface
b.
gathercast client interface
c.
d.
internal gathercast client interface
e.
internal gathercast server interface
f.

Figure 6: Gathercast server interfaces for primitive and composite components

> A *gathercast client interface* transforms a set of invocations coming from client interfaces of inner components (Fig. 6.f) or from the implementation code of the component (Fig.6.e), into a single invocation.

> A *gathercast server interface* transforms a set of invocations coming from server interfaces of external components into a single invocation to one server interface of an inner component (Fig. 6.d), or to the implementation code in case of a primitive component (Fig. 6.c).

Gathering operations require knowledge of the participants (i.e. the clients of the gathercast interface) in the collective communication. As a consequence, in the context of gathercast interfaces, we have explicitly to state that *bindings to gathercast interfaces are bidirectional links*, in other words: a gathercast interface is aware of which interfaces are bound to it.

7.2.2 Synchronization operations

Gathercast interfaces provide one type of synchronization operation, namely message-based synchronization capabilities: the message flow can be blocked upon user-defined message-based conditions. *Synchronization barriers* can be set on specified invocations, for instance the gathercast interface may wait - with a possible time-out - 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 synchronizations, based on the content of the messages or based on temporal conditions, and it is possible to combine these different kinds of synchronizations.

7.2.3 Gathering of parameters

The gathercast interface aggregates parameters from method invocations. Thus the parameters of an invocation coming from a gathercast (client) interface are actually lists of parameters (Fig. 7). Symmetrically with the case of multicast results, gathered parameters may be reduced, relaxing the constraint of having lists as parameters.

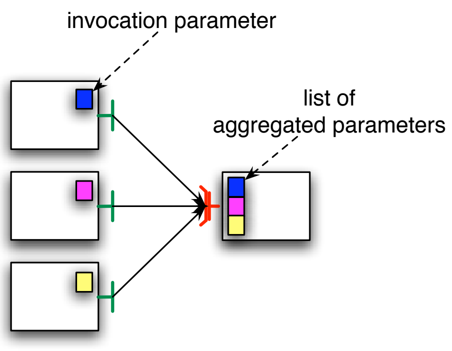

Figure 7: Aggregation of invocation parameters for a gathercast interface

7.2.4 Redistribution of results

The result of the invocation may be a simple result, or a list of results, in which case a redistribution of the enclosed values may occur. The distribution of results for gathercast interfaces is symmetrical with the distribution of parameters for multicast interfaces, and raises the question: where and how to specify the redistribution? The dispatch of the results is not problematic, as it is already given from the binding configuration: each component participating to the gather operation receives a single result.

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, specified either through annotations or in the type of the interface.

The way redistribution 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 a redistribution function can be defined. Otherwise, results should be broadcast to all of the invokers. The redistribution function f is defined as part of the distribution policy of the gathercast interface, it is configurable through its collective interface controller.

7.3 Typing for Collective Interfaces

7.3.1 Parameter Distribution and Aggregation

This section releases the constraints on compatibility between types on the multicast, and on the gathercast interfaces.

Multicast interfaces

We take first the example of the multicast interface.

The constraint of having lists as results for multicast invocations may be relaxed by providing an aggregation mechanism that performs a reduction. Until now,

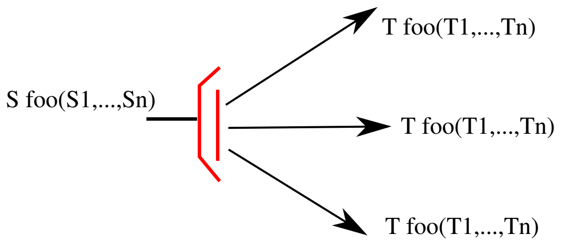

Figure 8: General case of type conversion through a multicast interface

we have defined a basic aggregation function which is concatenation, but more generally, any function can be used for aggregating results. However, depending on this function, the following constraint on interface types must be verified:

> If the returned type of the multicast interface is of type `S`, on the server side, and of type `T`, on the client side then the multicast interface should be configured (and reconfigured) with an aggregation function of type:
>
> `List<T>` → `S`

Section 7.1.2 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 is always possible, and sometimes mandatory, to perform a synchronization of the achievement of the different calls dispatched by the multicast operation.

This generalization could also be applied to the distribution of invocation parameters. In Section 7.1.3, we specified a way to define any kind of "scatter-like" redistribution based on the transformation of a list into subsets of its parameters. In Section 7.1.3, 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>` (argument scattered among the bound interfaces). More generally, we can have any transformation of arguments 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 server side, and of type `Ti`, $1 \leq i \leq n$ on the client side then the multicast interface should be configured (and reconfigured) with a distribution function of type:
>
> `S1..Sn` → Multiset of `T1..Tn`

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

- The result has to be 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 checks that all the results of the n calls distributed to the destination components are identical, or takes the result returned by the majority:

> n results are reduced to a single one; the signature of the aggregation function is: `List<T>` $\rightarrow$ `T`. The multicast interface has the return type: `T`.

- The multicast interface takes the result returned by the majority and also returns the number of clients that have returned this value.

> n results are reduced to a single one plus an occurrence count. The signature of the aggregation function becomes: `List<T>` $\rightarrow$ (`T`*mathttint*). The multicast interface has the return type: (`T`*mathttint*).

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

> The signature of the aggregation function is: `Array<A>` $\rightarrow$ `Array<A>`. The multicast interface has the return type: `Array<A>`.

Similarly, 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 7.1.3):

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

> 1 call with parameter of type `Array<A>` becomes N calls with parameter of type `Array<A>` containing pieces of the original array.

- Scatter:

> 1 call with parameter of type `List<A>` becomes $|\texttt{List} < \texttt{A} >|$ calls with parameter of type `A`

- Broadcast to N destination components in parallel:

> 1 call with parameter of type `A` becomes N calls with parameter of type `A`

Gathercast interfaces
The symmetric 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.

The mechanisms we have presented for multicast and gathercast could easily be integrated in the specification of collective interfaces. The constraints specified in this section should be used when type checking the composition.

7.3.2 Sequential and Collective Interface Typing

From a composition point of view, the parallel nature of a component is an implementation issue. Indeed, composition is based on the functional description and the functional type of components.

In order to allow for compatibility and interchangeability between parallel and sequential component, we propose to define a type compatibility relation between collective and sequential bindings, this is graphically shown in Figure 9.

The principle is that, provided the external type is the same, a collective interface can be used instead of a sequential one, but not the opposite. Indeed the existence of a collective interface implies the existence of a collective interface *controller* for such an interface, and ensures that computation can be performed in parallel.

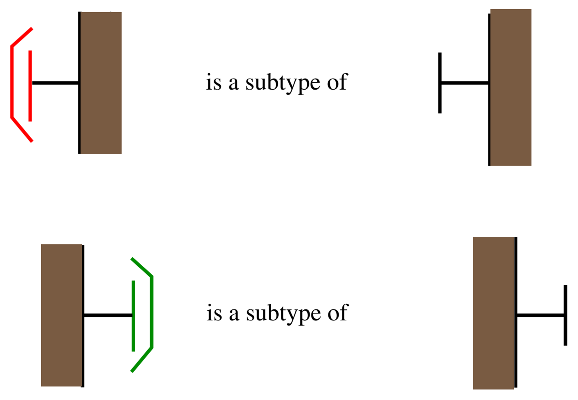

Figure 9: Type compatibility between collective and sequential interfaces

In any case, with such a type compatibility, the GCM allows the transparent use of parallel instead of sequential components, while only the sequential interface is required. Moreover this solution still allows to benefit from the collective nature of the interfaces *only when it is necessary*.

The sub-typing relation defined in the Fractal specification must be extended as follows in order to take into account the new *cardinalities*, multicast and gathercast:

> An interface type I1 is a sub type of a server interface type I2 if the following conditions are satisfied: I1 has the same name and the same role as I2; the language interface corresponding to I1 is a sub interface of the language interface corresponding to I2; if the contingency of I2 is mandatory, then the contingency of I1 is mandatory too; if the cardinality of I2 is collection, then the cardinality of I1 is collection too, *if the cardinality of I2 is multicast, then the cardinality of I1 is multicast too.*
>
> An interface type I1 is a sub type of a client interface type I2 if the following conditions are satisfied: I1 has the same name and the same role as I2; the language interface corresponding to I1 is a super interface of the language interface corresponding to I2; if the contingency of I2 is optional, then the contingency of I1 is optional too; if the cardinality of I2 is collection, then the cardinality of I1 is collection too, *if the cardinality of I2 is gathercast, then the cardinality of I1 is gathercast too.*

7.4 Gathercast to Multicast: the MxN case

The support of parallel components raises the concern of efficient communications between such codes. The MxN problem refers to the problem of efficiently communicating and exchanging data between parallel programs, for instance from a parallel program that contains M processes, to another parallel program that contains N processes. Such communications can be straightforwardly realized by binding a gathercast client interface to a multicast server interface.

Efficient communications between two parallel components requires direct binding and communication between the involved inner components on both sides; this mechanism is called MxN communications. End users expect to have MxN communications so as to provide performance scalability with the parallelism degree. This section proposes a general mechanism for GCM to support SPMD codes and MxN communications.

7.4.1 Principles

In the GCM binding together parallel components consist in binding together collective interfaces; the MxN case corresponds to bind a gathercast interface to a multicast interface. Thus, the naive and not optimized solution is shown in Figure 10. 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, data is redistributed by the multicast interface.

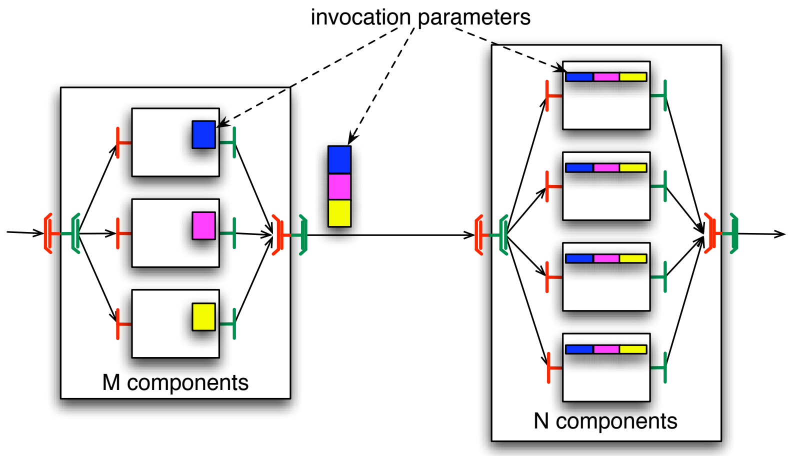

Figure 10: Gathercast to Multicast

Obviously, this naive solution creates a bottleneck both in the gathercast and in the multicast interfaces. Efficient communications requires some forms of direct bindings between the inner components according to the redistribution pattern. Figure 11 shows the general pattern implementing direct communications between the inner components. A couple gathercast-multicast interface is replaced by M multicast interfaces plus N gathercast interfaces. Each inner component on the left hand side is responsible for sending its own data; on the right hand side, each inner component is responsible for gathering the messages it receives, and performing its piece of the global synchronization. This redistribution of collective interfaces avoids the bottleneck occurring in the single gathercast or multicast interface.

Additionally to the data redistribution, the gathercast-multicast composition plays a synchronization role. Indeed, the computation can only start on the right hand side when all the inner components in the left hand side have sent their output. *A priori*, such a synchronization can only be implemented if the MxN direct pattern is all-to-all. Indeed, in the not-optimized version, the gathercast on the left hand side ensures that all inner components have sent their messages. The introduction of the gathercast interfaces in the right hand side moves this synchronization behaviour to the N inner components. Consequently, performing the same synchronization as the not optimized version should require all the clients to send a message to all the gathercast interfaces, some of these messages being only synchronization signals.

However, this global synchronization is not required by all the applications, and in this case more optimization is possible, as all the M inner components do not have to be connected with all the N ones. For example, if only data redistribution is important, then only bindings for transmitting data must be carried out.

The objective of the following of this section is to show, using examples, how such an optimization can be implemented in some classic scenarios.

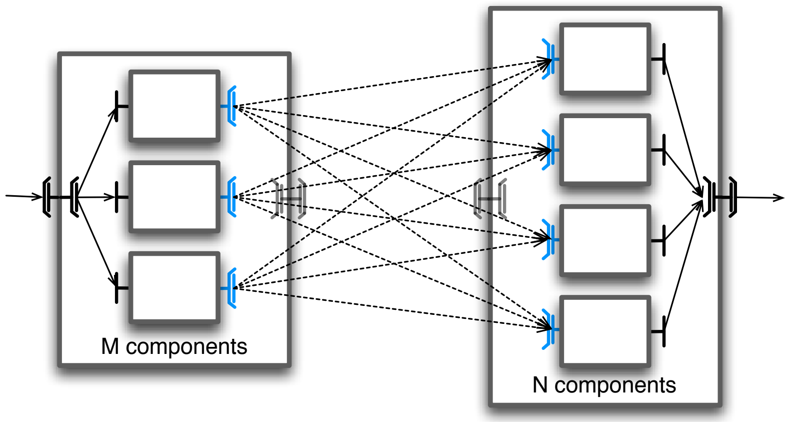

Figure 11: MxN general pattern: all-to-all

7.4.2 Example of a Direct Binding

Let us take the following assumptions:

- We consider parallel components exchanging a single parameter
- A composite component CM, with M inner components, each of this inner component sends an output of size d_i with $0 \leq i < M$
- A composite component CN, with N inner components, each of this inner component takes an input of size d'_j with $0 \leq j < N$
- M inner components have a client interface bound to a Gathercast inner interface of CM, which corresponds to a client interface bound to a server interface of CN, itself corresponding to a Multicast interface scattering the parameters by block to its N inner components,
- 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.

For the sake of simplicity, we add the following hypothesis and notations:

- The size of the global data produced by CM is D,
- each of the M inner component produces the same size of data (d): $d_i = D\ div\ M = d$, (modulo for the last one),
- each of the N inner component takes the same size of data (d'): $d'_j = D\ div\ N = d'$, (modulo for the last one),
- $D > M$ and $D > N$.

Fragments in each inner component First, let us identify the fragments and ranks sent by each inner components in CM, and received by each inner components in CN.

$$Mi \text{ produces } [d * i, d * (i+1)[$$
$$Nj \text{ consumes } [d' * j, d' * (j+1)[$$

For instance, inner Cp M0 receives elements [0,k] from the complete original `list<A>` that arrives upon the call `foo(List<A>)` on the composite M server interface.

Bindings to be realized Each of the M_i components will conceptually have its client interface turned into a client multicast interface with the same signature. Symmetrically, each of the N_i components will have its server interface turned into a gathercast interface.

Let us call MM_i the Multicast interface of component M_i, and NN_j the Gathercast interface of component N_j.

The direct bindings that must occur are defined as follows:

$$MM_i \text{ is to be bound to } NN_j \text{ iff } \exists l \in [d*i, d*(i+1)[, \text{such that } l \in [d'*j, d'*(j+1)[$$

Communications to be performed We define now what elements are directly sent from one inner component of CM to the inner components of CN.

We call c the cardinality of each MM_i, obtained from the definition above: $M/N - 1 \approx d/d' - 1 \leq c < d/d' \approx M/N$. Let r range over the c bindings of the MM_i interface, $0 \leq r < c$, we denote $b(r)$ the rank (between 0 and $N-1$) of the r^{th} gathercast interface of CN to which MM_i is connected. We can now define the elements directly send between two inner components.

When a multicast interface MM_i has been connected to a gathercast interface $NN_{b(r)}$, amongst the elements produced by MM_i, and locally indexed $0..d$, the element sent to the gathercast interfaces are defined as followed:

$$NN_{b(r)} \text{receives from} MM_i \text{elements indexed in} [d' * r, d' * (r+1)[$$

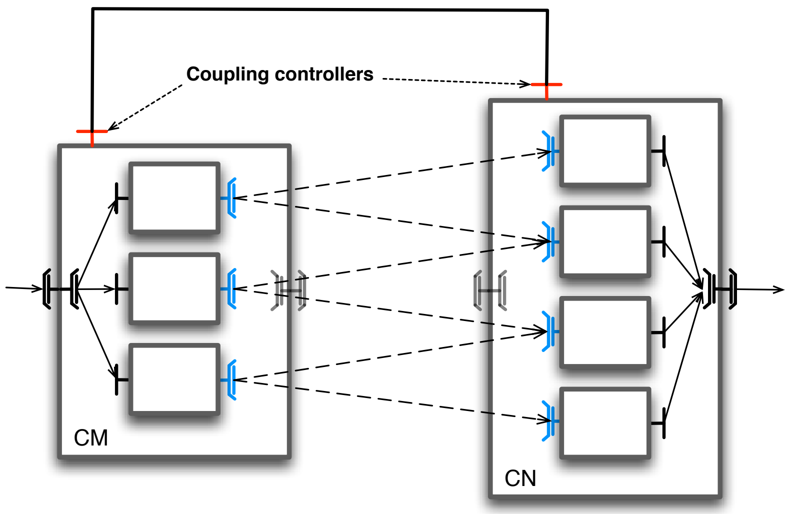

Figure 12: Communications resulting from an MxN direct binding

This example focuses on the case when only one parameter is to be redistributed, the most general case is more complex.

7.4.3 Using Controllers to set up MxN Bindings

This section defines a possible configuration phase for coupling two parallel components, in a MxN manner. It relies on the existence of controllers both at the level of parallel component (Figure 12) and at the level of the inner ones.

When binding two parallel components, both collective 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 collective controllers of the inner components so as to configure them correctly.

Once configured, the communication (data redistribution and synchronization) is straightforward. As every inner component is directly aware of the components it communicates with, as well as relevant data distribution information, a direct communication may occur. This allows one to implement efficiently any data redistribution or synchronization pattern.

This controller-based approach is suitable to implement the redistribution described in the example above (Section 7.4.2). In this case, controllers just have to exchange the cardinality of their respective interfaces (M and N), the size of data to be distributed (D), and the references to the inner components. Controllers only have to create and configure interfaces in the inner components accordingly to the formulas given in the preceding example.

7.5 Summary: Collective Interfaces and Collective Controllers

Overall, this section presented a proposal for introducing *collective* communications in the Fractal model, specified the behavior of the collective interfaces and their controllers. For example, a multicast interface allows any number of clients to be plugged dynamically to a single server and to consider the collective interface as a whole. In contrast, a collection interface implies a set of one-to-one bindings, and is represented by several distinct interfaces at runtime. A multicast interface is an interface that can be bound to several components at the same time and which dispatches messages, it has an associated collective controller allowing its behavior to be specified (distribution policy). A multicast interface is a classical Fractal interface, but with an associated collective controller. Symmetric arguments apply to gathercast interfaces which synchronize several invocations.

The collective interfaces together with their associated controllers constitute a framework for direct communications in the case of the MxN pattern. Such efficient communications also support any kind data redistribution or synchronization patterns thanks to the controllers.

8 Dynamic Controllers

8.1 Principles

Figure 13 shows the representation we suggest for implementing component controllers. As suggested as a classical extension of Fractal, it uses components as controllers, we detail in the following the usage and the necessity of these *component controllers* in the GCM. In the figure, we consider the example of a reconfiguration manager providing an abstraction for managing the reconfiguration of the components, thus avoiding triggering actions directly on the `BindingController` and `ContentController`. A reconfiguration manager is considered as being part of the execution platform; it decides "globally" when and which reconfigurations have to be performed. Then, dynamically, a reconfiguration controller can be added to each reconfigurable component. Such a controller component provides some high-level reconfiguration features. It receives requests/messages from the reconfiguration manager, but also sends information (e.g., about the actual topology of the subcomponent system). This approach allows a better adaptivity, both with respect to the platform – it is easier to upgrade the reconfiguration manager and with respect to the controlled components – one can add a new controller, or dynamically bind to the manager a component to be managed.

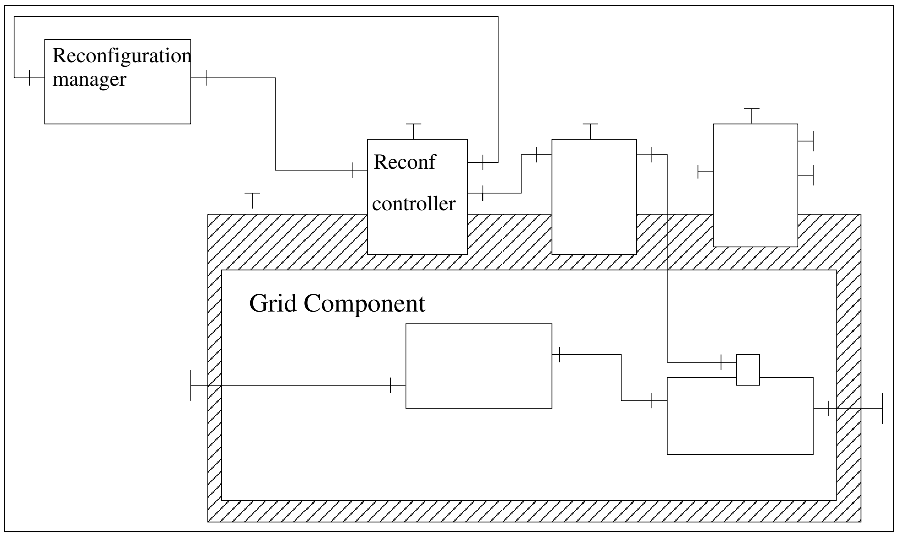

Figure 13: A composite with pluggable controllers

Dynamic controllers specify non-functional adaptive features for the GCM, Therefore, we presuppose in the following that the functional aspects of the considered components are adaptive. Functional adaptivity that is specified by Fractal specification: all levels supporting binding and content controller are adaptive at the functional level (Fractal conformance levels 0.1, 1.1, 2.1, 3.1 and above – denoted ?.1 in the following)

The Fractal specification suggests to refine the general component structure (a content plus a membrane), by specifying that the component's controller can, like the component's content, contain sub-components. Such a Fractal component can then provide new control interfaces to introspect and reconfigure the sub-components of its controller part.

As suggested in the Fractal specification, we propose to provide the possibility to consider a controller as a sub-component, which can then be added, plugged or unplugged dynamically. However, such *controller components* do not need to have all the functionality of a Grid component: a "simple" Fractal component is sufficient: such a component can be considered as a primitive component and only needs to provide the basic controllers (e.g. just the ability to be stopped, and to be bound-/unbound); and needs only to conform to the level 0.1 of the Fractal specification; however, in this case, only the binding and life-cycle controllers are mandatory. In other words, a controller component can be a primitive component without any configurable attribute. As a consequence, when implementing controller components, one can choose to use a simple object, that would provide the required functionality. Of course, implementing such controllers with an independent (possibly distributed) component is also possible.

In Fractal, this extension requires that, when instantiating a component (e.g. through a Factory component), one must be able to specify, as part of the controller description (`ControllerDesc`), a controller taking the form of a "controller component".

This approach is somewhat similar to previous work on aspect-oriented programming for Fractal [7, 12, 22]. Moreover, the current reference implementation of Fractal (Julia) already uses the AOKell framework for allowing a componentized definition of the membrane.

Such a component structure allows a more structured organization of the components:

- controllers can now have server and client interfaces
- Controllers can be managed and reconfigured in a hierarchical way

For example, according to [24], the solution implemented in AOKell is to provide a component-oriented approach for implementing membranes:

- each control membrane is associated to a composite component which exports the control interfaces provided by this membrane,
- each controller is programmed as a component and is inserted into the previously defined composite component,
- controllers are bound together depending on the relations deduced from their semantics[4].

This raises a technical question regarding Fractal and the model proposed here: "What is the lifecycle of component controllers?"

In Fractal, invocation on controller interfaces *must* be enabled when a component is *stopped*, but for changing the bindings of a component, this component must be *stopped*. In other words, in Fractal, the controller of a component is never stopped. If we strictly applied this strategy, we could never reconfigure the component controllers.

The solution we propose consists in having a more complex life-cycle for a composite component with component controllers, allowing to separate partially the life-cycle states of the controller 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 lifecycle controllers), and the other components in the membrane can be reconfigured.

8.2 Application to Reconfiguration

The implementation of controller components that trigger actions of the binding controller and the content controller (e.g., the reconfiguration controller of the example), or that directly implement binding and content control, provides a step towards higher-order components. Indeed, such pluggable controllers might receive components on their ports in order to add and bind them to the component they control, thus achieving a high level of dynamicity and adaptivity. Note that this is not sufficient for bringing a real higher-order level[5] to components since only some *controller* components can receive components, and components can only be manipulated at the non-functional level.

This raises the question: "When/why the component topology should be changed?". Several answers are possible and should coexist. First, some meta-data (attached to the components) can be used directly by the reconfiguration controller, directly triggering actions of the binding and content controllers. Second, a reconfiguration manager (as in the example above) can implement a given policy, and be parameterized with different policy, and of course some meta-informations exploited, in this case, directly by the manager. Finally several components of the platform could be dedicated to implement a very general and powerful reconfiguration policy.

[4]the semantics refers here to the one of aspects and to their composition.

[5]In general, higher-order means that the objects of the language (the components here) can be manipulated in the language itself (anywhere in the components here).

8.3 Application to HOCs

HOC stands for Higher Order Components, a framework that is being developed by the University of Münster.

Dynamic code loading in the context of skeleton programming allows the configuration of components by using entities, given dynamically as parameters when invoking a service on a component. In the HOC system for instance, `setWorker(Worker w)` or `setMaster(Master m)` like services are available for configuring a farm skeleton with some specific instances of Worker or Master sub-classes [10, 9]. On the Grid, such a functionality requires the ability to carry code and transmit it through some component bindings.

A way to implement such HOC framework is to have a `HOCfarmcontroller` that has an interface with two methods: `setWorker(Worker w)` or `setMaster(Master m)`.

When such an interface is invoked, a master or worker component is passed to the a controller, this controller triggers the `addFcSubComponent` method of the content controller in order to instantiate in the composite the adequate master / worker component, and performs the adequate bindings, by invoking the binding controller. In the case of a `setWorker` invocation, several workers can be instantiated depending on the parallelization degree.

These interfaces can also be triggered after the first instantiation in order to perform reconfiguration of the farm: this involves more complex operations entailing removal of old components and their replacement and binding changes, all those operations can be performed via the Fractal's binding and content controllers.

8.4 Summary: Component Controllers as a Fractal Extension

From the ADL and API point of view such an extension only necessitates guaranteeing that a controller descriptor indicating the controllers to be instantiated in a component can be specified in the ADL as well as when instantiating a component (e.g., using a *Factory*). Moreover, when a controller descriptor is required, the descriptor must be able to refer to a *controller component.* A *controller component* is an entity that only needs to conform to level 0.1 of the Fractal specification (mainly a binding and a life-cycle controller), and thus can be implemented in a lightweight manner, for example, by a particular kind of object.

It seems also reasonable, as GCM components are distributed to co-allocate controller components with the composite component for efficiency. But, to keep generality of the model, the GCM does not specify this efficiency issue as mandatory. Indeed, controller components can be very convenient to express remotely accessible non-functional services.

To sum up, a controller component is a standard GCM component, meaning that in theory it can provide any functionality of GCM components, but in practice, a given implementation can give any limit to the conformance level that can be encapsulated in the membrane. Limiting the conformance level to 0 would be very close to Fractal's controllers.

Moreover, the content controller should be extended in order to allow dynamic addition/removal of a controller component to the membrane. This entails extending the Fractal API. This can be either realized by an additional controller or by extending the content controller, or by permitting to gain access to some kind of non-functional membrane, thus allowing addition or removal of sub-components to it thanks to its content controller.

Nevertheless, this is not enough as it does not allow to add/remove ports to the controller component (which would require to implement a dynamic type system at least into controller components' controller). We thus propose to extend the Fractal type system in this way in high conformance levels of the GCM.

We do not require that Fractal's controllers are implemented as components themselves: the only requirement is to be able to instantiate user-defined controllers as components. In other words, the only requirement is for the component model to *accept user defined controllers defined by components*. Note that this neither requires that user-defined controllers are given on the form of components, nor that predefined controllers can be handled on the form of components.

9 Autonomic Components

9.1 Introduction

To achieve better adaptivity, we refine the definition of the component for *Autonomic Computing* (AC) [13, 27]. The term is emblematic of a vast hierarchy of natural self-governing systems, many of which consist of many interacting, self-governing components that in turn comprise a number of interacting, self-governing components at the next level down. Autonomic computing aims to attack the complexity, which entangles the management of complex systems (as Grid applications are) by equipping their parts with self-managements facilities [17, 2]. AC tries to tackle the problem with the often-quoted five "selves": self-configuration, self-healing, self-optimization, self-protection, and, as a combination of all, self-management. As shown in Fig. 14, an autonomic element will typically consist of one or more managed elements coupled with a single autonomic manager that controls them. The managed element could be a hardware resource (storage, CPU, etc.), or a software resource, such as a Web service, or a software *component*. In particular, self-management capabilities can be considered as *aspects* of a component enforcing yet more the separation between its functional and non-functional behaviour:

- Self-configuration: a component is self-configuring if it is able to handle reconfiguration inside itself; this aspect should provide reconfiguration of higher abstraction level than binding or control controllers. Self-configuration can be used to change the component structure for fulfilling some task requested by external clients. This feature might consist in triggering the adequate actions on the binding, content and attribute controllers.

- Self-healing: A component is self-healing if it is able to provide its services in spite of failures of any kind. Components can fail because of implementation and programming errors, or because of hardware faults. In general, faults originated from sub-components should be managed by their container.

- Self-optimization: a component is self-optimizing if it adapts its configuration and structure in order to achieve the best/required performance. For instance, this can be achieved by exploiting cost models and monitoring of the resources which the component is using.

- Self-protection: A component is self-protecting if it is able to predict, prevent, detect and identify attacks, and to protect itself against them. We consider it also interesting in order to design a framework in which some components explicitly deal with possibly malicious (sub)components, autonomously enforcing correct interaction protocols and other self-protection policies.

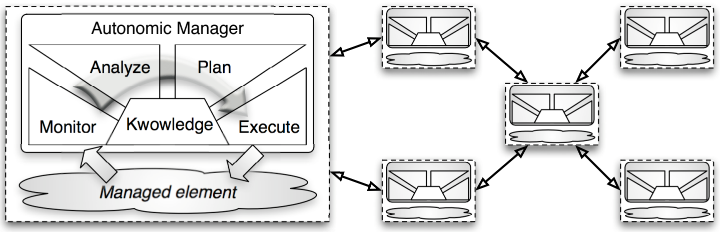

Figure 14: Structure of an autonomic element. Elements interact with other elements and with human programmers via their autonomic managers.

Since these aspects concern non-functional behaviour of the components and their orchestration, they can be naturally implemented within the controllers of components. The interfaces of components enabling to query and steer non-functional aspects behaviour are therefore bound to (possibly distinct) *autonomic controllers*, being them controllers pursuing an autonomic management activity. If a component does not provide one or more of the autonomic functionality, then the implementations of the respective interfaces are absent.

9.2 Autonomic Controllers

As sketched in Fig. 14, autonomic management of a single component leverages on the possibility to introspect (monitor) and intercede (execute) on the current behaviour of component itself, i.e. on both the content and the controllers. Since the content may recursively consists on a graph of GCM components, it is convenient to proceed inductively. In this section we describe the inductive and the ground step, while the big picture of application autonomic management is sketched in Section 9.3.

Let us consider a single composite component. We envision a spectrum of different degrees of autonomicity a component may exhibit. These match different degrees of abstraction of the information flow the component can deal with through its non-functional interfaces, and different conformance levels to the GCM specification.

The bottom level of this range is represented by components having no autonomic control at all; they do not have any additional non-functional interface devoted to QoS management (GCM levels 0–1 in Fig. 2, page 14).

At the next level lie components exhibiting a passive behaviour: they have (non-functional) server interfaces only, for both introspection and intercession. In the former case, they expose information about components current status (e.g. performance and security metrics, configuration metrics as parallelism degree, etc.). In the latter case, they are actually steering interfaces that may trigger content reconfiguration or component status change (e.g. change parallelism degree, content mapping, content orchestration and bindings, information encryption, etc). Somehow, this component is fully or partially equipped with mechanisms enabling dynamic management but it is not provided with strategic skills for self-management. Those two autonomicity levels are part of level 2.d presented in Fig. 2, page 14.

The top level in the range consists of the fully autonomic components. They exhibit self-management skills with respect to all or some of the aspects described in the previous section (self-*). A fully autonomic component exhibits an active approach to behaviour control: they expose a non-functional *server* port accepting (one or more) QoS contracts specifying component required QoS and possibly all the strategic plans needed to achieve it, and a non-functional *client* port providing events to the next level up of the component hierarchy about a possible violation of the QoS contract.

Autonomic controllers of composite components may use both export and import binding to exploit their non-functional activity, as well as all needed interaction with other controllers. In the case of primitive components, these activities are programmed via the component target language.

9.2.1 GCM autonomic conformance levels

In order to make explicit the autonomicity degree of GCM components we propose to extend the Fractal conformance levels specification. As the GCM specification is designed as a Fractal refinement, autonomicity conformance is designed as uniform refinement of the Fractal conformance levels.

The standard Fractal conformance level specification is built on two counters $\Theta.\kappa,\ 0 \leq \Theta, \kappa \leq 3$, where, roughly, $\Theta \geq 1$ means the described "object" exhibit a standard component interface, and $\kappa \geq 1$ means the described "object" exhibits the standard AC, BC, CC, LC, Fractal controllers (Attribute Controller, Binding Controller, Content Controller, and Lifecycle Controller).

This specification can be extended to take in account autonomicity degree by adding an additional counter $\Theta.\kappa.\alpha,\ 0 \leq \Theta, \kappa \leq 3,\ 0 \leq \alpha \leq 3$, where α indicates the level of autonomicity: $\perp$ (viz. NA), no, low-level passive, high-level active autonomicity, respectively. With respect to levels described in the previous section, an additional level $\perp$ is introduced for the sake of uniformity with Fractal levels. It indicates all cases in which the described "object" is not yet a real Fractal component, thus do not exhibit the basic controllers, thus preventing any reasoning about its introspection and intercession capability ($\Theta < 1 \vee \kappa < 1 \Rightarrow \alpha = \perp$).

9.2.2 Level $\Theta.\kappa.1$: No autonomic control

These are GCM components that expose the `Component` interface and include at least all basic controllers (AC, BC, CC, LC), thus they can be bound to membrane and other components (via normal binding). Also, they can be started and stopped. Because of these features, despite that they do not directly provide any autonomic control facility, they can appear as sub-components of composite component with equal or higher level of autonomicity. In the latter case, the outer component should explicitly take in account they cannot either expose their status (e.g. performance or security measure, parallelism degree, fault-tolerance degree), nor be asked to change behaviour in any way.

9.2.3 Level $\Theta.\kappa.2$: Low-level, passive autonomic control

These are GCM components that expose the `Component` interface and include at least all basic controllers (AC, BC, CC, LC), thus they can be bound to membrane and other components. Moreover, these components include one or more `AutonomicController` interfaces (that can be distinguished for their name, which is suffixed by "`-autonomic-controller`" in their list of interfaces provided by the `getFcInterfaces()` operation in the `Component` interface or by implementation-depend procedures.

The `AutonomicController` interface offers the list of methods provided by the component for handling autonomicity. Thus, its structure is:

```
interface AutonomicController {
  String[] listAutonomicOperations();
  any execOperation(String op, any...);
}
```

The `listAutonomicOperations()` method enables the listing of all operation names provided by the controller. Each operation is executed by invoking the `execOperation(String op, any...)`, where the second argument is the (vararg) list of parameters supplied to the `op` operation.

As an example, an `AutonomicController` could provide operations for evaluating service time, latency, bandwidth, getting or increasing/decreasing parallelism degree (in case of parallel components).

```
class SimpleAC implements AutonomicController {
  String toString(){return("SimpleAC-autonomic-controller")};
  String[] listAutonomicOperations(){
    return new String[]{"ServiceTime", "Throughtput",
              "ParDegree", "INC_ParDegree", "DEC_ParDegree"};
  }
  any execOperation(string opt, any... args){
    if(opt.equalsTo)("ServiceTime") return getServiceTime();
    if(opt.equalsTo("Throughtput") return getCurrentThroughtput
        ();
    if(opt.equalsTo("ParDegree") return getParDegree();
    if(opt.equalsTo("INC_ParDegree") return incParDegree(Adapt(
        args));
    if(opt.equalsTo("DEC_ParDegree")) return decParDegree(Adapt
        (args));
  }
  Long getServiceTime() { ... }
  long getCurrentThroughtput{ ... }
  int getParDegree(){ ... }
  int incParDegree(int units){ ... }
  int decParDegree(int units){ ... }
}
```

The operations have the main goal to provide information and support for handling the four aspects of self-management identified above at a low-level of detail. As an example, programmers who want to interact with a given remote component relying on its parallelism degree and service time, could follow these steps:

- detect if the component is adaptive or not by investigating its lists of interfaces provided by `getFcInterfaces()` of the `Component` interface.
- select autonomic controllers from the result of the invocation by using a implementation-dependent procedure (Fractal 1.1) or look for interface whose name is suffixed by "`-autonomic-controller`", provided the programmer has implemented a `getFcItfName()` of interface `fractal.api.Interface` returning a properly suffixed name (Fractal 2.1);
- acquire the list of operations through `listAutonomicOperations()` of each autonomic controller;
- eventually invoke an operation through `execOperation()`.

In the example shown above, the `SimpleAC` component controller offers the possibility to inquire some performance statistics of the service offered by the component (supposing a single service is offered), such as service time, throughput, and to query and modify some internal features, such as the parallelism degree.

We have to point out that the aspects (i.e. the parameters) related to the adaptivity of a component are still the subject of future work. In the future, we will be able to better identify which parameters define an adaptive behavior and, as a consequence, which is the minimal set of operations that the `AutonomicController` interface must provide.

Notice, both introspection and intercession are provided via server interfaces.

9.2.4 Level $\Theta.\kappa.3$: high-level, active autonomic control

These are GCM components that expose the `Component` interface and include at least all basic controllers (AC, BC, CC, LC). In addition to standard features, these components are supposed to provide high-level features for non-functional behaviour management. They are:

- A (server) operation to submit a QoS contract to the component;
- A (client) operation able to signal contract violations.

The former operation takes as input a QoS contract formatted in some structured way (e.g. XML), stating a goal (either qualitative or quantitative) the autonomic controller will try to reach. Also, the contract could include hints or strategies aiming at reaching the goal. These strategies should be constructively described in terms of operations available at the inner level of nesting. The nature of these goals and the suitable tools needed to express them are deliberately left unspecified in order to keep the GCM specification as general as possible.

In this respect, a GCM component compliant to level $\Theta.\kappa.3$ is a fully autonomic component. The only source of information needed by the manager handling autonomicity is a QoS contract, to which it reacts through contract violation or services. A fully autonomic component must implement the two following interfaces:

```
interface AutonomicServerManager {
  any commitContract(String qosContract);
}

interface AutonomicClientManager {
  any raiseViolation(any violationId);
}
```

The suffix "`-ac_manager-controller`" is used to indentify these kind of managers. This also enforces the distinction between passive and active controllers.

The `commitContract` server operation take a QoS contract as input and directs the component behavior in order to adapt itself to the Quality of Service required by the contract parameters. However, a contract violation could occur in any moment of the computation, so that a `raiseViolation()` is signaled. These events represent the inability of the controller in finding a local self-management strategy able to turn the component behaviour within contractually specified constraints (e.g. not enough resources available, too many faults, too strict contract constraints, etc.).

Whether the outermost component manager (a.k.a *Application Manager*) is not able to respect the contract, a system warning is raised toward the user (the Application Manager `raiseViolation()` interface is bound to the application launcher by default).

9.3 Hierarchy and Autonomicity

Applying autonomicity hierarchically is particularly important for a hierarchical component model like the GCM. Indeed, to fulfill an autonomic goal, a component should generally rely on its sub-components, and generate sub-goals that will be delegated to them.

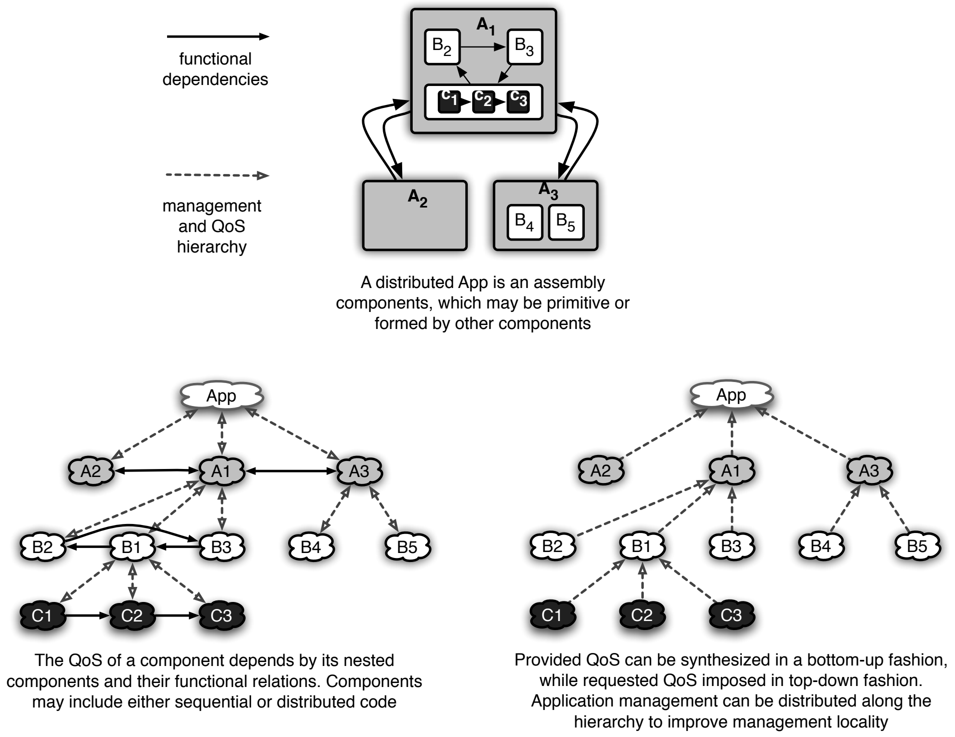
functional dependencies
management and QoS hierarchy
A distributed App is an assembly components, which may be primitive or formed by other components
The QoS of a component depends by its nested components and their functional relations. Components may include either sequential or distributed code
Provided QoS can be synthesized in a bottom-up fashion, while requested QoS imposed in top-down fashion. Application management can be distributed along the hierarchy to improve management locality
App
A1
A2
A3
B1
B2
B3
B4
B5
C1
C2
C3

Figure 15: Hierarchy of components: distribution of QoS constraints and synthesis of current status.

Overall, the hierarchy of components can be naturally described by a tree, where the leaves represent primitive components and the root represents the manager of the whole application. Conceptually, as sketched in Fig. 15, non-functional proprieties modeling run-time behaviour of the whole hierarchy can be synthesized in a bottom-up fashion: the behaviour of a composite component depends on the behaviour of nested components. Management actions (either level $\Theta.\kappa.2$ commands or $\Theta.\kappa.3$ contracts) should be projected along the tree in a top-down fashion: the users usually would like to declare a global goal they expect from an application. This matches the idea of submitting a contract at the root of tree. A fully autonomic system should automatically split the global goal into sub-goals that should be enforced on inner components.

A safe design option may require that all the paths from the root to the leaves along the tree describing component nesting hierarchy are described by a non-increasing succession of autonomic conformance levels. In this way, a controller is never required to interpret a sequence of requests coming from its non-functional interfaces by using operations that are more abstract than requests.

9.4 Summary

Autonomicity is the ability for a component to adapt to situations, without relying on the outside. The less the components rely on the outside the more autonomous they are, and the more abstract are the orders and information the components can receive from the outside the more autonomous they are. In this respect, the levels of autonomicity of a GCM component are arranged in four classes, which smoothly extend standard Fractal conformance levels. A GCM component exhibits autonomic features it implements the GCM `AutonomicController` interface, thus starting from conformance level $\Theta.\kappa.2$, while lower levels are supposed to capture non-adaptive legacy components (including Fractal components). Application management is organized along the hierarchy of GCM components in order to keep management decisions and reconfiguration actions as local as possible.

10 Behaviour Protocol Specification

We propose here an extension to Fractal that allows to specify the behavior of a component. This extension allows us to use different semantic formalisms and different underlying models to reach goals that are similar in nature: provide some guarantees that components in a composite system behave smoothly together, and/or respect some user requirements.

The support for behavioural model is optional in the GCM, but in case such behavioural specification is supported, it should fit the extension of the ADL defined below.

We expect that the extension to the Fractal ADL proposed here can be used together with any of behavioural model, because the choice of the communication model (and the choice of the semantical formalism) is encapsulated within the behaviour specification given in a separate file. In both cases, behavioural specifications are expressed in expressive and well-established languages, that have their own parsers. Both for separation of concern, and for practical reasons, it would not have been reasonable to incorporate these descriptions within the ADL syntax.

10.1 Proposed Addition to the ADL

Our proposal consists in adding the following elements to the existing ADL formalism. These elements seem to be open enough for any formalism to be integrated easily if needed.

Only minor changes to the ADL are needed, as we propose that every implementation-specific detail be treated by a specialized parser as they are not related to the component architecture. The elements and attributes needed to comply with the above requirements are summarized as:

An optional `behavior` element, that contains behavioural specifications. The element can be nested inside the `component`, `definition`, `interface` or `binding` element. The `behavior` element will have the following attributes:

- A `language` attribute, specifying the behaviour specification language. Currently it can have values `fc2`, `lotos`, or `behavior-protocol`.
- A `file` attribute of type String, used to designate a file containing the behaviour specification. The `language` attribute will be used to select the adequate parser for this file.
- A `value` attribute of type String, when the behaviour specification is provided inline in the ADL file. Again the `language` attribute will be used to select the adequate parser for this string. Only one of the attributes `file` or `value` should be specified.

Examples:

```
<component name="Phone">
  <interface ....>
  <content class="Phone"/>
  <controller desc="primitive"/>
```

```
    <behavior language="lotos"
              file="Phone.lotos">
  </component>
```

```
 <component name="IFirewall">
   <interface name="IFirewall" role="server"/>
   <content class="FirewallImpl"/>
   <controller desc="primitive"/>
   <behavior language="behavior-protocol"
          value="?IFirewall.Enable* | ?IFirewall.Disable*">
 </component>
```

As we want a generic extension of Fractal, we have decided to avoid extensions of the ADL dtd that would carry information specific to one approach or another. Instead, we group additional information together with the behaviour specification in a separate document, which can be, e.g., again an XML document, with syntax (DTD) determined by the specific Fractal extension — selected by the `language` attribute.

Extensibility: Adding a new possible behaviour formalism would not require any modification of the ADL DTD: it consists in adding another keyword as a valid value for the `language` attribute, and providing the corresponding parser.

10.2 Related Work and Additional Information

Work at Charles University [1] has shown that while for static and runtime checking, associating a component with a protocol was sufficient, the task of employing a model checker to check the compliance of the implementation of a primitive component with its specification (frame protocol) required additional information, which has been embedded in an additional element, `environment`.

Researchers from the SOFA team use "Behavior Protocols" [23], a notation specifying component behaviour in terms of ordering of method invocation events. A behaviour protocol is an expression composed from basic and advanced operators; its syntax stems from regular expressions. The *behaviour compliance* and *consent* relations are defined on behaviour protocols based on their trace semantics, allowing to reason on substitutability and compositional compatibility. The behaviour of a component is specified by its *frame protocol*. In a composite component, the *architecture protocol*, constructed from frame protocols of its subcomponents, is checked for compliance with the frame protocol. For a primitive component, its Java implementation may be checked for compliance with a model checking tool [19].

Researchers from the OASIS team are using parameterized, hierarchical, networks of labeled transition systems [4], and a bisimulation-based semantics, to specify and verify the behavioural properties of component systems. Finite abstractions of the component system behaviour can be built, taking advantage of the congruence properties of bisimulation theories, and using standard model-checking tools from the CADP toolset [14]. The models are generated from an ADL description and from the associated Java interfaces, and include specific controllers expressing the non-functional aspects of either synchronous (Julia) or asynchronous (ProActive) Fractal implementations [5].

11 Conclusion

This document presented the requested features for the GCM, and the way we propose to implement them, as a set of extensions to the Fractal specification. Among those extensions, most of them, for instance multicast interfaces and autonomic components, will not be mandatory. Depending on the implementation of these non-mandatory features by the component model, we will define several conformance levels for the GCM that will complement the ones defined in Fractal.

As a first conclusion, we would like to point out an illustrative example of what can be done with the GCM can be found in [20]; more precisely, this technical report is focused on the implementation of a Grid application using ProActive components which implements Fractal specification, support for deployment on the Grid, and multicast and gathercast interfaces. Thus a complete implementation of the GCM would moreover provide much more functionalities, including for example support for autonomicity, adaptivity and streaming communications.

To conclude, we summarize the requested features for the GCM, together with the solutions proposed to support them.

Requested feature	Concept in GCM to achieve it
Hierarchical composition	Fractal's component hierarchy
Extensibility	• From Fractal design • dynamic controllers (any non-functional feature can be added to any component) • open and extensible communication mechanisms
Support for reflection (introspection and intercession)	From the Fractal specification and API
Lightweight	• Support for adaptivity in the component model and extensibility both in the component model and the specification • Conformance levels • No controller imposed
Well-defined Semantics	API + ADL + well-defined extensions – via the (future versions of the) current document
ADL with support for deployment	Virtual Nodes + cf Section 3
Packaging	cf. packaging being defined by the Fractal community
Support for Higher-order component programming skeleton-parallel, functional programming	language neutrality + partial support for higher-order via controllers, and especially controller components
Sequential and parallel implementation	XML component specifications, and Multicast-Gathercast interfaces allow plugging and unplugging several components to the same interface dynamically
Asynchronous ports and Extended / Extensible port semantics	Asynchronous Method Invocation as the default semantics but any semantics can be defined via tags; special support for streaming $\Rightarrow$ Possibility to support method calls / message oriented / streaming / still unknown kinds of communication

Group related communication on interfaces	Multicast / Gathercast interfaces
MxN communications	Multicast / Gathercast interfaces (to be refined)
Adaptivity: • Exploit Component Hierarchical abstraction for adaptivity • Ability to plug/unplug components dynamically • Give a standard for adaptive behavior and unanticipated extension of the model • Give a standard for the management autonomic components • Plug/unplug non-functional interfaces	Globally due to dynamic controllers Dynamic controllers Fractal's content and binding controllers Dynamic controllers Autonomic controllers Dynamic controllers
Support for deployment	Notion of virtual nodes ADL with support for deployment
Parallel binding: Well-defined and verifiable composition	Multicast / Gathercast interfaces
Language neutrality	• API in various languages • Various interface specifications to be used (IDL, Java, WSDL, etc.) • "Systematic" exportation of a web-service port
Interoperability	Exportation and importation as web-services

Recall the GCM is planned to be both a model for the application components but also for the system (middleware) level components, allowing the component platform to benefit from the features of the GCM, in particular adaptivity, reconfigurability, separation of concerns, etc. The component controllers presented in Section 8 play a crucial role in this perspective. Future works on the GCM will provide implementation strategies for the GCM.

References

[1] Component reliability extensions for Fractal component model. http://kraken.cs.cas.cz/ft/public/public_index.phtml.

[2] Marco Aldinucci, Francoise André, Jérémy Buisson, Sonia Campa, Massimo Coppola, Marco Danelutto, and Corrado Zoccolo. An abstract schema modeling adaptivity management. In Sergei Gorlatch and Marco Danelutto, editors, *Integrated Research in Grid Computing*, CoreGRID. December 2006.

[3] B. Badrinath and P. Sudame. 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)*, 2000.

[4] T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed java objects. In *Forte'04 conference*, Madrid, 2004. LNCS 3235, Spinger Verlag.

[5] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for distributed components. In *FACS'05 Workshop*, Macao, 2005. LNCS , Spinger Verlag.

[6] Eric Bruneton. Julia tutorial. `http://fractal.objectweb.org/tutorials/julia/index.html`, 2003.

[7] Pierre-Charles David and Thomas Ledoux. Towards a framework for self-adaptive component-based applications. In Jean-Bernard Stefani, Isabelle Demeure, and Daniel Hagimont, editors, *Proceedings of Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6.1 International Conference, DAIS 2003*, volume 2893 of *Lecture Notes in Computer Science*, pages 1–14, Paris, November 2003. Federated Conferences, Springer-Verlag.

[8] Alexandre Denis, Christian Perez, Thierry Priol, and André Ribes. Bringing high performance to the corba component model. In *SIAM Conference on Parallel Processing for Scientific Computing*, 2004.

[9] J. Dünnweber, F. Baude, V. Legrand, N. Parlavantzas, and S. Gorlatch. Towards automatic creation of web services for grid component composition. In Vladimir Getov, Domenico Laforenza, and Alexander Reinefeld, editors, *Integrated Research in Grid Computing*, volume 4 of CoreGRID series. Springer, jan 2006. ISBN: 0-387-27935-0. Also as the CoreGrid TR003 report.

[10] Jan Dünnweber and Sergei Gorlatch. HOC-SA: A Grid Service architecture for Higher-Order Components. In *International Conference on Services Computing, Shanghai, China*. IEEE Computer Society Press, September 2004. ISBN 0-7695-2225-4.

[11] E.Bruneton, T.Coupaye, and J.B. Stefani. The Fractal Component Model `http://fractal.objectweb.org/specification/index.html`. Technical report, ObjectWeb Consortium, February 2004.

[12] H. Fakih, N. Bouraqadi, and L. Duchien. Aspects and software components: A case study of the fractal component model. In *International Workshop on Aspect-Oriented Software Development (WAOSD 2004)*, Beijing, China, September 2004.

[13] A.G. Ganek and T.A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal - Autonomic Computing*, 42(1):5–18, 2003.

[14] 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.

[15] A. Mayer, S. Mcough, M. Gulamali, L. Young, J. Stanton, S. Newhouse, and J. Darlington. Meaning and behaviour in grid oriented components. In *Third International Workshop on Grid Computing, GRID*, volume 2536 of *LNCS*, pages 100–111, 2002.

[16] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[17] Next Generation GRIDs Expert Group. *NGG3, Future for European Grids: GRIDs and Service Oriented Knowledge Utilities. Vision and Research Directions 2010 and Beyond*, January 2006.

[18] 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.

[19] 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. to appear.

[20] Nikos Parlavantzas, Matthieu Morel, Francçoise Baude, Fabrice Huet, Denis Caromel, and Vladimir Getov. Componentising a scientific application for the grid. Technical Report CoreGRID Technical Report, TR-0031, CoreGRID, Institute on Grid Systems, Tools and Environments, Apr 2006.

[21] C. Partridge, T. Menedez, and W. Milliken. Host anycasting service. RFC 1546, 1993.

[22] Nicolas Pessemier, Lionel Seinturier, and Laurence Duchien. Components, adl & aop: Towards a common approach. In Walter Cazzola, Shigeru Chiba, and Gunter Saake, editors, *RAM-SE*, pages 61–69. Fakultät für Informatik, Universität Magdeburg, 2004.

[23] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), nov 2002.

[24] Lionel Seinturier, Nicolas Pessemier, and Thierry Coupaye. AOKell: an Aspect-Oriented Implementation of the Fractal Specifications, 2005. http://www.lifl.fr/ seinturi/aokell/javadoc/overview.html.

[25] Jean-Bernard Stefani. Fractal objectweb project and further fractal developments. http://www.objectweb.org/wws/arc/fractal/2005-04/msg00003.html, 2005.

[26] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[27] S.R. White, J.E. Hanson, I. Whalley, D.M. Chess, and J.O. Kephart. An architectural approach to autonomic computing. In *Proceedings of the International Conference on Autonomic Computing*, pages 2–9. IEEE, May 2004.

A Short List of Important Definitions

Component content: one of the two parts of a component, the other one being its controller. A content is an abstract entity controlled by a controller. The content of a component is (recursively) made of sub components and bindings.

Component controller or **membrane**: one of the two parts of a component, the other one being its content. A controller is an abstract entity that embodies the control behavior associated with a particular component. A controller can exercise an arbitrary control over the content of the component it is part of (intercept incoming and outgoing operation invocations for instance).

Server interface: a component interface that receives operation invocations.

Client interface: a component interface that emits operation invocations.

Functional interface: a component interface that corresponds to a provided or required functionality of a component, as opposed to a control interface.

Control interface: a component interface that manages a "non functional aspect" of a component, such as introspection, configuration or reconfiguration, and so on.

Virtual node: Virtual Nodes are abstractions allowing a clear separation between design infrastructure and physical infrastructure. Virtual Nodes are used in the code or in the ADL and abstract names and creation and connection protocols to physical resources, from which applications remain independent.

Multicast interface: A multicast interface is an interface that is able to transform a single invocation into a list of invocations.

Gathercast interface: A gathercast interface is an interface that is able to transform a set of invocations into a single invocation.

Autonomicity: Autonomicity is the ability for a component to adapt to situations, without relying on the outside.

B Short List of Important Acronyms

AC: Autonomic Computing.
ADL: Architecture Description Language.
CCA: Common Component Architecture.
CCM: Corba Component Model.
GCM: Grid Component Model.
HOC: Higher-Order Components.
PSE: Problem Solving Environments.
RMI: Remote Method Invocation.
SOA: Service Oriented Architectures.
SPMD: Single Program Multiple Data.

C GCM API

C.1 Java API

This is a modified version of Fractal for the sake of being capable of controlling large GCM component configurations in an asynchronous manner:

- exceptions replaced with GCMRuntimeException,
- returned Arrays replaced with parameterized Lists,
- non-reifiable return types encapsulated by parameterized wrapper.

```
package net.coregrid.gcm.api;
import package net.coregrid.gcm.api;.factory.InstantiationException;
public interface Component {
  Type getFcType ();
  List<Interface> getFcInterfaces ();
  Object getFcInterface (String interfaceName) throws NoSuchInterfaceException;
}
public interface Interface {
  Component getFcItfOwner ();
  StringWrapper getFcItfName ();
  Type getFcItfType ();
  boolean isFcInternalItf ();
}
public interface Type {
  boolean isFcSubTypeOf (Type type);
}
public class Fractal {
  public static Component getBootstrapComponent () throws InstantiationException;
}
public class NoSuchInterfaceException extends GCMRuntimeException {
  public NoSuchInterfaceException (String itfName) { super(itfName); }
}
package net.coregrid.gcm.api.control;
import net.coregrid.gcm.api.Component;
import net.coregrid.gcm.api.NoSuchInterfaceException;
public interface AttributeController { }
public interface BindingController {
  List<String> listFc ();
  Object lookupFc (String clientItfName) throws NoSuchInterfaceException;
  void bindFc (String clientItfName, Object serverItf) throws
    NoSuchInterfaceException, IllegalBindingException, IllegalLifeCycleException;
  void unbindFc (String clientItfName) throws
    NoSuchInterfaceException, IllegalBindingException, IllegalLifeCycleException;
}
public interface ContentController {
  List<Interface> getFcInternalInterfaces ();
  Object getFcInternalInterface (String interfaceName) throws NoSuchInterfaceException;
  List<Component> getFcSubComponents ();
  void addFcSubComponent (Component subComponent)
    throws IllegalContentException, IllegalLifeCycleException;
  void removeFcSubComponent (Component subComponent)
    throws IllegalContentException, IllegalLifeCycleException;
}
public interface SuperController {
  List<Component> getFcSuperComponents ();
}
```

```
public interface LifeCycleController {
  Wrapper<String> getFcState ();
  void startFc () throws IllegalLifeCycleException;
  void stopFc () throws IllegalLifeCycleException;
}
public interface NameController {
  Wrapper<String> getFcName ();
  void setFcName (String name);
}

public class InstantiationException extends GCMRuntimeException {
  public InstantiationException (String msg) { super(msg); }
}

public class IllegalBindingException extends GCMRuntimeException {
  public IllegalBindingException (String msg) { super(msg); }
}
public class IllegalContentException extends GCMRuntimeException {
  public IllegalContentException (String msg) { super(msg); }
}
public class IllegalLifeCycleException extends GCMRuntimeException {
  public IllegalLifeCycleException (String msg) { super(msg); }
}
package org.objectweb.fractal.api.factory;
import net.coregrid.gcm.api.Component;
import net.coregrid.gcm.api.Type;
public interface Factory {
  Type getFcInstanceType ();
  Object getFcControllerDesc ();
  Object getFcContentDesc ();
  Component newFcInstance () throws InstantiationException;
}
public interface GenericFactory {
  Component newFcInstance (Type type, Object controllerDesc, Object contentDesc)
    throws InstantiationException;
}
public class InstantiationException extends GCMRuntimeException {
  public InstantiationException (String msg) { super(msg); }
}
package net.coregrid.gcm.api.type;
import net.coregrid.gcm.api.NoSuchInterfaceException;
public interface ComponentType extends net.coregrid.gcm.api.Type {
  List<InterfaceType> getFcInterfaceTypes ();
  InterfaceType getFcInterfaceType (String name) throws NoSuchInterfaceException;
}
public interface InterfaceType extends org.objectweb.fractal.api.Type {
  Wrapper<String> getFcItfName ();
  Wrapper<String> getFcItfSignature ();
  Wrapper<Boolean> isFcClientItf ();
  Wrapper<Boolean> isFcOptionalItf ();
  Wrapper<Boolean> isFcCollectionItf ();
}
public interface TypeFactory {
  InterfaceType createFcItfType (
    String name, String signature, boolean isClient,
    boolean isOptional, boolean isCollection)
      throws org.objectweb.fractal.api.factory.InstantiationException;
  ComponentType createFcType (InterfaceType[] interfaceTypes)
```

```
    throws org.objectweb.fractal.api.factory.InstantiationException;
}

package net.coregrid.gcm.util;
public class Wrapper<T> implements java.io.Serializable {
    private T wrappedObject ;
    public GenericTypeWrapper() {    }
    public GenericTypeWrapper(T o) {wrappedObject  = o;}
    public T getWrappedObject() { return wrappedObject;}
    public boolean equals(Object arg) {
      return ((arg instanceof Wrapper)?((Wrapper)arg).getWrappedObject().equals(wrappedObject):false;
    }
    public int hashCode() {return this.wrappedObject.hashCode();}
}
```

C.2 C API

```
typedef unsigned char jboolean;
typedef unsigned short jchar;
typedef signed char jbyte;
typedef signed short jshort;
typedef signed int jint;
typedef signed long long jlong;
typedef float jfloat;
typedef double jdouble;
struct Morg_objectweb_naming_Name {
  Rorg_objectweb_naming_NamingContext* (*getNamingContext)(void *_this);
  jbyte* (*encode)(void *_this);
};
struct Morg_objectweb_naming_NamingContext {
  Rorg_objectweb_naming_Name* (*export)(void *_this, void* o, void* hints);
  Rorg_objectweb_naming_Name* (*decode)(void *_this, jbyte* b);
};
struct Morg_objectweb_naming_Binder {
  Rorg_objectweb_naming_Name* (*export)(void *_this, void* o, void* hints);
  Rorg_objectweb_naming_Name* (*decode)(void *_this, jbyte* b);
  void* (*bind)(void *_this, Rorg_objectweb_naming_Name* n, void* hints);
};
struct Morg_objectweb_fractal_api_Component {
  Rorg_objectweb_fractal_api_Type* (*getFcType)(void *_this);
  void** (*getFcInterfaces)(void *_this);
  void* (*getFcInterface)(void *_this, char* interfaceName);
};
struct Morg_objectweb_fractal_api_Interface {
  Rorg_objectweb_fractal_api_Component* (*getFcItfOwner)(void *_this);
  char* (*getFcItfName)(void *_this);
  Rorg_objectweb_fractal_api_Type* (*getFcItfType)(void *_this);
  jboolean (*isFcInternalItf)(void *_this);
};
struct Morg_objectweb_fractal_api_Type {
  jboolean (*isFcSubTypeOf)(void *_this, Rorg_objectweb_fractal_api_Type* type);
};
struct Morg_objectweb_fractal_api_control_AttributeController { };
struct Morg_objectweb_fractal_api_control_BindingController {
  char** (*listFc)(void *_this);
  void* (*lookupFc)(void *_this, char* clientItfName);
  void (*bindFc)(void *_this, char* clientItfName, void* serverItf);
  void (*unbindFc)(void *_this, char* clientItfName);
};
struct Morg_objectweb_fractal_api_control_ContentController {
  void** (*getFcInternalInterfaces)(void *_this);
  void* (*getFcInternalInterface)(void *_this, char* interfaceName);
  Rorg_objectweb_fractal_api_Component** (*getFcSubComponents)(void *_this);
  void (*addFcSubComponent)(
    void *_this, Rorg_objectweb_fractal_api_Component* subComponent);
  void (*removeFcSubComponent)(
    void *_this, Rorg_objectweb_fractal_api_Component* subComponent);
};
struct Morg_objectweb_fractal_api_control_SuperController {
  Rorg_objectweb_fractal_api_Component** (*getFcSuperComponents)(void *_this);
};
struct Morg_objectweb_fractal_api_control_LifeCycleController {
  char* (*getFcState)(void *_this);
  void (*startFc)(void *_this);
  void (*stopFc)(void *_this);
};
struct Morg_objectweb_fractal_api_control_NameController {
  char* (*getFcName)(void *_this);
  void (*setFcName)(void *_this, char* name);
};
struct Morg_objectweb_fractal_api_factory_Factory {
  Rorg_objectweb_fractal_api_Type* (*getFcInstanceType)(void *_this);
  void* (*getFcControllerDesc)(void *_this);
  void* (*getFcContentDesc)(void *_this);
  Rorg_objectweb_fractal_api_Component* (*newFcInstance)(void *_this);
};
struct Morg_objectweb_fractal_api_factory_GenericFactory {
  Rorg_objectweb_fractal_api_Component* (*newFcInstance)(
    void *_this,   Rorg_objectweb_fractal_api_Type* type,
    void* ctrlDesc, void* cntntDesc);
};
struct Morg_objectweb_fractal_api_type_ComponentType {
  jboolean (*isFcSubTypeOf)(void *_this, Rorg_objectweb_fractal_api_Type* type);
  Rorg_objectweb_fractal_api_type_InterfaceType** (*getFcInterfaceTypes)
                                            (void *_this);
  Rorg_objectweb_fractal_api_type_InterfaceType* (*getFcInterfaceType)
                                            (void *_this, char* name);
};
struct Morg_objectweb_fractal_api_type_InterfaceType {
  jboolean (*isFcSubTypeOf)(void *_this, Rorg_objectweb_fractal_api_Type* type);
  char* (*getFcItfName)(void *_this);
  char* (*getFcItfSignature)(void *_this);
  jboolean (*isFcClientItf)(void *_this);
  jboolean (*isFcOptionalItf)(void *_this);
  jboolean (*isFcCollectionItf)(void *_this);
};
struct Morg_objectweb_fractal_api_type_TypeFactory {
  Rorg_objectweb_fractal_api_type_InterfaceType* (*createFcItfType)(
    void *_this, char* name, char* signature,
    jboolean isClient, jboolean isOptional, jboolean isCollection);
  Rorg_objectweb_fractal_api_type_ComponentType* (*createFcType)(
    void *_this, Rorg_objectweb_fractal_api_type_InterfaceType** interfaceTypes);
};
// where Rxyz types are defined by:
// typedef struct {
```

```
//   struct Mxyz *meth;
//   void *selfdata;
// } Rxyz;
```

C.3 OMG IDL API

```
typedef sequence<Object> ObjectArray;
typedef sequence<string> stringArray;
typedef sequence<octet> octetArray;
module org_objectweb_naming {
  exception NamingException { };
  interface NamingContext;
  interface Name {
    NamingContext getNamingContext ();
    octetArray encode () raises(NamingException);
  };
  interface NamingContext {
    Name export (in Object o, in Object hints) raises(NamingException);
    Name decode (in octetArray b) raises(NamingException);
  };
  interface Binder : NamingContext {
    Object bind (in Name n, in Object hints) raises(NamingException);
  };
};
module org_objectweb_fractal_api {
  exception NoSuchInterfaceException { };
  interface Type {
    boolean isFcSubTypeOf (in Type type);
  };
  interface Component {
    Type getFcType ();
    ObjectArray getFcInterfaces ();
    Object getFcInterface (in string interfaceName) raises(NoSuchInterfaceException);
  };
  typedef sequence<Component> ComponentArray;
  interface Interface {
    Component getFcItfOwner ();
    string getFcItfName ();
    Type getFcItfType ();
    boolean isFcInternalItf ();
  };
};
module org_objectweb_fractal_api_control {
  exception IllegalBindingException { };
  exception IllegalContentException { };
  exception IllegalLifeCycleException { };
  interface AttributeController { };
  interface BindingController {
    stringArray listFc ();
    Object lookupFc (in string clientItfName)
      raises(org_objectweb_fractal_api::NoSuchInterfaceException);
    void bindFc (in string clientItfName, in Object serverItf)
      raises(IllegalBindingException, IllegalLifeCycleException,
       org_objectweb_fractal_api::NoSuchInterfaceException);
    void unbindFc (in string clientItfName) raises(IllegalBindingException,
      IllegalLifeCycleException,
      org_objectweb_fractal_api::NoSuchInterfaceException);
  };
  interface ContentController {
    ObjectArray getFcInternalInterfaces ();
    Object getFcInternalInterface (in string interfaceName)
      raises(org_objectweb_fractal_api::NoSuchInterfaceException);
    org_objectweb_fractal_api::ComponentArray getFcSubComponents ();
    void addFcSubComponent (in org_objectweb_fractal_api::Component subComponent)
      raises(IllegalContentException, IllegalLifeCycleException);
    void removeFcSubComponent (in org_objectweb_fractal_api::Component subComponent)
      raises(IllegalContentException, IllegalLifeCycleException);
  };
  interface SuperController {
    org_objectweb_fractal_api::ComponentArray getFcSuperComponents ();
  };
  interface LifeCycleController {
    string getFcState ();
    void startFc () raises(IllegalLifeCycleException);
    void stopFc () raises(IllegalLifeCycleException);
  };
  interface NameController {
    string getFcName ();
    void setFcName (in string name);
  };
};
module org_objectweb_fractal_api_factory {
  exception InstantiationException { };
  interface GenericFactory {
    org_objectweb_fractal_api::Component newFcInstance (
      in org_objectweb_fractal_api::Type type,
      in Object controllerDesc, in Object contentDesc)
        raises(InstantiationException);
  };
  interface Factory {
    org_objectweb_fractal_api::Type getFcInstanceType ();
    Object getFcControllerDesc ();
    Object getFcContentDesc ();
    org_objectweb_fractal_api::Component newFcInstance ()
      raises(InstantiationException);
  };
};
module org_objectweb_fractal_api_type {
  interface InterfaceType : org_objectweb_fractal_api::Type {
    string getFcItfName ();
    string getFcItfSignature ();
    boolean isFcClientItf ();
    boolean isFcOptionalItf ();
    boolean isFcCollectionItf ();
  };
  typedef sequence<InterfaceType> InterfaceTypeArray;
  interface ComponentType : org_objectweb_fractal_api::Type {
    InterfaceTypeArray getFcInterfaceTypes ();
    InterfaceType getFcInterfaceType (in string name)
      raises(org_objectweb_fractal_api::NoSuchInterfaceException);
  };
  interface TypeFactory {
    InterfaceType createFcItfType (
      in string name, in string signature,
      in boolean isClient, in boolean isOptional, in boolean isCollection)
        raises(org_objectweb_fractal_api_factory::InstantiationException);
    ComponentType createFcType (in InterfaceTypeArray interfaceTypes)
      raises(org_objectweb_fractal_api_factory::InstantiationException);
  };
};
```

D GCM ADL

This is a Modified version of the standard Fractal ADL. It adds:

- new cardinalities for multicast and gathercast,
- exportation (renaming) of virtual nodes.

gcm.dtd

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!-- A DTD that includes all the "standard" Fractal ADL modules -->

<!-- ******************************************************************** -->
<!--                         AST nodes definitions                        -->
<!-- ******************************************************************** -->

<?add ast="definition" itf="org.objectweb.fractal.adl.Definition" ?>

<!-- components module -->
<?add ast="component"  itf="org.objectweb.fractal.adl.components.Component" ?>
<?add ast="definition" itf="org.objectweb.fractal.adl.components.ComponentDefinition" ?>

<!-- interfaces module -->
<?add ast="interface"  itf="org.objectweb.fractal.adl.interfaces.Interface" ?>
<?add ast="definition" itf="org.objectweb.fractal.adl.interfaces.InterfaceContainer" ?>
<?add ast="component"  itf="org.objectweb.fractal.adl.interfaces.InterfaceContainer" ?>

<!-- types module -->
<?add ast="interface"  itf="org.objectweb.fractal.adl.types.TypeInterface" ?>

<!-- bindings module -->
<?add ast="binding"    itf="org.objectweb.fractal.adl.bindings.Binding" ?>
<?add ast="definition" itf="org.objectweb.fractal.adl.bindings.BindingContainer" ?>
<?add ast="component"  itf="org.objectweb.fractal.adl.bindings.BindingContainer" ?>

<!-- attributes module -->
<?add ast="attribute"  itf="org.objectweb.fractal.adl.attributes.Attribute" ?>
<?add ast="attributes" itf="org.objectweb.fractal.adl.attributes.Attributes" ?>
<?add ast="definition" itf="org.objectweb.fractal.adl.attributes.AttributesContainer" ?>
<?add ast="component"  itf="org.objectweb.fractal.adl.attributes.AttributesContainer" ?>

<!-- implementations module -->
<?add ast="implementation" itf="org.objectweb.fractal.adl.implementations.Implementation" ?>
<?add ast="definition"     itf="org.objectweb.fractal.adl.implementations.ImplementationContainer" ?>
<?add ast="component"      itf="org.objectweb.fractal.adl.implementations.ImplementationContainer" ?>
<?add ast="controller"     itf="org.objectweb.fractal.adl.implementations.Controller" ?>
<?add ast="definition"     itf="org.objectweb.fractal.adl.implementations.ControllerContainer" ?>
<?add ast="component"      itf="org.objectweb.fractal.adl.implementations.ControllerContainer" ?>

<!-- loggers module -->
<?add ast="logger"     itf="org.objectweb.fractal.adl.loggers.Logger" ?>
<?add ast="definition" itf="org.objectweb.fractal.adl.loggers.LoggerContainer" ?>
<?add ast="component"  itf="org.objectweb.fractal.adl.loggers.LoggerContainer" ?>

<!-- nodes module -->
<?add ast="virtualNode" itf="net.coregrid.gcm.adl.nodes.VirtualNode" ?>
<?add ast="definition"  itf="org.objectweb.fractal.adl.nodes.VirtualNodeContainer" ?>
<?add ast="component"   itf="org.objectweb.fractal.adl.nodes.VirtualNodeContainer" ?>


<!-- arguments module -->
<?add ast="definition" itf="org.objectweb.fractal.adl.arguments.ArgumentDefinition" ?>

<!-- coordinates module -->
<?add ast="coordinates" itf="org.objectweb.fractal.adl.coordinates.Coordinates" ?>
<?add ast="definition"  itf="org.objectweb.fractal.adl.coordinates.CoordinatesContainer" ?>
<?add ast="component"   itf="org.objectweb.fractal.adl.coordinates.CoordinatesContainer" ?>
```

```
<!-- comments module -->
<?add ast="comment"        itf="org.objectweb.fractal.adl.comments.Comment" ?>
<?add ast="definition"     itf="org.objectweb.fractal.adl.comments.CommentContainer" ?>
<?add ast="component"      itf="org.objectweb.fractal.adl.comments.CommentContainer" ?>
<?add ast="interface"      itf="org.objectweb.fractal.adl.comments.CommentContainer" ?>
<?add ast="binding"        itf="org.objectweb.fractal.adl.comments.CommentContainer" ?>
<?add ast="attributes"     itf="org.objectweb.fractal.adl.comments.CommentContainer" ?>
<?add ast="attribute"      itf="org.objectweb.fractal.adl.comments.CommentContainer" ?>
<?add ast="controller"     itf="org.objectweb.fractal.adl.comments.CommentContainer" ?>
<?add ast="templateController" itf="org.objectweb.fractal.adl.comments.CommentContainer" ?>
<?add ast="implementation" itf="org.objectweb.fractal.adl.comments.CommentContainer" ?>


<!-- exported virtual nodes module -->
<?add ast="exportedVirtualNodes"  itf="net.coregrid.gcm.adl.vnexportation.ExportedVirtualNodes" ?>
<?add ast="exportedVirtualNode"  itf="net.coregrid.gcm.adl.vnexportation.ExportedVirtualNode" ?>
<?add ast="composedFrom"  itf="net.coregrid.gcm.adl.vnexportation.ComposedFrom" ?>
<?add ast="composingVirtualNode"  itf="net.coregrid.gcm.adl.vnexportation.ComposingVirtualNode" ?>

<?add ast="definition" itf="net.coregrid.gcm.adl.vnexportation.ExportedVirtualNodesContainer" ?>
<?add ast="component" itf="net.coregrid.gcm.adl.vnexportation.ExportedVirtualNodesContainer" ?>
<?add ast="exportedVirtualNodes" itf="net.coregrid.gcm.adl.vnexportation.ExportedVirtualNodeContainer" ?>
<?add ast="exportedVirtualNode" itf="net.coregrid.gcm.adl.vnexportation.ComposedFromContainer" ?>
<?add ast="composedFrom" itf="net.coregrid.gcm.adl.vnexportation.ComposingVirtualNodeContainer" ?>



<!-- ********************************************************************** -->
<!--                  Mapping from XML names to AST names                  -->
<!-- ********************************************************************** -->

<?map xml="binding.client" ast="binding.from" ?>
<?map xml="binding.server" ast="binding.to" ?>

<?map xml="content" ast="implementation" ?>
<?map xml="content.class" ast="implementation.className" ?>

<?map xml="controller.desc" ast="controller.descriptor" ?>

<?map xml="virtual-node" ast="virtualNode" ?>

<!-- ********************************************************************** -->
<!--                          XML syntax definition                          -->
<!-- ********************************************************************** -->

<!ELEMENT definition (comment*,interface*,exportedVirtualNodes?,
      component*,binding*,content?,attributes?,controller?,logger?,virtual-node?,coordinates*) >
<!ATTLIST definition
  name CDATA #REQUIRED
  arguments CDATA #IMPLIED
  extends CDATA #IMPLIED
>

<!ELEMENT component (comment*,interface*,exportedVirtualNodes?,
      component*,binding*,content?,attributes?,controller?,logger?,virtual-node?,coordinates*) >
<!ATTLIST component
  name CDATA #REQUIRED
  definition CDATA #IMPLIED
>

<!ELEMENT interface (comment*) >
<!ATTLIST interface
  name CDATA #REQUIRED
  role (client | server) #IMPLIED
  signature CDATA #IMPLIED
  contingency (mandatory | optional) #IMPLIED
  cardinality (singleton | collection | multicast | gathercast) #IMPLIED
>

<!ELEMENT binding (comment*) >
<!ATTLIST binding
  client CDATA #REQUIRED
  server CDATA #REQUIRED
>

<!ELEMENT attributes (comment*,attribute*) >
<!ATTLIST attributes
  signature CDATA #IMPLIED
>
```

```
<!ELEMENT attribute (comment*) >
<!ATTLIST attribute
  name CDATA #REQUIRED
  value CDATA #REQUIRED
>

<!ELEMENT controller (comment*) >
<!ATTLIST controller
  desc CDATA #REQUIRED
>

<!ELEMENT content (comment*) >
<!ATTLIST content
  class CDATA #REQUIRED
>

<!ELEMENT logger EMPTY >
<!ATTLIST logger
  name CDATA #REQUIRED
>

<!ELEMENT virtual-node EMPTY >
<!ATTLIST virtual-node
  name CDATA #REQUIRED
  cardinality (single | multiple) "single"
>

<!ELEMENT coordinates (coordinates*) >
<!ATTLIST coordinates
  name  CDATA #REQUIRED
  x0    CDATA #REQUIRED
  x1    CDATA #REQUIRED
  y0    CDATA #REQUIRED
  y1    CDATA #REQUIRED
  color CDATA #IMPLIED
>

<!ELEMENT comment EMPTY >
<!ATTLIST comment
  language CDATA #IMPLIED
  text CDATA #IMPLIED
>

<!ELEMENT exportedVirtualNodes (exportedVirtualNode*) >


<!ELEMENT exportedVirtualNode (composedFrom) >
<!ATTLIST exportedVirtualNode
     name CDATA #REQUIRED
>

<!ELEMENT composedFrom (composingVirtualNode+) >

<!ELEMENT composingVirtualNode EMPTY>
<!ATTLIST composingVirtualNode
     component CDATA #REQUIRED
     name CDATA #REQUIRED
>
```

E Deployment descriptor schema

This is the schema definition for the file that describes the mapping between virtual nodes and the Grid infrastructure. It is currently implemented within the ProActive library and references it. An independent specification must evolve from this first step.

Interoperability and deployment on the followings protocols and schedulers are currently specified: ssh, gsissh, rsh, rlogin, LSF, PBS, Sun grid engine, Oar, Prun, Globus, Unicore, glite EGEE, Arc Nordugrid.

DeploymentDescriptor.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
     xmlns:xs="http://www.w3.org/2001/XMLSchema">
     <!--
The following element ProActiveDescriptor defines the root of the schema
-->
     <xs:element name="ProActiveDescriptor" type="ProActiveDescriptorType">
          <xs:annotation>
               <xs:documentation>Deployment Descriptor Schema</xs:documentation>
          </xs:annotation>
          <xs:unique name="mainId">
               <xs:selector xpath="mainDefinition"/>
               <xs:field xpath="id"/>
          </xs:unique>
          <xs:unique name="processDefid">
               <xs:selector xpath="infrastructure/processes/processDefinition"/>
               <xs:field xpath="@id"/>
               <!--Uniqueness of the id attribute for infrastructure/processes/processDefinition element-->
          </xs:unique>
          <xs:unique name="serviceDefid">
               <xs:selector xpath="infrastructure/services/serviceDefinition"/>
               <xs:field xpath="@id"/>
               <!--Uniqueness of the id attribute for infrastructure/services/serviceDefinition element-->
          </xs:unique>
          <xs:unique name="filetransferDefid">
               <xs:selector xpath="fileTransferDefinitions/fileTransfer"/>
               <xs:field xpath="@id"/>
               <!--Uniqueness of the id attribute for FileTransferDefinitions/FileTransfer element-->
          </xs:unique>
          <xs:unique name="MapToVirtualNodeDefid">
               <xs:selector xpath="mainDefinition/mapToVirtualNode"/>
               <xs:field xpath="@value"/>
          </xs:unique>
          <xs:unique name="technicalServiceDefid">
               <xs:selector xpath="technicalServices/technicalServiceDefinition"/>
               <xs:field xpath="@id"/>
          </xs:unique>
          <xs:keyref name="MapToVirtualNodeValue" refer="MapToVNName">
               <xs:selector xpath="mainDefinition/mapToVirtualNode"/>
               <xs:field xpath="@value"/>
          </xs:keyref>
          <xs:keyref name="VirtNodeName" refer="vnName">
               <xs:selector xpath="componentDefinition/virtualNodesDefinition/virtualNode"/>
               <xs:field xpath="@name"/>
          </xs:keyref>
          <xs:keyref name="VirtNodeDeployFileTransfer" refer="filetransferid">
               <xs:selector xpath="componentDefinition/virtualNodesDefinition/virtualNode"/>
               <xs:field xpath="@fileTransferDeploy"/>
          </xs:keyref>
          <xs:keyref name="VirtNodeRetrieveFileTransfer" refer="filetransferid">
               <xs:selector xpath="componentDefinition/virtualNodesDefinition/virtualNode"/>
               <xs:field xpath="@fileTransferRetrieve"/>
          </xs:keyref>
          <xs:keyref name="vnFtServiceId" refer="serviceid">
               <xs:selector xpath="componentDefinition/virtualNodesDefinition/virtualNode"/>
               <xs:field xpath="@ftServiceId"/>
          </xs:keyref>
          <xs:keyref name="vnNameinAcquisitionPart" refer="virtualNodeNameForLookup">
               <xs:selector xpath="componentDefinition/virtualNodesAcquisition/virtualNode"/>
               <xs:field xpath="@name"/>
               <!--This section references the virtualNodeNameForLookup key defined above to ensure that
                   the value of attribute name in virtualNodeAcquisition/virtualNode element will have
                   the corresponding definition in deployment/lookup@virtualNode-->
          </xs:keyref>
          <xs:keyref name="vnNameinRegisterPart" refer="virtualNodeDefinitionName">
               <xs:selector xpath="deployment/register"/>
               <xs:field xpath="@virtualNode"/>
               <!--This section references the virtualNodeDefinitionName key defined above to ensure that
                   the virtualNode attribute of element register has a corresponding value defined in
                   attribute name of componentDefinition/virtualNodesDefinition/virtualNode-->
          </xs:keyref>
          <xs:keyref name="vmNameinMappingPart" refer="jvmNameinJvmsPart">
               <xs:selector xpath="deployment/mapping/map/jvmSet/vmName"/>
               <xs:field xpath="@value"/>
               <!--This section references the jvmNameinJvmsPart key defined above to ensure that the value
                   attribute of element mapping/map/jvmSet/vmName has a corresponding value defined in
```

```
         attribute name of  jvms/jvm-->
</xs:keyref>
<xs:keyref name="vnNameinLookupPart" refer="virtualNodeAcquisitionName">
     <xs:selector xpath="deployment/lookup"/>
     <xs:field xpath="@virtualNode"/>
     <!--This section references the virtualNodeAcquisitionName key defined above to ensure that
         the virtualNode attribute of element lookup  has a corresponding value defined in
         attribute name of componentDefinition/virtualNodesAcquisition/virtualNode-->
</xs:keyref>
<xs:keyref name="jvmReferenceid" refer="processid">
     <xs:selector xpath="infrastructure/processes/processDefinition/jvmProcess/extendedJvm"/>
     <xs:field xpath="@refid"/>
     <!-- This section references the processid key reference defined above to ensure that the
          refid attribute defined in deployment/jvms/jvm/creation/ has a correponding value in
          infrastructure/processes/processDefinition@id-->
</xs:keyref>
<xs:keyref name="processReferenceid" refer="processid">
     <xs:selector xpath="deployment/jvms/jvm/creation/processReference"/>
     <xs:field xpath="@refid"/>
     <!-- This section references the processid key reference defined above to ensure that the
          refid attribute defined in deployment/jvms/jvm/creation/ has a correponding value in
          infrastructure/processes/processDefinition@id-->
</xs:keyref>
<xs:keyref name="serviceReferenceid" refer="serviceid">
     <xs:selector xpath="deployment/jvms/jvm/acquisition/serviceReference"/>
     <xs:field xpath="@refid"/>
     <!-- This section references the serviceid key reference defined above to ensure that the
          refid attribute defined in deployment/jvms/jvm/acquisition/ has a correponding value
          in infrastructure/processes/processDefinition@id-->
</xs:keyref>
<!-- This section references the serviceid key reference defined above to ensure that the refid
     attribute defined in deployment/jvms/jvm/acquisition/ has a correponding value in
     infrastructure/services/serviceDefinition@id-->
<!--         <xs:keyref name="serviceReferenceid" refer="serviceid">
     <xs:selector xpath="deployment/jvms/jvm/acquisition/serviceReference"/>
     <xs:field xpath="@refid"/>
</xs:keyref>-->
<xs:keyref name="processListReferenceid" refer="processid">
     <xs:selector
          xpath="infrastructure/processes/processDefinition/processList/processReference"/>
     <xs:field xpath="@refid"/>
     <!--This section references the processid key defined above to ensure that the refid
          attribute defined in  processes/processDefinition/processList/processReference has a
          corresponding value in infrastructure/processes/processDefinition@id-->
</xs:keyref>
<xs:keyref name="processListbyHostReferenceid" refer="processid">
     <xs:selector
          xpath="infrastructure/processes/processDefinition/processListbyHost/processReference"/>
     <xs:field xpath="@refid"/>
     <!--This section references the processid key defined above to ensure that the refid attribute
         defined in  processes/processDefinition/processListbyHost/processReference has a
         corresponding value in infrastructure/processes/processDefinition@id-->
</xs:keyref>
<xs:keyref name="rshprocessReferenceid" refer="processid">
     <xs:selector
          xpath="infrastructure/processes/processDefinition/rshProcess/processReference"/>
     <xs:field xpath="@refid"/>
     <!--This section references the processid key defined above to ensure that the refid attribute
         defined in  processes/processDefinition/rshProcess/processReference has a corresponding
         value in infrastructure/processes/processDefinition@id-->
</xs:keyref>
<xs:keyref name="MapRshprocessReferenceid" refer="processid">
     <xs:selector
          xpath="infrastructure/processes/processDefinition/maprshProcess/processReference"/>
     <xs:field xpath="@refid"/>
     <!--This section references the processid key defined above to ensure that the refid attribute
         defined in processes/processDefinition/maprshProcess/processReference has a corresponding
         value in infrastructure/processes/processDefinition@id-->
</xs:keyref>
<xs:keyref name="sshprocessReferenceid" refer="processid">
     <xs:selector
          xpath="infrastructure/processes/processDefinition/sshProcess/processReference"/>
     <xs:field xpath="@refid"/>
     <!--This sectionreferences the processid key defined above to ensure that the refid attribute
         defined in  processes/processDefinition/sshProcess/processReference has a corresponding
         value in infrastructure/processes/processDefinition@id-->
</xs:keyref>
<xs:keyref name="rloginprocessReferenceid" refer="processid">
     <xs:selector
          xpath="infrastructure/processes/processDefinition/rloginProcess/processReference"/>
     <xs:field xpath="@refid"/>
     <!--This section references the processid key defined above to ensure that the refid attribute
         defined in  processes/processDefinition/rloginProcess/processReference has a corresponding
         value in infrastructure/processes/processDefinition@id-->
</xs:keyref>
<xs:keyref name="bsubprocessReferenceid" refer="processid">
     <xs:selector
          xpath="infrastructure/processes/processDefinition/bsubProcess/processReference"/>
     <xs:field xpath="@refid"/>
     <!--This section references the processid key defined above to ensure that the refid attribute
         defined in  processes/processDefinition/bsubProcess/processReference has a corresponding
         value in infrastructure/processes/processDefinition@id-->
</xs:keyref>
<xs:keyref name="globusprocessReferenceid" refer="processid">
     <xs:selector
          xpath="infrastructure/processes/processDefinition/globusProcess/processReference"/>
     <xs:field xpath="@refid"/>
     <!--This section references the processid key defined above to ensure that the refid attribute
         defined in  processes/processDefinition/globusProcess/processReference has a corresponding
         value in infrastructure/processes/processDefinition@id-->
</xs:keyref>
<xs:keyref name="unicoreprocessReferenceid" refer="processid">
     <xs:selector
          xpath="infrastructure/processes/processDefinition/unicoreProcess/processReference"/>
     <xs:field xpath="@refid"/>
     <!--This section references the processid key defined above to ensure that the refid attribute
         defined in  processes/processDefinition/unicoreProcess/processReference has a corresponding
          value in infrastructure/processes/processDefinition@id-->
</xs:keyref>
<xs:keyref name="hierarchicalprocessProcessReferenceid" refer="processid">
     <xs:selector
          xpath="infrastructure/processes/processDefinition/hierarchicalProcess/processReference"/>
     <xs:field xpath="@refid"/>
     <!--This section references the processid key defined above to ensure that the refid attribute
         defined in  processes/processDefinition/hierarchicalProcess/processReference has a
         corresponding value in infrastructure/processes/processDefinition@id-->
</xs:keyref>
<xs:keyref name="hierarchicalprocessHierarchicalReferenceid" refer="processid">
     <xs:selector
          xpath="infrastructure/processes/processDefinition/hierarchicalProcess/hierarchicalReference"/>
     <xs:field xpath="@refid"/>
     <!--This section references the processid key defined above to ensure that the refid attribute
         defined in  processes/processDefinition/hierarchicalProcess/hierarchicalReference has a
         corresponding value in infrastructure/processes/processDefinition@id-->
</xs:keyref>
<xs:keyref name="tsReferenceid" refer="technicalServiceid">
     <xs:selector xpath="componentDefinition/virtualNodesDefinition/virtualNode"/>
     <xs:field xpath="@technicalServiceId"/>
</xs:keyref>
<xs:key name="virtualNodeDefinitionName">
     <xs:selector xpath="componentDefinition/virtualNodesDefinition/virtualNode"/>
     <xs:field xpath="@name"/>
     <!-- Key definition for the attribute name located at
          componentDefinition/virtualNodesDefinition/virtualNode. This key will be used to ensure
          that the virtualNode attribute of deployment/register element has a corresponding definition
          in componentDefinition/virtualNodesDefinition/virtualNode@name-->
</xs:key>
<xs:key name="virtualNodeAcquisitionName">
     <xs:selector xpath="componentDefinition/virtualNodesAcquisition/virtualNode"/>
     <xs:field xpath="@name"/>
     <!-- Key definition for the attribute name located at
          componentDefinition/virtualNodesAcquisition/virtualNode. This key will be used to
          ensure that the virtualNode attribute of deployment/lookup element has a corresponding
          definition in componentDefinition/virtualNodesAcquisition/virtualNode@name-->
</xs:key>
<xs:key name="vnName">
     <xs:selector xpath="deployment/mapping/map"/>
     <xs:field xpath="@virtualNode"/>
     <!-- Key definition for the attribute virtualNode located at deployment/mapping/map. This
          key will be used to ensure that the name attribute of virtualNodes/virtualNode element
          has a corresponding definition in deployment/mapping/map@virtualNode-->
</xs:key>
<xs:key name="MapToVNName">
     <xs:selector xpath="componentDefinition/virtualNodesDefinition/virtualNode"/>
     <xs:field xpath="@name"/>
     <!-- Key definition for the attribute virtualNode located at deployment/mapping/map. This key
          will be used to ensure that the name attribute of virtualNodes/virtualNode element has a
           corresponding definition in deployment/mapping/map@virtualNode-->
</xs:key>
<xs:key name="jvmNameinJvmsPart">
```

```
            <xs:selector xpath="deployment/jvms/jvm"/>
            <xs:field xpath="@name"/>
            <!--Key definition for the attribute name located at deployment/jvms/jvm. This key will be
                used to ensure that the value attribute of deployment/mapping/map/jvmSet/vmName element
                has a corresponding value defined in attribue name of jvms/jvm  -->
        </xs:key>
        <xs:key name="virtualNodeNameForLookup">
            <xs:selector xpath="deployment/lookup"/>
            <xs:field xpath="@virtualNode"/>
            <!--Key definition for the attribute virtualNode located at deployment/lookup. This key will
                be used to ensure that the name attribute of
                componentDefinition/virtualNodesAcquisition/virtualNode element has a corresponding value
                 defined in attribue virtualNode of deployment/lookup  -->
        </xs:key>
        <xs:key name="processid">
            <xs:selector xpath="infrastructure/processes/processDefinition"/>
            <xs:field xpath="@id"/>
            <!-- Key definition for the attribute id located at infrastructure/processes/processDefinition.
                 This key will be used to ensure either that the refid attribute defined in
                 deployment/jvms/jvm/creation/ has a correponding value in
                 infrastructure/processes/processDefinition@id and also that the refid attribute defined in
                 infrastructure/processDefinition/rshProcess-sshProcess-rloginProcess-bsubProcess-globusProcess/
                 processReference@refid has a correponding value in
                 infrastructure/processes/processDefinition@id -->
        </xs:key>
        <xs:key name="serviceid">
            <xs:selector xpath="infrastructure/services/serviceDefinition"/>
            <xs:field xpath="@id"/>
            <!-- Key definition for the attribute id located at  infrastructure/processes/processDefinition.
                 This key will be used to ensure either that the refid attribute defined in
                 deployment/jvms/jvm/creation/ has a correponding value in
                 infrastructure/processes/processDefinition@id and also that the refid attribute defined in
                 infrastructure/processDefinition/rshProcess-sshProcess-rloginProcess-bsubProcess-globusProcess/
                 processReference@refid has a correponding value in
                 infrastructure/processes/processDefinition@id -->
        </xs:key>
        <!-- Key definition for the attribute id located at  infrastructure/services/serviceDefinition.
             This key will be used to ensure either that the refid attribute defined in
             deployment/jvms/jvm/acquisition/ has a correponding value in
             infrastructure/services/serviceDefinition@id -->
<!--        <xs:key name="serviceid">
            <xs:selector xpath="infrastructure/services/serviceDefinition"/>
            <xs:field xpath="@id"/>
        </xs:key>-->
        <xs:key name="filetransferid">
            <xs:selector xpath="fileTransferDefinitions/fileTransfer"/>
            <xs:field xpath="@id"/>
            <!-- Key definition for the attribute id located at FileTransferDefinitions/FileTransfer.
                 This key will be used to ensure that the FileTransferDeploy & FileTransferRetrieve
                 attribute defined in the VirtualNodeExtType has a correponding value in
                 FileTransferDefinitions/FileTransfer@id -->
        </xs:key>
        <xs:key name="technicalServiceid">
            <xs:selector xpath="technicalServices/technicalServiceDefinition"/>
            <xs:field xpath="@id"/>
        </xs:key>
    </xs:element>
    <!--processReference element is defined as a global element, hence it can be referenced in many part
        of the document. It has one attribute: refid that must occur once and only once. This element must
        reference a process defined elsewhere in the infrastructure section. Hence in the infrastructure section
        there must be an element ProcessDefinition with the corresponding value for id attribute   -->
    <xs:element name="processReference" type="ProcessReferenceType"/>
    <xs:complexType name="ProcessReferenceType">
        <xs:attribute name="refid" type="xs:string" use="required"/>
    </xs:complexType>
    <!--commandPath element is defined as a global element, hence it can be referenced in many part of
        the document. It has one attribute: value that must occur once and only once. This element
        represents the path of the command to be executed   -->
    <xs:element name="commandPath" type="CommandPathType"/>
    <xs:complexType name="CommandPathType">
        <xs:attribute name="value" type="xs:string" use="required"/>
    </xs:complexType>
    <!--environment element is defined as global element, hence it can be referenced in many part of the
        document. It has one child: variable element that must occur once and only once. This element is
        used to defined environment variables for some processes. -->
    <xs:element name="environment" type="EnvironmentType"/>
    <xs:complexType name="EnvironmentType">
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="variable" type="EnvironmentVariableType"/>
        </xs:sequence>
```

```
</xs:complexType>
<!--variable element has two attributes: name and value. Both are required.The syntax is for instance
    name="DISPLAY" value="machine_name0.0"  -->
<xs:complexType name="EnvironmentVariableType">
     <xs:attribute name="name" type="xs:string" use="required"/>
     <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>
<!--variableRefType is defined as a global element, it will be used to define references to variables.-->
<xs:simpleType name="variableRefType">
     <xs:restriction base="xs:string">
          <xs:pattern value="\$\{[A-Za-z0-9.]+\}"/>
     </xs:restriction>
</xs:simpleType>
<!--PosintOrVariableType is defined as a union of a positive integer or a variable ref-->
<xs:simpleType name="PosintOrVariableType">
     <xs:union memberTypes="xs:positiveInteger variableRefType"/>
</xs:simpleType>
<!--BooleanOrVariableType is defined as a union of a positive integer or a variable ref-->
<xs:simpleType name="BooleanOrVariableType">
     <xs:union memberTypes="xs:boolean variableRefType"/>
</xs:simpleType>
<!--ListType is defined as a union of a positive integer or a variable ref-->
<xs:simpleType name="ListType">
     <xs:union memberTypes="variableRefType">
          <!--hostlist element is a list of string. For instance galere10.inria.fr pf3.inria.fr
          galere11.inria.fr -->
          <xs:simpleType>
               <xs:list itemType="xs:string"/>
          </xs:simpleType>
     </xs:union>
</xs:simpleType>
<!--PatternType is defined as a global element, hence it can be referenced in many part of the document.-->
<!--PatternType is defined as a union of a pattern or a variable type.-->
<xs:simpleType name="PatternType">
     <xs:union memberTypes="variableRefType">
          <xs:simpleType>
               <xs:restriction base="xs:string">
                    <!-- The value of the following pattern is on several only for cosmetic needs-->
                    <xs:pattern
                         value="((\s)*[\[](\s)*[0-9]+(\s)*[\-](\s)*([0-9])+(\s)*([\]](\s)*|[;]
                                (\s)*[0-9](\s)*[\]](\s)*))((\s)*|(\s)*[\^](\s)*[\[](\s)*
                                (([0-9]+|[0-9]+(\s)*[\-](\s)*[0-9]+)(\s)*[,]*(\s)*)+[\]](\s)*)"
                    />
               </xs:restriction>
          </xs:simpleType>
     </xs:union>
</xs:simpleType>
<!--CloseStreamType is defined as a global element, it's the union of a CloseStream and a variable type.-->
<xs:simpleType name="CloseStreamType">
     <xs:union memberTypes="variableRefType">
          <xs:simpleType>
               <xs:restriction base="xs:string">
                    <xs:enumeration value="yes"/>
                    <xs:enumeration value="no"/>
               </xs:restriction>
          </xs:simpleType>
     </xs:union>
</xs:simpleType>
<!--ProActiveDescriptor element has three or four childs: mainDefinition element, that may appears
    any number of times, componentDefinition element that must appear once and only once
    deployment element thay must appear once and only once infrastructure element that must appear
    once and only once -->
<xs:complexType name="ProActiveDescriptorType">
     <xs:sequence>
          <xs:element name="variables" type="VariablesType" minOccurs="0"/>
          <xs:element name="security" type="SecurityType" minOccurs="0"/>
          <xs:element name="mainDefinition" type="MainDefinitionType" minOccurs="0"/>
          <xs:element name="componentDefinition" type="ComponentDefinitionType">
               <xs:unique name="idName">
                    <xs:selector xpath=".//*"/>
                    <xs:field xpath="@name"/>
                    <!-- uniqueness of every name attribute in the component definition part -->
               </xs:unique>
               <!--<xs:keyref name="VirtNodeName" refer="vnName">
                    <xs:selector xpath="virtualNodesDefinition/virtualNode"/>
                    <xs:field xpath="@name"/> -->
               <!--This section references the vnName key defined above to ensure that the value
                   of attribute name in virtualNodeDefinition/virtualNode element will have the
                   corresponding definition in deployment/mapping/map@virtualNode -->
               <!-- </xs:keyref>-->
```

```
            </xs:element>
            <xs:element name="deployment" type="DeploymentType"/>
            <xs:element name="fileTransferDefinitions" type="FileTransferDefinitionsType"
                 minOccurs="0"/>
            <xs:element name="infrastructure" type="InfrastructureType" minOccurs="0"/>
            <xs:element name="technicalServices" type="TechnicalServiceType" minOccurs="0"/>
       </xs:sequence>
  </xs:complexType>

  <xs:complexType name="VariablesType">
       <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element name="descriptorVariable">
                 <xs:complexType>
                      <xs:attribute name="name" type="xs:string" use="required"/>
                      <xs:attribute name="value" type="xs:string" use="required"/>
                 </xs:complexType>
            </xs:element>
            <xs:element name="descriptorDefaultVariable">
                 <xs:complexType>
                      <xs:attribute name="name" type="xs:string" use="required"/>
                      <xs:attribute name="value" type="xs:string" use="required"/>
                 </xs:complexType>
            </xs:element>
            <xs:element name="programDefaultVariable">
                 <xs:complexType>
                      <xs:attribute name="name" type="xs:string" use="required"/>
                      <xs:attribute name="value" type="xs:string" use="required"/>
                 </xs:complexType>
            </xs:element>
            <xs:element name="programVariable">
                 <xs:complexType>
                      <xs:attribute name="name" type="xs:string" use="required"/>
                      <xs:attribute name="value" type="xs:string" use="optional" fixed=""/>
                 </xs:complexType>
            </xs:element>
            <xs:element name="javaPropertyVariable">
                 <xs:complexType>
                      <xs:attribute name="name" type="xs:string" use="required"/>
                      <xs:attribute name="value" type="xs:string" use="optional" fixed=""/>
                 </xs:complexType>
            </xs:element>
            <xs:element name="javaPropertyDescriptorDefault">
                 <xs:complexType>
                      <xs:attribute name="name" type="xs:string" use="required"/>
                      <xs:attribute name="value" type="xs:string" use="required"/>
                 </xs:complexType>
            </xs:element>
            <xs:element name="javaPropertyProgramDefault">
                 <xs:complexType>
                      <xs:attribute name="name" type="xs:string" use="required"/>
                      <xs:attribute name="value" type="xs:string" use="optional"/>
                 </xs:complexType>
            </xs:element>
            <xs:element name="includeXMLFile">
                 <xs:complexType>
                      <xs:attribute name="location" type="xs:string" use="required"/>
                 </xs:complexType>
            </xs:element>
            <xs:element name="includePropertyFile">
                 <xs:complexType>
                      <xs:attribute name="location" type="xs:string" use="required"/>
                 </xs:complexType>
            </xs:element>
       </xs:choice>
  </xs:complexType>
  <!--main definition must have at least one child, a mapToVirtualNode and a not-empty attribute,
      the main class name. It may also have, in first, a group of elements of type ArgType, which
      define the arguments of the main class.-->
  <xs:complexType name="MainDefinitionType">
       <xs:sequence>
            <xs:group ref="ArgGroup" minOccurs="0" maxOccurs="unbounded"/>
            <xs:group ref="MapToVirtualNodeGroup" maxOccurs="unbounded"/>
            <xs:group ref="ClassPathGroup" maxOccurs="unbounded"/>
       </xs:sequence>
       <xs:attribute name="class" type="xs:string" use="required"/>
       <xs:attribute name="id" type="xs:string" use="required"/>
  </xs:complexType>
  <!--Sequence of element of type ArgType-->
  <xs:group name="ArgGroup">
       <xs:sequence>
```

```
          <xs:element name="arg" type="ArgType" minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
</xs:group>
<xs:group name="ClassPathGroup">
     <xs:sequence>
          <xs:element name="classpath" type="ClasspathType" minOccurs="0"/>
     </xs:sequence>
</xs:group>
<!--a arg is defined by a required value attribute, containing the argument.-->
<xs:complexType name="ArgType">
     <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>
<!--MapToVirtualNode is composed of only one required attribute : the value, what must match
    with a virtual node defined in the descriptor.-->
<xs:complexType name="MapToVirtualNodeType">
     <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>
<!--MapToVirtualNodeGroup is composed of any number but at least one of MapToVirtualNode-->
<xs:group name="MapToVirtualNodeGroup">
     <xs:sequence>
          <xs:element name="mapToVirtualNode" type="MapToVirtualNodeType" maxOccurs="unbounded"/>
     </xs:sequence>
</xs:group>
<!--component definition element has one child, that is a group of element, one element(or many)
    of the group has to appear at least once or more -->
<xs:complexType name="ComponentDefinitionType">
     <xs:group ref="VirtualNodesGroup"/>
</xs:complexType>
<!--VirtualNodesGroup is composed of two elements VirtualNodesDefinition where are defined
    virtualNodes that will be built from the descriptor and VirtualNodesAcquisition where
    are defined virtualNodes created by performing a lookup with a given protocol(rmi or jini).
    Both element can appear 0 or more-->
<xs:group name="VirtualNodesGroup">
     <xs:sequence>
          <xs:element name="virtualNodesDefinition" type="VirtualNodesDefintionType" minOccurs="0"/>
          <xs:element name="virtualNodesAcquisition" type="virtualNodesAcquisitionType"
               minOccurs="0"/>
     </xs:sequence>
</xs:group>
<!--virtualnodesDefinition element has one child: virtualNode element that must appear once or more-->
<xs:complexType name="VirtualNodesDefintionType">
     <xs:sequence>
          <xs:element name="virtualNode" maxOccurs="unbounded">
               <xs:complexType>
                    <xs:complexContent>
                         <xs:extension base="VirtualNodeExtType">
                              <xs:attribute name="minNodeNumber" type="PosintOrVariableType"
                                   use="optional"/>
                         </xs:extension>
                    </xs:complexContent>
               </xs:complexType>
          </xs:element>
     </xs:sequence>
</xs:complexType>
<!-- virtualNodesAcquisition element has only one child: virtualNode element that must appear once or more-->
<xs:complexType name="virtualNodesAcquisitionType">
     <xs:sequence>
          <xs:element name="virtualNode" type="VirtualNodeType" maxOccurs="unbounded"/>
     </xs:sequence>
</xs:complexType>
<!-- virtualNode element of type VirtualNodeType has one child: information element that may not appear.
     This elemnt is used to add textual information about the virtualNode     and one
     attributes: name that must appear once and only once, it may have any value of type string.-->
<xs:complexType name="VirtualNodeType">
     <xs:sequence>
          <xs:element name="information" type="xs:string" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
<!--virtualNode of type VirtualNodeExtType is an exyension of VirtualNodeType. It has two other
    attributes: property that may not appear and can take four defined values. This attribute
    just aims to provide informations about the virtualNode, in case of different people writing
    different part of the document. If property unique is set whereas the virtualNode has more
    than one machine defined in the jvmSet element, an exception will be generated when parsing
    the document, even if the document is valid against the schema. The other attribute is timeout,
    its value is defined in second. When activating the virtualNode, if this attribute is set,
    the VirtualNode will wait until timeout expires before giving access to all its nodes -->
<xs:complexType name="VirtualNodeExtType">
     <xs:complexContent>
          <xs:extension base="VirtualNodeType">
```

```
               <xs:attribute name="property">
                    <xs:simpleType>
                         <xs:union memberTypes="variableRefType">
                              <xs:simpleType>
                                   <xs:restriction base="xs:string">
                                        <xs:enumeration value="unique"/>
                                        <xs:enumeration value="unique_singleAO"/>
                                        <xs:enumeration value="multiple"/>
                                        <xs:enumeration value="multiple_cyclic"/>
                                   </xs:restriction>
                              </xs:simpleType>
                         </xs:union>
                    </xs:simpleType>
               </xs:attribute>
               <xs:attribute name="ftServiceId" type="xs:string" use="optional"/>
               <xs:attribute name="timeout" type="PosintOrVariableType" use="optional"/>
               <xs:attribute name="waitForTimeout" type="BooleanOrVariableType" use="optional"
                             default="false"/>
               <xs:attribute name="fileTransferDeploy" type="xs:string" use="optional"/>
               <xs:attribute name="fileTransferRetrieve" type="xs:string" use="optional"/>
               <xs:attribute name="technicalServiceId" type="xs:string" use="optional"/>
          </xs:extension>
     </xs:complexContent>
</xs:complexType>
<!-- deployment element has two child: mapping element that must appear once and only once
                                      jvms element that must appear once and only once-->
<xs:complexType name="DeploymentType">
     <xs:group ref="DeploymentGroup"/>
</xs:complexType>
<xs:group name="DeploymentGroup">
     <xs:sequence>
          <xs:element name="register" type="RegisterType" minOccurs="0" maxOccurs="unbounded"/>
          <xs:element name="mapping" type="MappingType" minOccurs="0"/>
          <xs:element name="lookup" type="LookupType" minOccurs="0" maxOccurs="unbounded"/>
          <xs:element name="jvms" type="JvmsType" minOccurs="0"/>
     </xs:sequence>
</xs:group>
<xs:complexType name="RegisterType">
     <xs:attribute name="virtualNode" type="xs:string" use="required"/>
     <xs:attribute name="protocol" use="optional">
          <xs:simpleType>
               <xs:union memberTypes="variableRefType">
                    <xs:simpleType>
                         <xs:restriction base="xs:string">
                              <xs:enumeration value="rmi"/>
                              <xs:enumeration value="rmissh"/>
                              <xs:enumeration value="http"/>
                              <xs:enumeration value="jini"/>
                         </xs:restriction>
                    </xs:simpleType>
               </xs:union>
          </xs:simpleType>
     </xs:attribute>
</xs:complexType>
<!-- mapping element has only one child: map element that must appear once or more-->
<xs:complexType name="MappingType">
     <xs:sequence>
          <xs:element name="map" type="MapType" maxOccurs="unbounded"/>
     </xs:sequence>
</xs:complexType>
<!-- map element has one child: jvmSet element that must appear once and only once
                     one attribute: virtualNode that must appear once and only once, it may
                     have any value of type string-->
<xs:complexType name="MapType">
     <xs:sequence>
          <xs:element name="jvmSet" type="JvmSetType">
               <xs:unique name="vmname">
                    <xs:selector xpath="vmName"/>
                    <xs:field xpath="@value"/>
                    <!--Uniqueness of the value attribute for vmName element
                        inside a jvmSet collection-->
               </xs:unique>
          </xs:element>
     </xs:sequence>
     <xs:attribute name="virtualNode" type="xs:string" use="required"/>
</xs:complexType>
<!-- jvmSet element has one child, it is a choice between an element  vmName that must occur
     once or more and a group: CurrentVMGroup that must occur once and only once-->
<xs:complexType name="JvmSetType">
     <xs:sequence>
```

```
          <xs:group ref="CurrentVMGroup"/>
     </xs:sequence>
</xs:complexType>
<!--CurrentVMGroup consists in two elements  currentJVM  that must occur once and only once
    and vmName that may not appear, or can appear many times-->
<xs:group name="CurrentVMGroup">
     <xs:sequence>
          <xs:element name="currentJVM" type="CurrentJVMType" minOccurs="0"/>
          <xs:element name="vmName" type="VmNameType" minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
</xs:group>
<!--currentVM element has one attribute: protocol wich is required and can take only two values
    rmi or jini. This element when set, indicates that a node will be created  on the jvm running
    the program and this node will be mapped on the virtualNode defined in
    deployment/mapping/map@virtualNode-->
<xs:complexType name="CurrentJVMType">
     <xs:attribute name="protocol" use="optional">
          <xs:simpleType>
               <xs:union memberTypes="variableRefType">
                    <xs:simpleType>
                         <xs:restriction base="xs:string">
                              <xs:enumeration value="rmi"/>
                              <xs:enumeration value="rmissh"/>
                              <xs:enumeration value="jini"/>
                              <xs:enumeration value="http"/>
                         </xs:restriction>
                    </xs:simpleType>
               </xs:union>
          </xs:simpleType>
     </xs:attribute>
</xs:complexType>
<!-- vmName element is a simpleType that can take any value of type String. This element
     represents the name of a virtualMachine. This name must have a corresponding value
     in jvms/jvm section(name attribute)-->
<xs:complexType name="VmNameType">
     <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="LookupType">
     <xs:complexContent>
          <xs:extension base="RegisterType">
               <xs:attribute name="host" type="xs:string" use="required"/>
               <!-- should be integer for the port, but * would not be supported-->
               <xs:attribute name="port" type="xs:string" use="optional"/>
          </xs:extension>
     </xs:complexContent>
</xs:complexType>
<!-- jvms element has only one child: jvm element that must appear once or more-->
<xs:complexType name="JvmsType">
     <xs:sequence>
          <xs:element name="jvm" type="JvmType" maxOccurs="unbounded"/>
     </xs:sequence>
</xs:complexType>
<!--jvm element has two childs:acquisition element that must appear once and only once
    creation element that must appear once and only once two attributes: name that must
    appear once and only once. It represents the name of the virtualmachine to be created
    nodeNumber which is optionnal and represents the number of nodes that will be created
     on the enclosing jvm. Default is one-->
<xs:complexType name="JvmType">
     <xs:choice>
          <xs:element name="creation" type="CreationType"/>
          <xs:element name="acquisition" type="AcquisitionType"/>
     </xs:choice>
     <xs:attribute name="name" type="xs:string" use="required"/>
     <xs:attribute name="askedNodes" type="PosintOrVariableType" use="optional"/>
</xs:complexType>
<!--acquisition element has one element: serviceReference that identifies uniquely the
    service to use in order to acquire the jvm. -->
<xs:complexType name="AcquisitionType">
     <xs:sequence>
          <xs:element name="serviceReference" type="ServiceReferenceType"/>
     </xs:sequence>
</xs:complexType>
<xs:complexType name="ServiceReferenceType">
     <xs:attribute name="refid" use="required"/>
</xs:complexType>
<!--creation element has one child: processReference element defined as global element.
    This element must occur once and only once. The reference is used to map the jvm with a
    process defined in infrastructure section. This process will originate jvm’s creation.
    Hence such process must exist in the infrastructure section. -->
<xs:complexType name="CreationType">
```

```
          <xs:sequence>
               <xs:element ref="processReference"/>
          </xs:sequence>
     </xs:complexType>
     <!--FileTransferDefinitions element has one child:
         FileTransfer that can appear cero or more times -->
     <xs:complexType name="FileTransferDefinitionsType">
          <xs:sequence>
               <xs:element name="fileTransfer" type="FileTransferType" minOccurs="0"
                    maxOccurs="unbounded"/>
          </xs:sequence>
     </xs:complexType>
     <xs:complexType name="FileTransferType">
          <xs:sequence>
               <xs:element name="file" minOccurs="0" maxOccurs="unbounded">
                    <xs:complexType>
                         <xs:attribute name="src" type="xs:string" use="required"/>
                         <xs:attribute name="dest" type="xs:string" use="optional"/>
                    </xs:complexType>
               </xs:element>
               <xs:element name="dir" minOccurs="0" maxOccurs="unbounded">
                    <xs:complexType>
                         <xs:attribute name="src" type="xs:string" use="required"/>
                         <xs:attribute name="dest" type="xs:string" use="optional"/>
                         <xs:attribute name="includes" type="xs:string" use="optional"/>
                         <xs:attribute name="excludes" type="xs:string" use="optional"/>
                    </xs:complexType>
               </xs:element>
          </xs:sequence>
          <xs:attribute name="id" type="xs:string" use="required"/>
     </xs:complexType>
     <!--infrastructure element has two childs: processes and services element that can
         appear zero or  once -->
     <xs:complexType name="InfrastructureType">
          <xs:sequence>
               <xs:element name="processes" type="ProcessesType" minOccurs="0"/>
               <xs:element name="services" type="ServicesType" minOccurs="0"/>
          </xs:sequence>
     </xs:complexType>
     <!--  Technical Service is a sequence of services -->
     <xs:complexType name="TechnicalServiceType">
          <xs:sequence>
               <xs:element name="technicalServiceDefinition" type="TechnicalServiceDefinitionType"
                    minOccurs="1" maxOccurs="unbounded"/>
          </xs:sequence>
     </xs:complexType>
     <xs:complexType name="TechnicalServiceDefinitionType">
          <xs:sequence>
               <xs:element name="arg" type="TechnicalServiceArgType" minOccurs="0"
                    maxOccurs="unbounded"/>
          </xs:sequence>
          <xs:attribute name="id" type="xs:string" use="required"/>
          <xs:attribute name="class" type="xs:string" use="required"/>
     </xs:complexType>
     <xs:complexType name="TechnicalServiceArgType">
          <xs:attribute name="name" type="xs:string" use="required"/>
          <xs:attribute name="value" type="xs:string" use="required"/>
     </xs:complexType>
     <!--services element has one child: serviceDefinition element that must appear once or more -->
     <xs:complexType name="ServicesType">
          <xs:sequence>
               <xs:element name="serviceDefinition" type="ServiceDefinitionType" maxOccurs="unbounded"
               />
          </xs:sequence>
     </xs:complexType>
     <!-- ServiceDefinition has two childs: RMIRegistryLookup, P2PLookup but only one
     can be present in the parent element. This element contains also an attribute: id
     which must appear once and only once and represents the id of the service in
     order for
     jvms(serviceReference element in deployment/jvms/jvm/creation/processReference)
     to reference other services-->
     <xs:complexType name="ServiceDefinitionType">
          <xs:choice>
               <xs:element name="RMIRegistryLookup" type="RMIRegistryLookupType"/>
               <xs:element name="P2PService" type="P2PServiceType"/>
               <xs:element name="ProActiveScheduler" type="ProActiveSchedulerType"/>
               <xs:element name="faultTolerance" type="faultToleranceType"/>
          </xs:choice>
          <xs:attribute name="id" type="xs:string" use="required"/>
     </xs:complexType>
```

```
<!-- ftService has -->
<xs:complexType name="faultToleranceType">
     <xs:sequence>
          <xs:element name="protocol" type="protocolType"/>
          <xs:element name="globalServer" type="serverType" minOccurs="0"/>
          <xs:element name="locationServer" type="serverType" minOccurs="0"/>
          <xs:element name="recoveryServer" type="serverType" minOccurs="0"/>
          <xs:element name="checkpointServer" type="serverType" minOccurs="0"/>
          <xs:element name="resourceServer" type="serverType" minOccurs="0"/>
          <xs:element name="ttc" type="ttcType" minOccurs="0"/>
     </xs:sequence>
</xs:complexType>
<xs:complexType name="protocolType">
     <xs:attribute name="type" use="required">
          <xs:simpleType>
               <xs:union memberTypes="variableRefType">
                    <xs:simpleType>
                         <xs:restriction base="xs:string">
                              <xs:enumeration value="cic"/>
                              <xs:enumeration value="pml"/>
                         </xs:restriction>
                    </xs:simpleType>
               </xs:union>
          </xs:simpleType>
     </xs:attribute>
</xs:complexType>
<xs:complexType name="serverType">
     <xs:attribute name="url" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="ttcType">
     <xs:attribute name="value" type="PosintOrVariableType" use="required"/>
</xs:complexType>
<!-- RMIRegistryLookup has one attribute: the url to looup-->
<xs:complexType name="RMIRegistryLookupType">
     <xs:attribute name="url" type="xs:string" use="required"/>
</xs:complexType>
<!-- ProActiveScheduler has no childs. It has only 3 attributes:
     numberOfNodes: an estimate of the number of nodes needed
          to start the application. This number may be altered at runtime,
          if the job needs to have more ressources.
     schedulerUrl: the url of the scheduler. By default we connect to the
          following url: rmi://localhost:1234
     jvmParameters: JVM parameters, mainly system variables, that need to
          be submitted.
-->
<xs:complexType name="ProActiveSchedulerType">
     <xs:attribute name="numberOfNodes" type="PosintOrVariableType" use="required"/>
     <xs:attribute name="minNumberOfNodes" type="PosintOrVariableType" use="optional"/>
     <xs:attribute name="schedulerUrl" type="xs:string" use="optional"/>
     <xs:attribute name="jvmParameters" type="xs:string" use="optional"/>
</xs:complexType>
<!-- P2PService has one child peerSet that represents a set of peer's url.
It has also 6 attributes:
     nodesAsked: that represents the number of nodes requested,
     acq: the acquisition method (rmi, http, ibis, etc.),
     port: the listeming port number,
     NOA: the number of acquaintances,
     TTU: time to update,
     TTL: time to live,
     multi_proc_nodes: true for deploying 1 node / cpu,
     xml_path: the xml deployment descriptor path.
-->
<xs:complexType name="P2PServiceType">
     <xs:sequence>
          <xs:element name="peerSet" type="peerSetType" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="nodesAsked" use="required">
          <xs:simpleType>
               <xs:union memberTypes="PosintOrVariableType MaxType"/>
          </xs:simpleType>
     </xs:attribute>
     <xs:attribute name="NOA" type="PosintOrVariableType" use="optional"/>
     <xs:attribute name="TTU" type="PosintOrVariableType" use="optional"/>
     <xs:attribute name="TTL" type="PosintOrVariableType" use="optional"/>
     <xs:attribute name="acq" type="xs:string" use="optional"/>
     <xs:attribute name="port" type="PosintOrVariableType" use="optional"/>
     <xs:attribute name="multi_proc_nodes" type="BooleanOrVariableType" use="optional"/>
     <xs:attribute name="xml_path" type="xs:string" use="optional"/>
     <xs:attribute name="node_family_regexp" type="xs:string" use="optional"/>
</xs:complexType>
```

```
<xs:simpleType name="MaxType">
     <xs:restriction base="xs:string">
          <xs:enumeration value="MAX"/>
     </xs:restriction>
</xs:simpleType>
<!-- peerSet element has one child peer that can occurs once or more-->
<xs:complexType name="peerSetType">
     <xs:sequence>
          <xs:element name="peer" type="xs:string" maxOccurs="unbounded"/>
     </xs:sequence>
</xs:complexType>
<!--processes element has one child: processDefinition element that must appear once or more -->
<xs:complexType name="ProcessesType">
     <xs:sequence>
          <xs:element name="processDefinition" type="ProcessDefinitionType" maxOccurs="unbounded"
          />
     </xs:sequence>
</xs:complexType>
<!-- processDefinition element has seven childs: jvmProcess, rshProcess, sshProcess, rloginProcess,
     bsubProcess, globusProcess, prunProcess. But only one child can be present in
     the parent element. Such processes will be used to create jvms. This element contains also
     an attribute: id which must appear once and only once and represents the id of the process
     in order for processes or  jvms(processReference element in
     deployment/jvms/jvm/creation/processReference)  to reference other processes-->
<xs:complexType name="ProcessDefinitionType">
     <xs:choice>
          <xs:element name="jvmProcess" type="JvmProcessType"/>
          <xs:element name="rshProcess" type="RshProcessType"/>
          <xs:element name="maprshProcess" type="MapRshProcessType"/>
          <xs:element name="sshProcess" type="SshProcessType"/>
          <xs:element name="processList" type="ProcessListType"/>
          <xs:element name="processListbyHost" type="ProcessListbyHostType"/>
          <xs:element name="rloginProcess" type="RloginProcessType"/>
          <xs:element name="bsubProcess" type="BsubProcessType"/>
          <xs:element name="pbsProcess" type="PbsProcessType"/>
          <xs:element name="oarProcess" type="oarProcessType"/>
          <xs:element name="oarGridProcess" type="oarGridProcessType"/>
          <xs:element name="globusProcess" type="GlobusProcessType"/>
          <xs:element name="prunProcess" type="prunProcessType"/>
          <xs:element name="gridEngineProcess" type="sgeProcessType"/>
          <xs:element name="gLiteProcess" type="gLiteProcessType"/>
          <xs:element name="unicoreProcess" type="unicoreProcessType"/>
          <xs:element name="ngProcess" type="NGProcessType"/>
          <xs:element name="hierarchicalProcess" type="hierarchicalProcessType"/>
          <xs:element name="mpiProcess" type="mpiProcessType"/>
          <xs:element name="dependentProcessSequence" type="dependentProcessSequenceType"/>
          <xs:element name="independentProcessSequence" type="independentProcessSequenceType"/>
     </xs:choice>
     <xs:attribute name="id" type="xs:string" use="required"/>
</xs:complexType>
<!--jvmProcess element has six childs: bootclasspath, classpath, javaPath, policyFile,
log4jpropertiesFile, jvmParameters. They may not appear. classpath element represents
the classpath value for the enclosing jvmProcess. javaPath represents the location of the java command
 to run for the jvmProcess. policyFile element represents the location of the policy file
 to use in the jvmProcess.log4jpropertiesFilerepresents the location of the log4j properties file.
 jvmParameters element represents all options(except of course classpath and policy file)
 that can be given to the the jvm. This element has also one attribute: class which is required.
 This attribute can take only one value: org.objectweb.proactive.core.process.JVMNodeProcess.
 This element is used to create a local jvm-->
<xs:complexType name="JvmProcessType">
     <xs:sequence>
          <xs:element name="extendedJvm" type="ExtendedJVMType" minOccurs="0"/>
          <xs:element name="bootclasspath" type="ClasspathType" minOccurs="0"/>
          <xs:element name="classpath" type="ClasspathType" minOccurs="0"/>
          <xs:element name="javaPath" type="FilePathType" minOccurs="0"/>
          <xs:element name="policyFile" type="FilePathType" minOccurs="0"/>
          <xs:element name="log4jpropertiesFile" type="FilePathType" minOccurs="0"/>
          <xs:element name="ProActiveUserPropertiesFile" type="FilePathType" minOccurs="0"/>
          <xs:element name="jvmParameters" type="JvmParameterType" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="class" type="xs:string" use="required"
          fixed="org.objectweb.proactive.core.process.JVMNodeProcess"/>
     <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
</xs:complexType>
<xs:complexType name="ExtendedJVMType">
     <xs:complexContent>
          <xs:extension base="ProcessReferenceType">
               <xs:attribute name="overwriteParameters" use="optional">
                    <xs:simpleType>
                         <xs:union memberTypes="variableRefType">
```

```
                              <xs:simpleType>
                                   <xs:restriction base="xs:string">
                                        <xs:enumeration value="yes"/>
                                        <xs:enumeration value="no"/>
                                   </xs:restriction>
                              </xs:simpleType>
                         </xs:union>
                    </xs:simpleType>
               </xs:attribute>
          </xs:extension>
     </xs:complexContent>
</xs:complexType>
<!--classpath element has two child: absolutePath and relativePath.  One of the child must appear
    once or more. Both element can be present in the classpath definition. For instance an
    absolutePath element followed by a relativePath element-->
<xs:complexType name="ClasspathType">
     <xs:choice maxOccurs="unbounded">
          <xs:element name="absolutePath" type="AbsolutePathType"/>
          <xs:element name="relativePath" type="RelativePathType"/>
     </xs:choice>
</xs:complexType>
<!--elements of type FilePathType have two childs: absolutePath and relativePath. Only one child
    can be present in the parent element. javaPath, policyFile(defined above), and scriptPath element
    are of such type.-->
<xs:complexType name="FilePathType">
     <xs:choice>
          <xs:element name="absolutePath" type="AbsolutePathType"/>
          <xs:element name="relativePath" type="RelativePathType"/>
     </xs:choice>
</xs:complexType>
<!-- path element of type RelativePathType has three attributes:type which is fixed with value relative,
     origin which can take three values, and value.  -->
<xs:complexType name="RelativePathType">
     <xs:attribute name="origin" use="required">
          <xs:simpleType>
               <xs:union memberTypes="variableRefType">
                    <xs:simpleType>
                         <xs:restriction base="xs:string">
                              <xs:enumeration value="user.home"/>
                              <xs:enumeration value="user.dir"/>
                              <xs:enumeration value="user.classpath"/>
                         </xs:restriction>
                    </xs:simpleType>
               </xs:union>
          </xs:simpleType>
     </xs:attribute>
     <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>
<!-- path element of type AbsolutePathType has two attributes:type which is fixed with value absolute
     and value which is required.  -->
<xs:complexType name="AbsolutePathType">
     <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>
<!--jvmParameters represents options to be passed to the jvm. It has one child : parameter -->
<xs:complexType name="JvmParameterType">
     <xs:choice maxOccurs="unbounded">
          <xs:element name="parameter" type="parameter"/>
     </xs:choice>
</xs:complexType>
<!-- parameter has one attribute: value which is a string representing the parameter -->
<xs:complexType name="parameter">
     <xs:attribute name="value" type="xs:string"/>
</xs:complexType>
<!-- FileTransfer Process Specific Information -->
<xs:complexType name="FileTransferStructureType">
     <xs:sequence>
          <xs:element name="copyProtocol" minOccurs="0"/>
          <xs:element name="sourceInfo" type="FileTransferStructureInfoType" minOccurs="0"/>
          <xs:element name="destinationInfo" type="FileTransferStructureInfoType" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="refid" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="FileTransferStructureInfoType">
     <xs:attribute name="prefix" type="xs:string" use="optional"/>
     <xs:attribute name="hostname" type="xs:string" use="optional"/>
     <xs:attribute name="username" type="xs:string" use="optional"/>
     <xs:attribute name="password" type="xs:string" use="optional"/>
</xs:complexType>
<xs:complexType name="FileTransferRetrieveStructureType">
     <xs:sequence>
```

```
          <xs:element name="sourceInfo" type="FileTransferRetrieveStructureInfoType" minOccurs="0"/>
          <xs:element name="destinationInfo" type="FileTransferRetrieveStructureInfoType"
               minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="refid" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="FileTransferRetrieveStructureInfoType">
     <xs:attribute name="prefix" type="xs:string" use="optional"/>
</xs:complexType>
<!--rshProcess element has two childs: processReference element which represents a process
    defined elsewhere in the infrastructure section(ProcessDefinition@id). Most of time the referenced
    process will be a jvmProcess, it means once logged on a remote host with rsh protocol
    a jvm will be created on the remote host.This element has also an environment element
    that may not appear and three attributes: class which is required and can take only two values:
    org.objectweb.proactive.core.process.rsh.RSHJVMProcess or
    org.objectweb.proactive.core.process.rsh.RSHProcess, username attribute which is optional,
    and hostname which is required and represents the remote host to log on with rsh-->
<xs:complexType name="RshProcessType">
     <xs:sequence>
          <xs:element ref="environment" minOccurs="0"/>
          <xs:element ref="processReference"/>
          <xs:element ref="commandPath" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="class" use="required"
          fixed="org.objectweb.proactive.core.process.rsh.RSHProcess"/>
     <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
     <xs:attribute name="username" type="xs:string" use="optional"/>
     <xs:attribute name="hostname" type="xs:string" use="required"/>
     <!--     <xs:simpleType>
               <xs:restriction base="xs:string">
                    <xs:enumeration value="org.objectweb.proactive.core.process.rsh.RSHProcess"/>
                    <xs:enumeration value="org.objectweb.proactive.core.process.rsh.RSHJVMProcess"/>
               </xs:restriction>
          </xs:simpleType>
     </xs:attribute>-->
</xs:complexType>
<!-- The ProcessListType represents a list of ssh, rsh or rlogin processes, they are defined by
     the class attributes that identifies wich process is included in the list by fixedName
     that is the fixed part of hostnames, list that represents the sequence of number,
     padding and domain-->
<xs:complexType name="ProcessListType">
     <xs:sequence>
          <xs:element ref="environment" minOccurs="0"/>
          <xs:element ref="processReference"/>
          <xs:element ref="commandPath" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="class" use="required">
          <xs:simpleType>
               <xs:union memberTypes="variableRefType">
                    <xs:simpleType>
                         <xs:restriction base="xs:string">
                              <xs:enumeration
                                   value="org.objectweb.proactive.core.process.ssh.SSHProcessList"/>
                              <xs:enumeration
                                   value="org.objectweb.proactive.core.process.rsh.RSHProcessList"/>
                              <xs:enumeration
                                   value="org.objectweb.proactive.core.process.rsh.RLoginProcessList"/>
                         </xs:restriction>
                    </xs:simpleType>
               </xs:union>
          </xs:simpleType>
     </xs:attribute>
     <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
     <xs:attribute name="fixedName" type="xs:string" use="required"/>
     <xs:attribute name="list" type="PatternType" use="required"/>
     <xs:attribute name="padding" type="PosintOrVariableType" use="optional"/>
     <xs:attribute name="domain" type="xs:string" use="required"/>
     <xs:attribute name="username" type="xs:string" use="optional"/>
</xs:complexType>
<!-- The ProcessListbyHostType represents a list of ssh, rsh or rlogin processes, they are defined
     by the class attributes that identifies wich process is included in the list by hostlist
     which is a list of hostname and domain if all the host specified are in the same domain-->
<xs:complexType name="ProcessListbyHostType">
     <xs:sequence>
          <xs:element ref="environment" minOccurs="0"/>
          <xs:element ref="processReference"/>
          <xs:element ref="commandPath" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="class" use="required">
          <xs:simpleType>
```

```
                <xs:union memberTypes="variableRefType">
                     <xs:simpleType>
                          <xs:restriction base="xs:string">
                               <xs:enumeration
                                    value="org.objectweb.proactive.core.process.ssh.SSHProcessList"/>
                               <xs:enumeration
                                    value="org.objectweb.proactive.core.process.rsh.RSHProcessList"/>
                               <xs:enumeration
                                    value="org.objectweb.proactive.core.process.rsh.RLoginProcessList"/>
                          </xs:restriction>
                     </xs:simpleType>
                </xs:union>
           </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
      <xs:attribute name="hostlist" type="ListType" use="required"/>
      <xs:attribute name="domain" type="xs:string" use="optional"/>
      <xs:attribute name="username" type="xs:string" use="optional"/>
 </xs:complexType>
 <!--maprshProcess element has two child: processReference element which represents a process defined
     elsewhere in the infrastructure section(ProcessDefinition@id). Most of time  the referenced
     process will be a jvmProcess, it means once logged on a remote hosts with rsh protocol a jvm
     will be created on the remote host.This element has also  a scriptPath child which represents
     the location of the script oasis-exp.sh and three attributes: class which is required and can
     take only one value: org.objectweb.proactive.core.process.rsh.maprsh.MapRSHProcess,
     netgroup which is required and represents  group of remote hosts to log on with rsh.
     and parallelize which is optional and represents the parallelization degree -->
 <xs:complexType name="MapRshProcessType">
      <xs:sequence>
           <xs:element ref="processReference"/>
           <xs:element ref="commandPath" minOccurs="0"/>
           <xs:element name="scriptPath" type="FilePathType" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="class" use="required">
           <xs:simpleType>
                <xs:union memberTypes="variableRefType">
                     <xs:simpleType>
                          <xs:restriction base="xs:string">
                               <xs:enumeration
                                    value="org.objectweb.proactive.core.process.rsh.maprsh.MapRshProcess"
                               />
                          </xs:restriction>
                     </xs:simpleType>
                </xs:union>
           </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
      <xs:attribute name="parallelize" type="PosintOrVariableType" use="optional"/>
      <xs:attribute name="hostname" type="ListType" use="required"/>
 </xs:complexType>
 <!--sshProcess element has two child: processReference element which represents a process defined
     elsewhere in the infrastructure section(ProcessDefinition@id).and environment element that may
     not appear and three attributes: class which is required and can take only two values:
     org.objectweb.proactive.core.process.ssh.SSHJVMProcess or
     org.objectweb.proactive.core.process.ssh.SSHProcess, username attribute which is optional,
     and hostname which is required and represents the remote host to log on with ssh-->
 <xs:complexType name="SshProcessType">
      <xs:sequence>
           <xs:element ref="environment" minOccurs="0"/>
           <xs:element ref="processReference"/>
           <xs:element ref="commandPath" minOccurs="0"/>
           <xs:element name="fileTransferDeploy" type="FileTransferStructureType" minOccurs="0"
                maxOccurs="1"/>
           <xs:element name="fileTransferRetrieve" type="FileTransferRetrieveStructureType"
                minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="class" use="required"
           fixed="org.objectweb.proactive.core.process.ssh.SSHProcess"/>
      <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
      <xs:attribute name="username" type="xs:string" use="optional"/>
      <xs:attribute name="hostname" type="xs:string" use="required"/>
      <!--<xs:simpleType>
                <xs:restriction base="xs:string">
                     <xs:enumeration value="org.objectweb.proactive.core.process.ssh.SSHProcess"/>
                     <xs:enumeration value="org.objectweb.proactive.core.process.ssh.SSHJVMProcess"/>
                </xs:restriction>
           </xs:simpleType>
      </xs:attribute>-->
 </xs:complexType>
 <!--rloginProcess element has two childs: processReference element which represents a process
```

```
    defined elsewhere in the infrastructure section(ProcessDefinition@id) and environment element
    that may not appear and two attributes: class which is required and can take only one value:
    org.objectweb.proactive.core.process.lsf.RLoginProcess, and hostname which is required and
    represents the remote host to log on with rlogin-->
<xs:complexType name="RloginProcessType">
     <xs:sequence>
          <xs:element ref="environment" minOccurs="0"/>
          <xs:element ref="processReference"/>
          <xs:element ref="commandPath" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="class" type="xs:string" use="required"
          fixed="org.objectweb.proactive.core.process.rlogin.RLoginProcess"/>
     <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
     <xs:attribute name="hostname" type="xs:string" use="required"/>
     <xs:attribute name="username" type="xs:string" use="optional"/>
</xs:complexType>
<!--bsubProcess element has two childs: processReference element which represents a process
    defined elsewhere in the infrastructure section(ProcessDefinition@id)  and may not appear,
    bsubOption element that may not appear and represents different options of bsub command and
    three attributes: class which is required and can take only one value:
    org.objectweb.proactive.core.process.lsf.LSFBSubProcess, interactive which is optinal to run
    bsub command in interactive mode, and queue to specify in which queue to run the job.
    This element will be used to run bsub command-->
<xs:complexType name="BsubProcessType">
     <xs:sequence>
          <xs:element ref="processReference"/>
          <xs:element ref="commandPath" minOccurs="0"/>
          <xs:element name="bsubOption" type="BsubOptionType" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="class" use="required">
          <xs:simpleType>
               <xs:restriction base="xs:string">
                    <xs:enumeration value="org.objectweb.proactive.core.process.lsf.LSFBSubProcess"/>
                    <xs:enumeration value="org.objectweb.proactive.core.process.lsf.CNLSFProcess"/>
               </xs:restriction>
          </xs:simpleType>
     </xs:attribute>
     <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
     <xs:attribute name="interactive" type="BooleanOrVariableType" use="optional"/>
     <xs:attribute name="jobname" type="xs:string" use="optional"/>
     <xs:attribute name="queue"  type="xs:string" use="optional"/>
</xs:complexType>
<!--bsubOption element has three childs:hostlist which represents the -m option of bsub command,
    processor which indicate the number of processor requested, scriptPath element which represents
    the location of the startRuntime.sh script to run jobs on LSF cluster.These three elements
    may not appear -->
<xs:complexType name="BsubOptionType">
     <xs:sequence>
          <xs:element name="hostlist" type="ListType" minOccurs="0"/>
          <xs:element name="processor" type="PosintOrVariableType" minOccurs="0"/>
          <xs:element name="resourceRequirement" minOccurs="0">
               <xs:complexType>
                    <xs:attribute name="value" type="xs:string"/>
               </xs:complexType>
          </xs:element>
          <xs:element name="scriptPath" type="FilePathType" minOccurs="0"/>
     </xs:sequence>
</xs:complexType>
<!--PBSProcess-->
<xs:complexType name="PbsProcessType">
     <xs:sequence>
          <xs:element ref="processReference"/>
          <xs:element ref="commandPath" minOccurs="0"/>
          <xs:element name="pbsOption" type="PbsOptionType" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="class" type="xs:string" use="required"
          fixed="org.objectweb.proactive.core.process.pbs.PBSSubProcess"/>
     <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
     <xs:attribute name="interactive" type="BooleanOrVariableType" use="optional"/>
     <xs:attribute name="queue" type="xs:string" use="optional"/>
</xs:complexType>
<!--PbsOption-->
<xs:complexType name="PbsOptionType">
     <xs:complexContent>
          <xs:extension base="prunOptionType">
               <xs:sequence>
                    <xs:element name="scriptPath" type="FilePathType" minOccurs="0"/>
               </xs:sequence>
          </xs:extension>
     </xs:complexContent>
```

```
</xs:complexType>
<!--gLiteProcess element has six children: processReference element, gLiteOptions which contains
    JDLFilePath child and SdtOutput child
environment, rank, inputData, requirements-->
<!--Environment : List of string representing environment settings that hava to be performed on
    the execution machine.-->
<!--Requirements : String representing job requirements on ressources. Default is :
    Requirements = other.GlueCEStateStatusus == "Production";-->
<!--InputData : string or list of string representing physical file location on SE.
    Names have to be prefixed with "lfn", "guid", "lds", "query".-->
<!--Rank : ClassAd foating-point expression that states how to rank CEs that have already met
    the Requirements expression. Default is : Rank = -other.GlueCEStateEstimatedResponseTime; -->
<xs:complexType name="gLiteProcessType">
     <xs:sequence>
          <xs:element ref="processReference"/>
          <xs:element name="environment" type="xs:string" minOccurs="0"/>
          <xs:element name="requirements" type="xs:string" minOccurs="0"/>
          <xs:element name="inputData" type="gLiteInputDataType" minOccurs="0"/>
          <xs:element name="rank" minOccurs="0">
               <xs:complexType>
                    <xs:simpleContent>
                         <xs:extension base="xs:string">
                              <xs:attribute name="fuzzyrank" type="xs:string"/>
                         </xs:extension>
                    </xs:simpleContent>
               </xs:complexType>
          </xs:element>
          <xs:element name="gLiteOptions" type="gLiteOptionsType" minOccurs="0"/>
          <xs:element name="JdlTransferDeploy" type="FileTransferStructureType" minOccurs="0"
               maxOccurs="1"/>
     </xs:sequence>
     <xs:attribute name="class" type="xs:string" use="required"
          fixed="org.objectweb.proactive.core.process.glite.GLiteProcess"/>
     <xs:attribute name="hostname" type="xs:string"/>
     <xs:attribute name="virtualOrganisation" type="xs:string" use="required"/>
     <xs:attribute name="JDLFileName" type="xs:string" use="required"/>
     <xs:attribute name="Type" type="xs:string" default="Job"/>
     <xs:attribute name="executable" type="xs:string" default="/bin/sh"/>
     <xs:attribute name="stdOutput" type="xs:string" default="output.log"/>
     <xs:attribute name="stdInput" type="xs:string" default="input.log"/>
     <xs:attribute name="stdError" type="xs:string" default="error.log"/>
     <xs:attribute name="outputse" type="xs:string"/>
     <xs:attribute name="retryCount" type="xs:string"/>
     <xs:attribute name="myProxyServer" type="xs:string"/>
</xs:complexType>
<!--gLiteRankType-->
<xs:complexType name="gLiteRankType">
     <xs:attribute name="fuzzyrank" type="BooleanOrVariableType" default="false"/>
</xs:complexType>
<!--gLiteInputDataType-->
<xs:complexType name="gLiteInputDataType">
     <xs:attribute name="dataAccessProtocol" type="xs:string"/>
     <xs:attribute name="storageIndex" type="xs:string"/>
     <xs:attribute name="dataCatalog" type="xs:string"/>
</xs:complexType>
<!--gLiteJobType-->
<xs:complexType name="gLiteJobType">
     <xs:choice>
          <xs:element name="Normal"/>
          <xs:element name="Interactive">
               <xs:complexType>
                    <xs:attribute name="listerport" type="xs:string"/>
               </xs:complexType>
          </xs:element>
          <xs:element name="MPICH">
               <xs:complexType>
                    <xs:attribute name="nodeNumber" type="PosintOrVariableType" use="required"/>
               </xs:complexType>
          </xs:element>
          <xs:element name="Checkpointable">
               <xs:complexType>
                    <xs:attribute name="currentStep" type="xs:string" use="required"/>
                    <xs:attribute name="jobStep" type="xs:string" use="required"/>
               </xs:complexType>
          </xs:element>
          <xs:element name="Partitionable">
               <xs:complexType>
                    <xs:attribute name="currentStep" type="xs:string" use="required"/>
                    <xs:attribute name="jobStep" type="xs:string" use="required"/>
               </xs:complexType>
```

```
            </xs:element>
        </xs:choice>
    </xs:complexType>
    <!--gLiteOptions-->
    <xs:complexType name="gLiteOptionsType">
        <xs:sequence>
            <xs:element name="configFile" type="FilePathType" minOccurs="0"/>
            <xs:element name="JDLFilePath" type="FilePathType" minOccurs="0"/>
            <xs:element name="JDLRemoteFilePath" type="FilePathType" minOccurs="0"/>
            <xs:element name="jobType" type="gLiteJobType" minOccurs="0"/>
            <xs:element name="arguments" type="xs:string" minOccurs="0"/>
            <xs:element name="inputSandbox" type="xs:string" minOccurs="0"/>
            <xs:element name="outputSandbox" type="xs:string" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
    <!-- end of gLiteProcessType -->
    <!--prunProcess element has two childs: processReference element which represents a process
        defined elsewhere in the infrastructure section(ProcessDefinition@id)  and may not appear,
        prunOption element that may not appear and represents different options of prun command
        and three attributes: class which is required and can take only one value:
        org.objectweb.proactive.core.process.prun.PrunSubProcess, interactive which is optinal
        to run prun command in interactive mode, and queue to specify in which queue to run the job.
        This element will be used to run prun command-->
    <xs:complexType name="prunProcessType">
        <xs:sequence>
            <xs:element ref="processReference"/>
            <xs:element ref="commandPath" minOccurs="0"/>
            <xs:element name="prunOption" type="prunOptionType" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="class" type="xs:string" use="required"
            fixed="org.objectweb.proactive.core.process.prun.PrunSubProcess"/>
        <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
        <xs:attribute name="queue" type="xs:string" use="optional"/>
    </xs:complexType>
    <!--prunOption element has three childs:hostlist which represents the -m option of bsub command,
        processor which indicate the number of processor requested, scriptPath element which
        represents the location of the startRuntime.sh script to run jobs on LSF cluster.These
        three elements may not appear -->
    <xs:complexType name="prunOptionType">
        <xs:sequence>
            <xs:element name="hostlist" type="ListType" minOccurs="0"/>
            <xs:element name="hostsNumber" type="PosintOrVariableType" minOccurs="0"/>
            <xs:element name="processorPerNode" type="PosintOrVariableType" minOccurs="0"/>
            <xs:element name="bookingDuration" type="xs:string" minOccurs="0"/>
            <xs:element name="outputFile" type="xs:string" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
    <!--oarProcess-->
    <xs:complexType name="oarProcessType">
        <xs:sequence>
            <xs:element ref="processReference"/>
            <xs:element ref="commandPath" minOccurs="0"/>
            <xs:element name="oarOption" type="OarOptionType" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="class" type="xs:string" use="required"
            fixed="org.objectweb.proactive.core.process.oar.OARSubProcess"/>
        <xs:attribute name="bookedNodesAccess" use="optional">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="rsh"/>
                    <xs:enumeration value="ssh"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
        <xs:attribute name="interactive" type="BooleanOrVariableType" use="optional"/>
        <xs:attribute name="queue" type="xs:string" use="optional"/>
    </xs:complexType>
    <!--oarOption-->
    <xs:complexType name="OarOptionType">
        <xs:sequence>
            <xs:element name="resources" type="xs:string" minOccurs="0"/>
            <!--<xs:element name="properties" type="xs:string" minOccurs="0"/>-->
            <xs:element name="scriptPath" type="FilePathType" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
    <!--oarGridProcess-->
    <xs:complexType name="oarGridProcessType">
        <xs:sequence>
            <xs:element ref="processReference"/>
```

```
          <xs:element ref="commandPath" minOccurs="0"/>
          <xs:element name="oarGridOption" type="OarGridOptionType"/>
     </xs:sequence>
     <xs:attribute name="class" type="xs:string" use="required"
          fixed="org.objectweb.proactive.core.process.oar.OARGRIDSubProcess"/>
     <xs:attribute name="queue" type="xs:string" use="optional"/>
     <xs:attribute name="bookedNodesAccess" use="optional">
          <xs:simpleType>
               <xs:restriction base="xs:string">
                    <xs:enumeration value="rsh"/>
                    <xs:enumeration value="ssh"/>
               </xs:restriction>
          </xs:simpleType>
     </xs:attribute>
     <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
</xs:complexType>
<!--oarGridOption-->
<xs:complexType name="OarGridOptionType">
     <xs:sequence>
          <xs:element name="resources" type="xs:string"/>
          <xs:element name="walltime" type="xs:string" minOccurs="0"/>
          <xs:element name="scriptPath" type="FilePathType" minOccurs="0"/>
     </xs:sequence>
</xs:complexType>
<!--sge process -->
<xs:complexType name="sgeProcessType">
     <xs:sequence>
          <xs:element ref="processReference"/>
          <xs:element ref="commandPath" minOccurs="0"/>
          <xs:element name="gridEngineOption" type="GridEngineOptionType"/>
     </xs:sequence>
     <xs:attribute name="class" type="xs:string" use="required"
          fixed="org.objectweb.proactive.core.process.gridengine.GridEngineSubProcess"/>
     <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
     <xs:attribute name="interactive" type="BooleanOrVariableType" use="optional"/>
     <xs:attribute name="queue" type="xs:string" use="optional"/>
</xs:complexType>
<xs:complexType name="GridEngineOptionType">
     <xs:sequence>
          <xs:element name="hostsNumber" type="xs:string"/>
          <xs:element name="bookingDuration" type="xs:string"/>
          <xs:element name="scriptPath" type="FilePathType"/>
          <xs:element name="hostlist" type="xs:string" minOccurs="0"/>
          <xs:element name="parallelEnvironment" type="xs:string" minOccurs="1" maxOccurs="1"/>
     </xs:sequence>
</xs:complexType>

<!--globusProcess element has three children: processReference element which represents a process
    defined elsewhere in the infrastructure section(ProcessDefinition@id) , environment,
    and globusOption element  which represents different options of globus command.
    They may not appear and two attributes: class which is required and can take only one value:
    org.objectweb.proactive.core.process.globus.GlobusProcess, and hostname which is required and
    represents the globus host where the RSL request is destinated -->
<xs:complexType name="GlobusProcessType">
     <xs:sequence>
          <xs:element ref="environment" minOccurs="0"/>
          <xs:element ref="processReference"/>
          <!--     <xs:element ref="commandPath" minOccurs="0"/>-->
          <xs:element name="globusOption" type="GlobusOptionType" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="class" type="xs:string" use="required"
          fixed="org.objectweb.proactive.core.process.globus.GlobusProcess"/>
     <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
     <xs:attribute name="hostname" type="xs:string" use="required"/>
     <xs:attribute name="queue" type="xs:string" use="optional"/>
</xs:complexType>
<!--globusOption element has one child count element-->
<xs:complexType name="GlobusOptionType">
     <xs:sequence>
          <xs:element name="count" type="PosintOrVariableType" minOccurs="0"/>
          <xs:element name="maxTime" type="PosintOrVariableType" minOccurs="0"/>
          <xs:element name="outputFile" type="xs:string" minOccurs="0"/>
          <xs:element name="errorFile" type="xs:string" minOccurs="0"/>
     </xs:sequence>
</xs:complexType>
<xs:complexType name="SecurityType">
     <xs:attribute name="file" type="xs:string"/>
</xs:complexType>
<!--unicoreProcess element has 4 childs: processReference element which represents a process
    defined elsewhere, unicoreDirPath which represents the unicore client configuration directory,
    keyFilePath which represnets de unicore cliente certificate file and the unicoreOption
    which represent a unicore site options-->
<xs:complexType name="unicoreProcessType">
     <xs:sequence>
          <xs:element ref="processReference"/>
          <xs:element name="unicoreDirPath" type="FilePathType" minOccurs="0"/>
          <xs:element name="keyFilePath" type="FilePathType" minOccurs="0"/>
          <xs:element name="unicoreOption" type="unicoreOptionType" minOccurs="0"/>
          <xs:element name="fileTransferDeploy" type="FileTransferStructureType" minOccurs="0"
               maxOccurs="1"/>
          <xs:element name="fileTransferRetrieve" type="FileTransferRetrieveStructureType"
               minOccurs="0" maxOccurs="1"/>
     </xs:sequence>
     <xs:attribute name="class" type="xs:string" use="required"
          fixed="org.objectweb.proactive.core.process.unicore.UnicoreProcess"/>
     <xs:attribute name="keypassword" type="xs:string" use="optional"/>
     <xs:attribute name="jobname" type="xs:string" use="optional" default="ProActiveUnicoreJob"/>
     <xs:attribute name="submitjob" type="BooleanOrVariableType" use="optional" default="true"/>
     <xs:attribute name="savejob" type="BooleanOrVariableType" use="optional" default="false"/>
     <!--xs:attribute name="closeStream" type="CloseStreamType" use="optional"/-->
</xs:complexType>
<!--unicoreOption has 2 childs: unicoreUsiteType represents the Unicore site, unicoreVsiteType
    represents the virtual site
within the Unicore site. Notice that not all unicore features are supported. -->
<xs:complexType name="unicoreOptionType">
     <xs:sequence>
          <xs:element name="usite" type="unicoreUsiteType" minOccurs="0"/>
          <xs:element name="vsite" type="unicoreVsiteType" minOccurs="0"/>
     </xs:sequence>
</xs:complexType>
<xs:complexType name="unicoreUsiteType">
     <xs:attribute name="name" type="xs:string" use="optional"/>
     <xs:attribute name="url" type="xs:string" use="required"/>
     <xs:attribute name="type" use="required">
          <xs:simpleType>
               <xs:union memberTypes="variableRefType">
                    <xs:simpleType>
                         <xs:restriction base="xs:string">
                              <xs:enumeration value="CLASSIC"/>
                              <xs:enumeration value="REGISTRY"/>
                         </xs:restriction>
                    </xs:simpleType>
               </xs:union>
          </xs:simpleType>
     </xs:attribute>
</xs:complexType>
<xs:complexType name="unicoreVsiteType">
     <xs:attribute name="name" type="xs:string" use="required"/>
     <xs:attribute name="nodes" type="xs:string" use="optional"/>
     <xs:attribute name="processors" type="xs:string" use="optional"/>
     <xs:attribute name="memory" type="xs:string" use="optional"/>
     <xs:attribute name="runtime" type="xs:string" use="optional"/>
     <xs:attribute name="priority" use="optional">
          <xs:simpleType>
               <xs:union memberTypes="variableRefType">
                    <xs:simpleType>
                         <xs:restriction base="xs:string">
                              <xs:enumeration value="normal"/>
                              <xs:enumeration value="high"/>
                              <xs:enumeration value="low"/>
                              <xs:enumeration value="development"/>
                              <xs:enumeration value="whenever"/>
                         </xs:restriction>
                    </xs:simpleType>
               </xs:union>
          </xs:simpleType>
     </xs:attribute>
</xs:complexType>
<xs:complexType name="hierarchicalProcessType">
     <xs:sequence>
          <xs:element ref="processReference"/>
          <xs:element name="hierarchicalReference" type="ProcessReferenceType"/>
          <xs:element ref="commandPath" minOccurs="0"/>
          <xs:element name="fileTransferDeploy" type="FileTransferStructureType" minOccurs="0"
               maxOccurs="1"/>
          <xs:element name="fileTransferRetrieve" type="FileTransferRetrieveStructureType"
               minOccurs="0" maxOccurs="1"/>
     </xs:sequence>
     <xs:attribute name="class" use="required">
          <xs:simpleType>
```

```
                <xs:restriction base="xs:string">
                     <xs:enumeration
                          value="org.objectweb.proactive.core.process.ssh.SSHHierarchicalProcess"/>
                     <xs:enumeration
                          value="org.objectweb.proactive.core.process.ssh.RSHHierarchicalProcess"/>
                </xs:restriction>
           </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
      <xs:attribute name="username" type="xs:string" use="optional"/>
      <xs:attribute name="hostname" type="xs:string" use="required"/>
      <xs:attribute name="internal_ip" type="xs:string" use="optional"/>
 </xs:complexType>
 <xs:complexType name="NGProcessType">
      <xs:sequence>
           <xs:element ref="environment" minOccurs="0"/>
           <xs:element ref="processReference"/>
           <xs:element ref="commandPath" minOccurs="0"/>
           <xs:element name="fileTransferDeploy" type="FileTransferStructureType" minOccurs="0"
                maxOccurs="1"/>
           <xs:element name="fileTransferRetrieve" type="FileTransferRetrieveStructureType"
                minOccurs="0" maxOccurs="1"/>
           <xs:element name="ngOption" type="NGOptionType" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="class" type="xs:string" use="required"
           fixed="org.objectweb.proactive.core.process.nordugrid.NGProcess"/>
      <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>
      <xs:attribute name="hostname" type="xs:string" use="required"/>
      <xs:attribute name="queue" type="xs:string" use="optional"/>
      <xs:attribute name="jobname" type="xs:string" use="optional"/>
 </xs:complexType>
 <!--globusOption element has one child count element-->
 <xs:complexType name="NGOptionType">
      <xs:sequence>
           <xs:element name="executable" type="FilePathType" minOccurs="0"/>
           <xs:element name="count" type="PosintOrVariableType" minOccurs="0"/>
           <xs:element name="outputFile" type="xs:string" minOccurs="0"/>
           <xs:element name="errorFile" type="xs:string" minOccurs="0"/>
      </xs:sequence>
 </xs:complexType>
 <!-- mpiProcessType -->
 <xs:complexType name="mpiProcessType">
      <xs:sequence>
           <xs:element minOccurs="1" ref="commandPath"/>
           <xs:element minOccurs="1" name="mpiOptions" type="mpiOptionsType"/>
      </xs:sequence>
      <xs:attribute fixed="org.objectweb.proactive.core.process.mpi.MPIDependentProcess"
           name="class" type="xs:string" use="required"/>
      <xs:attribute name="mpiFileName" type="xs:string" use="required"/>
      <xs:attribute name="hostsFileName" type="xs:string" use="optional"/>
      <xs:attribute name="mpiCommandOptions" type="xs:string" use="optional"/>
 </xs:complexType>
 <!--mpiOptions-->
 <xs:complexType name="mpiOptionsType">
      <xs:sequence>
           <xs:element minOccurs="1" name="processNumber" type="xs:string"/>
           <xs:element minOccurs="1" name="localRelativePath" type="FilePathType"/>
           <xs:element minOccurs="0" name="remoteAbsolutePath" type="FilePathType"/>
      </xs:sequence>
 </xs:complexType>
 <!-- end mpiOptions-->
 <!-- end of mpiProcessType -->
 <!-- dependentProcessSequenceType -->
 <xs:complexType name="dependentProcessSequenceType">
      <xs:sequence>
           <xs:choice minOccurs="1">
                <xs:element name="serviceReference" type="ServiceReferenceType"/>
                <xs:element ref="processReference"/>
           </xs:choice>
           <xs:element maxOccurs="unbounded" ref="processReference"/>
      </xs:sequence>
      <xs:attribute fixed="org.objectweb.proactive.core.process.DependentListProcess" name="class"
           type="xs:string" use="required"/>
 </xs:complexType>
 <!-- end of dependentProcessSequenceType -->
 <!-- independentProcessSequenceType -->
 <xs:complexType name="independentProcessSequenceType">
      <xs:sequence>
           <xs:element maxOccurs="unbounded" ref="processReference"/>
      </xs:sequence>
```

```
          <xs:attribute fixed="org.objectweb.proactive.core.process.IndependentListProcess"
               name="class" type="xs:string" use="required"/>
     </xs:complexType>
     <!-- end of independentProcessSequenceType -->
</xs:schema>
```