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.02 – Proposals for a Grid Component Model

Due date of deliverable: November 30, 2005
Actual submission date: February 15, 2006

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

1 Summary

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 Virtual Institute. By defining the GCM, the Virtual Institute aims at the precise specification of an effective Grid Component Model.

The features are discussed taking Fractal as the reference model. As explained in the text, the features are defined as extensions to the Fractal specification in order to better target Grid infrastructure: mainly virtual nodes, collective interfaces, dynamic controllers, and autonomic components. The Virtual Institute actually expects several different implementations of the GCM, not necessarily relying on existing Fractal implementations.

1.1 A Programming Model for the Grid

The Virtual 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 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 first proposal for a Grid Component Model is to define the main features to be included in the GCM, as currently assessed in the Programming Model Virtual Institute.

1.2 Challenges, Requirements, and Main Characteristics 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.
- Semantics must be defined, precisely modelling 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.

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.

1.3 Main Technical Contributions of this Proposal

The proposal for the definition 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.
- *Abstract Component Model and Architecture:* this document presents the basis for defining an *abstract view* of the Grid Component Model. 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 joint 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 in this document 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 away 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.*

1.4 Next Objectives and Future Works

The long term objective of the Programming Model Virtual Institute concerning the GCM is to finally provide the specification of the GCM standard. Such an objective necessitates further refinements, discussions inside the Virtual Institute, and collection of requirements expressed by other Virtual Institutes.

Next phase of the development of the GCM standard will be the preparation of the technical specifications of the GCM. Based on the specifications of Fractal (ADL), but also the technical specifications of other components models like CCM, the long term objective of this work is to specify the ADL, the API, and all the necessary technical details of the GCM. This specification should be precise enough to allow interoperability of GCM components, but general enough to allow almost all the existing component programming methodologies to be implemented as GCM components.

2 Introduction

This document is a proposal for a standard Grid Component Model that will be defined by the Programming Model Virtual Institute. Its first aim is to identify the required features for a Grid Component Model, deduced from the specificities of Grid computing, and the kind of components we want to address. In order to fulfill those requirements, we first describe the rationale for the GCM and then describe the core features of the proposed GCM framework.

The general strategy adopted by the Virtual 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 Virtual 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 Virtual 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 fulfil the requirements in the final specification.

This document is organized as follows. First, Section 3 presents the context in terms of Grid computing, component models and existing approaches for distributed and Grid component models. Then, in Section 4, we present the very general objectives and constraints that are crucial for our *Grid Component Model (GCM)*. Section 5 presents the rationale for the need for a Grid specific component model. We focus in this section on the aspects that become crucial in the context of Grid computing and explain how this can be realized by extending the Fractal component model. However, we expect that GCM implementations not relying on existing Fractal implementations will be developed. Then, Section 6 specifies the Grid Component Model. Finally, Section 8 summarizes the required features for a well-designed Grid component model, together with the solution proposed by the GCM to implement those features.

3 Context

3.1 Grid Specific Problematics

Grid computing aims at providing transparent access to computing power and data storage from many heterogeneous computers in different places - this is also called *virtualization* of resources. Component based programming may be used in this context for the same reasons than it is useful in a less heterogeneous and distributed context: clarity in design and easier understanding of the design, and increased reusability of software. Component frameworks, however, have to handle the crucial features of Grid computing:

- **multiple administrative domains**: the resources can be spread around many different networks, each with their own management and security policies
- **distribution**: resources can be physically distant from each other, resulting in higher, and sometimes unpredictable, network latency
- **heterogeneity**: unlike cluster computing, where the resources are homogeneous, Grids gather resources from multiple hardware vendors, running on heterogeneous operating systems, and relying on different network protocols
- **dynamicity**: configuration may change at runtime, due to environmental changes (for optimized performance, or in case of failures), or by the addition of new resources while the application is running.
- **legacy software**: in order to reuse already existing and optimized software, legacy software should be wrapped, enabling integration in broader systems
- **complexity**: Grid applications can be complex since in addition to the complexity of highly specialized pieces of software they address issues of integration, configuration and interoperability
- **high performance**: components frameworks should be designed for efficiency, notably by offering parallel programming facilities.

Several discussion about core features for Grid components can be found in the literature, for example in [36, 37].

Most of these features concern both Grid applications and Grid middlewares. The following of the document will show how the GCM aims to achieve these goals.

3.2 State of the Art

3.2.1 Component Models

Let us first focus on two commonly known models for a component-oriented approach [35] to distributed computing : the *Common Component Architecture (CCA)* [19, 21] and the *CORBA Component Model (CCM)* [29].

- *CCA* has been defined by a group of researchers from laboratories and academic institutions committed to defining standard component architectures for high performance computing. The basic definition of a component in CCA states that a component "is a software object, meant to interact with other components, encapsulating certain functionality or a set of functionalities. A component has a clearly defined interface and conforms to a prescribed behavior common to all components within an architecture." Currently the CCA Forum maintains a web-site gathering documents, projects and other CCA-related work (www.cca-forum.org) including the definition of a CCA-specific format of component interfaces (Babel/SIDL – SRPC Interface Description Language) and framework implementations (Ccaffeine)
- *CCM* is a component model defined by the Object Management Group (OMG), an open membership for-profit consortium that produces and maintains computer industry specifications (e.g. CORBA, UML, XMI, ...). The CCM specifications include a Component Implementation Definition Language (CIDL);

the semantics of the CORBA Component Model (CCM); a Component Implementation Framework (CIF), which defines the programming model for constructing component implementations and a container programming model. Important work has been performed in order to turn the CCM in a Grid component model (e.g. GridCCM).

We observe from the features available in the current implementations of CCM, CCA and other component models for the Grid that most current projects exchanging data, executable code or both across network boundaries use a portable format. Thus, the underlying technologies XML, Java and Web Services should be supported by a future Grid component model.

More generally, recent years have seen component technology becoming an important software construction technique. Standard component models such as EJB (Enterprise Java Beans) [28] and CCM [29] are now used in industry at production level. However, such components fall short of addressing all the Grid issues. Even if components can fit into a distributed infrastructure, a single component is not itself distributed per se. This means that a single component cannot be used to manage the complexity of a computation spanning several computers. Moreover, business components do not currently take into account the underlying Grid resources: they do not take advantage of all available resources, nor such constraints as the multitude of deployment protocols and dynamic distributed security mechanisms.

Recently, the US initiative CCA brought together a number of efforts in component-related research projects, with the aim of developing an interoperable Grid component model [5, 19] and extensions for parallelism and distribution [8]. The participants of the CoreGrid Programming Model Virtual Institute are closely following this activity [22]. We believe there are several reasons why we should go for a new and independent Grid component model springing from the European community. First of all, the CCA model is non-hierarchical, thereby making it difficult to handle large configurations such as those needed in the Grid (several thousands of machines). Such a hierarchical approach to component systems, is indeed one of the specificities of the Fractal component model; as it is the starting point for the GCM, this component model is dedicated a special position in this state of the art (see 3.3). In addition, the CCA model is rather poor with regards to managing, when needed, components at execution time. This makes it hard to realize certain features, for instance, dynamically reconfiguring components based on observed performance or failures. The CCA activity seems to be somewhat US-centric, with difficulties for European or world-wide component researchers to influence and incorporate their contributions and requirements.

3.2.2 Grid Middleware

Since Grid components must be deployed on Grid infrastructures, integration with Grid middleware is essential.

Currently, at scheduling and deployment levels, much Grid middleware is available. Besides the Globus / OGSA (Open Grid Services Architecture) / WSRF (Web Services Resource Framework) toolkit and architecture, the standards network protocols (i.e. ssh, rsh), the open or proprietary job schedulers (LSF, OPBS, PBS, Sun Grid Engine, etc.), there are the integrated European-financed software ranging from system level middlewares like Unicore, EGEE gLite, up to application level ones like GridLab GAT (Grid Application Toolkit), and ProActive. The compatibility and interoperability of those tools is, at the moment, rather poor. A general aim of the proposed work will be to achieve strong portability of the component framework being developed.

To conclude this state-of-the art review, it is useful to clarify the relationship between components and Web or Grid Services. While the latter is a fundamental aspect of Grid infrastructure, especially at the level of finding services to carry out scheduling, allocation, deployment etc., they will not replace the need for building applications using off-the-shelf components – maybe for the unique reason that, at some level, the integration must occur at code composition time, not only at deployment time, hence the need for a well-defined component model. Nevertheless, it is clear that a component should provide the means to automatically expose a Web Service interface and integrate Web Services, in order, for instance, to facilitate the seamless integration of components into a service-oriented and dynamic infrastructure.

3.3 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 [16, 9, 17] 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 – content – and non-functional – controllers – aspects.

A brief summary of the Fractal specification is given in Section 6.2, but the reader should refer to the Fractal specification [17] for a complete description of the Fractal model. similar to most component models, Fractal uses an object-oriented terminology [34, 27]. 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. 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), 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 6.2 will recall those levels, called conformance levels in the Fractal specification.

4 General Features

The GCM is the component model adapted for the grid that the Programming Model Virtual Institute proposes. It should possess some basic and essential characteristics.

- Support for *reflection*: Both introspection and intercession;
- *Hierarchical structure*: Facilitate the identification of architectural units, thereby facilitating the design of complex systems. Grid systems are usually composed of many different and heterogeneous software units. Each software unit may itself integrate other software units. For example a unit responsible for data-mining may itself be constituted of several units running on a given cluster,

and this data-mining unit may itself be part of a larger system, like a weather prediction system;

- Model *Extensibility*: The model should include placeholders to allow for new features to be added to components and the component model;
- *Language neutrality*: GCM should be specified in a programming language independent way;
- Support for *adaptivity*, that is provide automatic ways of adapting running GCM programs to the features required/provided by the target Grid and component architecture;
- *Interoperability*: GCM should allow components to export and import functionalities from other frameworks; for instance, one should be able to use a Web Service or CCM from within the GCM as well as being able to export a Web Service port for any GCM component.

Additionally the model should be *lightweight*, in order to allow compact and portable implementations to be designed, and have a *well defined semantics*. Moreover, the GCM should take into account the necessity for its implementations to ensure high performance.

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.

5 Rationale for a Grid Component Model

The definition of the component model we propose is based on Fractal concepts and terminology. In particular we will use many of the notions defined by the Fractal specification. For example hierarchical composition of components follows the one defined in Fractal. We will also separate functional and non-functional interfaces. We wish to provide introspection and intercession as defined in Fractal, and we will rely on a type system, which will be an extension of the one suggested in the Fractal specification.

As in Fractal, we will use in this section and in Section 6 an extension of the graphical representation for components that is already used in Fractal. In this document, this graphical representation is here to illustrate the concepts presented and facilitate the reading of the document. However, Fractal components representation can be used to design component systems; thus, one can expect that an adaptation of the representation used in this document will be used in the future as an abstraction for creating real applications.

We will focus below on those features which we consider to be important in a Grid component platform, and in particular discuss to what degree they are supported by Fractal.

Implementing the features presented in this section in a component framework ensuring the general features of Section 4, will lead to a well-designed component model solving the Grid-specific problems presented in Section 3.1.

5.1 Communication

Generally, one of the challenges in Grid computing is mastering of network latency. The Fractal model provides means to overcome this challenge, because it supports all kinds of communication pattern. In addition, binding components can implement sophisticated message-passing mechanisms, such as publish/subscribe or pulling. For instance, the Dream framework implements complex communications as composite bindings [1]. Also notification mechanisms, built upon asynchronous communications, are useful when realizing compositions in time (workflows), where the logic is driven by a sequence of events.

Asynchronous communications is one of the communication patterns that is particularly useful in mastering the latency. It can even be realized without binder components, thereby avoiding unnecessary intermediate components. This is the case, for example, in the implementation of Fractal with ProActive[7], where inter-component communications can be asynchronous and offer data-based synchronization capabilities (wait-by-necessity, first class futures).

However, ProActive implements asynchronism with rendezvous to guarantee ordering of messages. More generally, different communication paradigms should be implemented depending on the guarantees to be ensured and on the efficiency one wants to achieve. Thus, even if asynchronism seems, in general, the most efficient communication paradigm, asynchrony is achieved very differently in different frameworks, depending on the assumptions and guarantees of the model.

Thus, it seems impossible to decide on a simple communication paradigm for component communication over the Grid, which implies that the kind of communication implemented/accepted by each interface of a component should be specified in the interface type, 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.

The port semantics for some predefined (and classical) ways of communication can be predefined, but such a semantics should be extensible.

The envisaged communication paradigms include synchronous, purely asynchronous, asynchronous with rendez-vous, deferred mode, asynchronous method invocation with futures, CCM events, ASSIST [3] data-flow streams, data-space oriented, and yet unknown ways of communicating (e.g., explicitly nondeterministic communication). Section 6.5 will also present a proposal for collective ports, leading to collective communications.

5.2 Parameterized Instantiation

A crucial feature for Grid components that is supported by the Fractal component model is the parameterized instantiation. Indeed, depending on the environment (execution environment, platform, application developed) the deployment of a given component should be parameterized. This relates to the functional aspect (content), and mainly the non-functional parts. Indeed, adding different controllers depending on the usage of a given component is essential. For example, when instantiating a component with a Factory, Fractal relies on a method of the form:

```
Component newFcInstance (Type t, any controllerDesc, any contentDesc).
```

This allows the parameterization of the component creation by a description of the controllers to be associated with the component, and a description of the content of the component.

5.3 Controllers

Controllers, as defined in Fractal, handle non-functional behavior of components.

As Fractal is an open model, it is possible to introduce new controllers for the management of non-functional properties. It is important to allow users of a component framework to customize the framework for their needs. Controllers could thus be defined to solve problems specific to the Grid. For instance, migration controllers for moving a computation from an overloaded site to a less loaded one, or disconnection controllers for handling the mobility on the client side. Communications between different administrative domains, although usually manageable at a lower level of the infrastructure, could also be specialized or configured on a component-by-component basis using custom controllers.

The ability to add or trigger controllers dynamically is also a key requirement for achieving adaptivity of the component model. Fractal does not explicitly specify that controllers should be determined and configured only at instantiation time (c.f. Section 5.2), and Section 5.6 will propose a way to provide controllers that are configurable at run-time.

Life-cycle controllers, content controllers and binding controllers as defined in Fractal are necessary to handle hierarchical composition, application components and platform components. However, they might be extended and adapted to a Grid environment.

For instance, a binding controller adapted to a Grid component system should be able to ensure, when binding components together, that they rely on compatible communication paradigms, or, if it is possible, include an adapter component.

With regard to lifecycle, the controller might be extended to include Grid-related features, for example, one might include the notion of session and transactions. However, such notions could also be defined as separate controllers.

A controller can also act as an *interceptor*: it can intercept incoming and outgoing operation invocations targeting or originating from the component's sub components. This is also useful in the context of aspect-oriented programming and the definition of transactions. As for any controller, such interceptors are dynamically configurable.

Section 5.6 will focus on a particular aspect of controllers that are, we believe, crucial when addressing the dynamicity and adaptivity of component control.

5.4 Higher-order Components and Code Mobility

One could imagine several conformance levels towards real higher-order components. Basically, for components, the term *higher-order* means that the components themselves are first class objects and can be manipulated by the application, this implies that they can be passed along any binding (as parameters of service invocation).

This could be partially simulated by parameterizing the instantiation of a component either by some executable code, or by other components. A step further consists in dynamically changing the component configuration (acting on binding and content controllers). The various ways of acting on the component system topology results in the following classification into four levels:

- First, 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 [14]. On the Grid, such a functionality requires the ability to carry code and transmit it through some component bindings.

- Second, parameterized instantiation as supported by Fractal, allows, at instantiation time, the initial content of a composite component to be expressed as a set of component instances;
- Third, Fractal content controllers allow the dynamic addition or removal of components from a composite component, combined with the usage of binding controllers; this allows the topology of a component system to be dynamically changed but relies on a meta-level;
- Finally, real higher-order means components of any type can be passed along any binding, thus triggering uncontrolled (in the sense of not being managed by a controller) dynamic reconfiguration. This means code mobility, not only in a physical sense but primarily in a logical way; contrary to level three, this would be triggered at the application level, not at a meta-level. Note that, this fourth level cannot be seen as a natural or direct extension of the level three. This level requires the definition of a semantics for passing components as parameters (copy vs.reference, sharing vs. instantiation, . . .).

The three first levels correspond to facilities at the meta level only (i.e. controllers). The fourth level provides higher-order capabilities to the application level itself (real higher-order), allowing a greater expressivity.

Inherited from Fractal, the GCM provides support for the three first levels; real higher-order components can be defined in the future as an extension of the GCM.

In this version of the GCM, via dynamic controllers, we make the third level above powerful enough to express all the higher-order features that seem necessary.

Having the three first levels, one could implement real higher-order components by *intercepting* communications between components and triggering adequate binding/content control operations to simulate the passing of components along the bindings.

5.5 Interoperability

Any realistic programming model has to take into account the need to interoperate with existing, legacy software. As such, GCM needs to interoperate with existing component frameworks and allow applications to be built from a mixture of various kinds of components.

From the legacy code point of view, several approaches exist, as for example [12] generally consisting in encapsulating legacy code into a single component or a single Grid-service. At the opposite, the GCM should allow a much deeper interoperability: GCM component should be able to control any other component, and to interact with it. In other words, any GCM component should be able to trigger non-functional actions of other GCM components, but also to achieve complex communication patterns, involving requests-replies, streaming, etc. Indeed, those communication patterns are already used between components of the same platform, thus real interoperability between GCM components necessitates the availability of such communication patterns.

As stated in [24], multi-language interoperability is difficult in the general case, and would require manual or semi-automatic generation of wrappers or adapters. However for a specific programming model, with specific constraints and with well-defined APIs, this goal is easier to achieve. In the GCM, APIs will be defined in order to maximize interoperability within a given language, and Web-service interfaces will be provided for full interoperability, even when dealing with very different implementations of the GCM.

Two different issues have to be solved with respect to *framework interoperability.*

- Devising a proper mapping of the GCM interfaces over those of the other framework, both for exporting GCM components and applications, and for importing alien components into GCM compositions of components.
- Developing a GCM run-time and non-functional interfaces which allow the components and the deployment system to select the appropriate communication support and to translate interfaces crossing the framework boundary.

Obvious candidates for GCM interoperability are Web Services, as a framework already available on Grids, the Corba Component Model (CCM), and the Common Component Architecture (CCA), a component framework which has several different implementations targeted at parallel and distributed systems.

5.5.1 Web Services Interoperability

Web Service interoperability is essential: in order to conform to industrial standards, we need to be able to export a WSDL for any components.

Of course, GCM components also need to be able to use existing Web Services thus importing and interacting with existing Web Services is also a required feature for the GCM. Exposition of GCM components as Web Services can be inspired, for example, by the integration of Fractal with HOCs as proposed by WWU and INRIA [15].

In any case, the GCM will have to circumscribe the compatibility with Web Services. For this, we first need to answer the question: "By how much will WSDL compatibility limit/influence the component model used to develop applications?": for example, can any type existing in a GCM interface be mapped to a type in a WSDL description?

We propose to use Web Services as the standard for interoperability between GCM components, and with external frameworks. However, Web Services is not the only way to implement interoperability and the GCM must allow interoperability with other frameworks to be implemented.

5.5.2 Multi-language Interoperability

Interoperability issues come into play if we consider multi-language support within the GCM framework. This is a complex issue that requires either an interface definition language for GCM (this is the approach adopted within CCA using Babel [32] to generate mediators and adapters), or to exploit framework interoperability and a language-neutral framework as a language mediator (e.g. Web Services, CCM, CCA) and inherit an existing IDL (WSDL, IDL3 or Babel itself, respectively). The former solution is the most difficult to implement, while the latter has to be evaluated with respect to the amount of features and performance that can be lost at the framework interface.

Relying on Web-services or other mediators in the general case does not prevent GCM from using manually or automatically generated glue components (or wrappers) when suitable. Babel is a good example of such glue components. These adapters will rely on existing IDL and API for the GCM, but they will not be defined within GCM.

5.6 Adaptivity

Parallel and Grid computing platforms allow us to run programs exploiting a number of different resource configurations, the most trivial example being the degree of parallelism we choose to exploit. Adaptivity means that a component, or a component-based application, is able to run and change its configuration over time, dynamically adapting to a specific need or a series of events. Three major issues call for adaptive reconfiguration on Grids:

- changing behaviour of the computing resource/platform,
- the presence of resource/program faults,
- changing application needs, either self-detected (autonomic control) or user-controlled (computation steering).

In all cases the basic operation the component(s) performs is a reconfiguration, that is a change in the logical or physical structure of the application. A reconfiguration consists either in a change in the set of used resources, or in a change in the set of used components, or in the dependencies (bindings) between existing components. A natural prerequisite for a correct reconfiguration is the preservation of the semantics of the ongoing computation.

Reaction to dynamic resource behaviour Reconfigurations can help the system to adapt to its environment, which may be evolving over time: for example, a cluster may be overloaded, in which case the work would better be done somewhere else, or an end user may want to change the configuration during runtime to obey some external condition (e.g. the user wants to put more computational power to get faster or more accurate results).

Reaction to faults Adaptive reconfiguration is useful when fault tolerance is provided, to restore the required amount and/or allocation of resources or set of components for the running application.

When a system comprises many machines (Grid systems range from a few to several thousand hosts), and when computations can last for days, failures must be considered. These failures are not bugs in the application, but rather bugs of the underlying operating system, hardware faults, computing resources becoming unavailable due to network failures. In some cases, parts of the system become unavailable without any failure, for example the connection to some clusters is lost, or all the machines were busy, the jobs cannot be run in the place and time they were scheduled, and a timeout is reached, or simply a laptop is unconnected. The system must be capable of reorganization to cope with these kinds of disruption in the accessibility of some components.

Fault detection and handling is a separate issue from adaptivity. However, recovery generally relies on adaptivity features in order to react to faults, e.g. automatically reconfigure parts of the system if necessary. Moreover, once the components have reacted and corrected the fault, or as a consequence of the additional computation needed to handle each fault, program configuration in the general case will be sub-optimal and will have to be adapted.

Reaction to program changes Adaptivity is useful not only in reacting to the behaviour of resources, but also to the behaviour of the program itself. There is a large class of algorithms whose computational cost depends on the kind/size/values of data and parameters (e.g. a user steering a simulation), and systems whose QoS has to remain within specified bounds no matter what amount of input is provided (e.g. computation servers receiving streams of tasks to process).

More generally, program changes may also concern services provided and required by the application. Indeed, the set of services a component application needs may evolve over time, and it is necessary to adapt the component system to these dynamically changing requirements.

To cope with all of these different kinds of events, the system must be capable of reconfiguration. Beside resource-related reconfiguration, if components can be moved or reconfigured, they necessarily have to be adapted to their new environment. This can mean adding or removing some of their non-functional behaviors.

The dynamic capabilities of the Fractal model can help tackle these kinds of issues. The ability to reconfigure a running component can be exploited either by user intervention or by an autonomic control, which reacts to a triggering condition in such a way to enhance the component configuration.

Autonomic control of adaptivity will be detailed in Section 6.7 with respect to the implications on component interfaces. However, we must underline here that autonomic behaviour relies on adaptivity management.

When devising a general model of the ongoing works for handling adaptivity in parallel and distributed programs, it is natural to pursue abstraction with respect to implemented adaptation techniques, monitoring infrastructure and reconfiguration strategies. Higher-level actions will rely on lower-level actions and mechanisms, and autonomic functionalities will rely on adaptive reconfiguration [4]. On a coarse description level, the difference between adaptive and autonomic behaviour is in the presence of managing code that generates reconfiguration requests from events, such events being collected autonomously by the component itself.

Adaptivity is therefore a strong requirement. We want to exploit component hierarchical abstraction for adaptivity [2]. A solution to implement this would be to provide local managers for component-level adaptation together with a hierarchy of manager components. Those aspects will interact by well-defined external/internal server and client interfaces for non-functional aspects (e.g., configuration, monitoring, deployment). These interfaces will have the capability to be added, plugged and unplugged dynamically.

As a consequence, GCM needs to allow adding, plugging, and unplugging controllers dynamically, thus allowing the dynamic addition or removal of some management/non-functional aspects to any component. We need to generalize Fractal controller interfaces for handling these aspects (e.g. consider controllers as components as proposed in [26]).

This is especially true when high-level programming environments are able to automatically generate code implementing adaptivity control policies and autonomic behaviour management. In all cases the code generated has to be plugged into components, which must provide specific external and internal interfaces for that purpose, but management code behaviour is completely external and an orthogonal issue w.r.t. the GCM definition.

Our approach to adaptivity has some strong similarities with the study of aspect oriented programming for components architectures [11, 18, 31]. Indeed, dynamic management of aspects also requires the ability to bind, unbind and add managing code dynamically.

As suggested as an extension of Fractal in the Fractal specification, the solution that will be detailed in Section 6.6 considers controllers as Fractal components, allowing us to bind and unbind them using the `BindingController` and add them using the `ContentController`. Section 6.6 presents a proposal for such dynamic controllers as part of the GCM.

5.7 Parallelism, Parallel Interfaces

One of the objectives is to support both sequential and parallel implementation for a given component. This means supporting several alternatives and associated meta-data. For example, one can rely on an external (platform) component to decide which implementation of a given component should be used.

Parallelism for components is covered under a number of headings.

A *parallel component* is a component that executes a parallel code (in the sense of several activities – not necessarily identical – running in parallel). In this sense, most of the GCM composites are parallel components, as they will embrace several activities (running in parallel). These activities can, in general, be deployed on several machines.

Very often in Grid computing, we also need to have components running *identical* operations in parallel. A *parallel service component* is a component comprising several *identical* entities running in parallel: sub-components in the case of a composite; hidden parallel code (threads) in the case of a primitive.

Parallel interfaces - also called *collective interfaces* in this document - allow interaction with parallel service components in an efficient, customizable, and adaptive way. The GCM proposes two main kinds of parallel interfaces: multicast and gathercast interfaces.

Note that parallel interfaces are not mandatory, and it is possible either not to use parallel interfaces at all, or to encapsulate parallel interfaces inside a component: hierarchical composition plays a crucial role here. In both cases, parallel components do not have explicit parallel interfaces, thus allowing safely exchangeable sequential/parallel components.

Parallel interfaces are particularly adequate to express SPMD-style programming.

5.7.1 Deployment of Parallel Components

As a parallel component can run on several machines, we need to be able to express the distribution of the activities constituting the component over the Grid. Locating and recruiting resources is a much more complex issue for parallel, Grid-distributed components than it is for sequential components.

We do not want the user to deal with details of deploying parallel components over heterogeneous parallel resources, either in the component code itself or at the application level, because this violates the encapsulation requirement of components. This is especially true as any GCM component can have several different parallel and sequential implementations, and they can be safely replaced with each other dynamically.

Component deployment will need some form of component packaging that includes an architectural description of the component in an appropriate Architecture Description Language (ADL). The description of a parallel component will refer at least to a set of independent activities (threads, processes) that, from an abstract point of view, are deployed on a number of virtual nodes/processors.

Starting from a description of a component[1] and a user objective function, the deployment process is responsible for automatically performing the steps needed to start the execution of the component on a set of selected resources.

A framework for the (automatic) deployment execution of applications is composed of several interacting entities in charge of distinct activities (Submission, Resource Discovery, Resource Selection, Planning, Enactment, Execution). For those steps that are automatically performed, the user may want to specify an objective function (e.g. to choose faster or cheaper computing resources), whose specification is part of the deployment system.

Whatever the actual implementation of the deployment system, it uses a common description of a component structure and of the kind of resources the component needs.

[1]The extension to component-based application as graphs of components is straightforward.

From the point of view of the execution, a component contains a structured set of binary executables and requirements for their instantiation. Such information can be used to generate deployment plans for GCM components

- to deploy sequential and parallel components using a common interface and run-time,
- to deploy components in a multi-middleware environment,
- to dynamically alter a previous configuration, adding new computational resources to a running application,
- for re-deployment, when a concrete component is dynamically substituted with a different one, and when a complete restart from a previous checkpoint is needed due to severe performance degradation or failure of several resources.

Note that the deployment process does not need to exploit all the parallelism specified within a component description. As long as the minimum requirements specified within the component package are met (e.g. kind of architecture, available memory and so on) a component comprising several virtual processes can be deployed in practice on a smaller set of resources, possibly on a single machine.

5.7.2 Parallel interfaces: Multicast, Gathercast

Managing the communications between the components of a Grid application implies managing interactions to and from several components, sometimes in a parallel fashion. This implies that a component model for the Grid should include multiway [2] and parallel programming facilities. We propose to express multiway bindings simply by exposing the collective nature of component interfaces, and introduce new cardinalities for Fractal interfaces: *collective* cardinalities. Collective interfaces are specified as a *collective* cardinality in the type of the component interfaces, and their behavior is customizable.

Currently, the Fractal model defines two kinds of cardinalities for interfaces: single and collection. Interfaces of cardinality *single* are unique and exist at runtime. Single interfaces can have a server role (Fig. 1.a), or a client role (Fig. 1.b). Fig 1.c and Fig 1.d show respectively a single server interface and a single client interface for primitive components. In the case of composite components, there is a complementary internal interface of opposite role associated with each external interface: Fig 1.e shows a single client interface along with its complementary internal single server interface, while Fig 1.f shows a single server interface along with its complementary internal single client interface.

Interfaces of cardinality *collection* represent collections (i.e. an arbitrary number of lazily created interfaces) of interfaces of the same type, with a common prefixed name. One can define collection server interfaces (Fig. 2.a), or collection client interfaces (Fig. 2.b). A collection interface does not exist at runtime, only the lazily created interfaces that are part of the collection are accessible at runtime (see Fig 2). In the case of composite components, there is a complementary internal interface associated with each of these lazily created interfaces (Fig. 2.e and 2.f), which is not the case for primitive components (Fig. 2.c and 2.d). In order to perform an invocation on all of the lazily created interfaces, it is necessary first to get a reference on each of the interfaces, and second perform the invocation on each of these references. In this current work, we intend to provide a mechanism for performing collective invocations without having to refer to each of the interfaces successively.

[2] We name multiway interactions the interactions to and from several components.

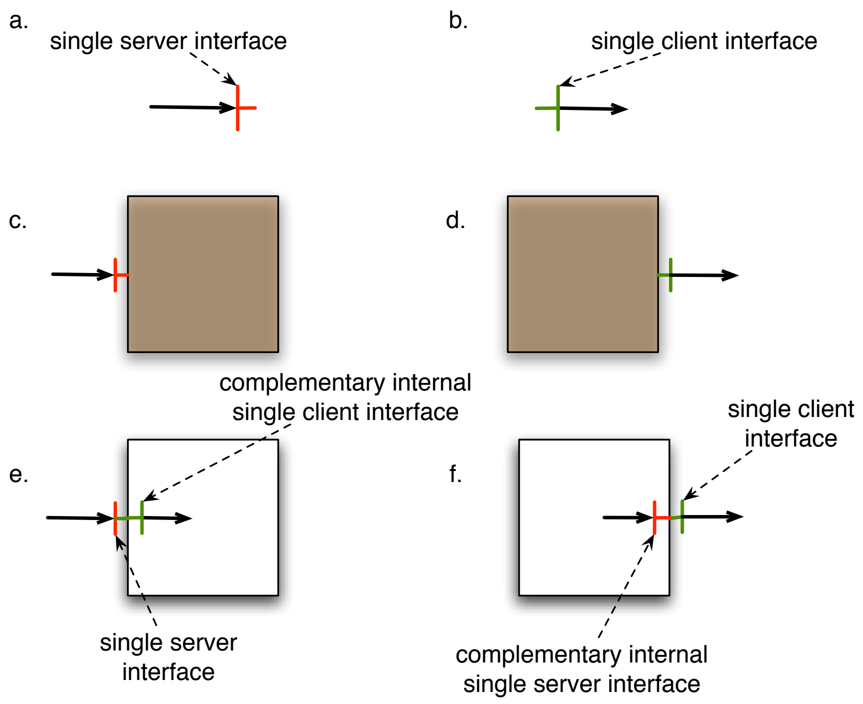

Figure 1: Single interfaces for primitive and composite components

The current cardinalities of Fractal interfaces allow only one-to-one communications. It is possible though to introduce *binding components*, which act as brokers and may handle different communication paradigms. Using these intermediate binding components, it is therefore possible to achieve one-to-n, n-to-one or n-to-n communications between components. It is not possible however for an interface to express collective behavior: explicit binding components are needed for this.

To meet the specific requirements and conditions of Grid computing for multiway communications, we propose the addition of new cardinalities in the specification of Fractal interfaces, namely *multicast* and *gathercast*.

Multicast and *gathercast* interfaces give the possibility to *manage a group of interfaces as a single entity* (which is not the case with a collection interface, where the user can only manipulate individual members of the collection), and *expose* the collective nature of a given interface. Moreover, specific semantics for multiway invocations can be configured, providing users with flexible communications to or from gathercast and multicast interfaces. Lastly, avoiding the use of explicit intermediate binding components simplifies the programming model and type compatibility can be automatically verified by the framework.

The role and use of multicast and gathercast interfaces are complementary. Multicast interfaces are used for parallel invocations, whereas gathercast interfaces are used for synchronization, gathering or redispatching purposes. We also introduce *gather-multicast* interfaces, which combine the capabilities of multicast and gathercast interfaces. More information may be found in Section 6.5.

5.8 Component Packaging and Deployment

Reusable components available from repositories is an essential feature for building complex applications. Unfortunately this feature is not available yet for Fractal,

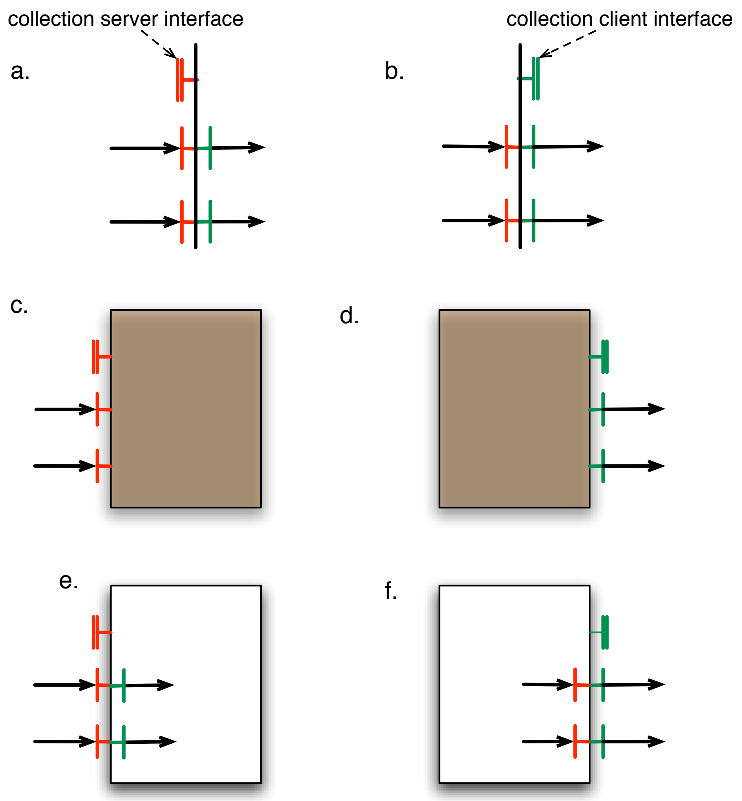

Figure 2: Collection interfaces for primitive and composite components

although some discussions were initiated on the Fractal mailing list. Packaging should provide component encapsulation (of compiled code) and description (ADL), as well as versioning capabilities. In the Fractal repository, there is a packaging tool that sketches these yet to be provided features (encapsulation, description and versioning), but it is still in alpha state. A packaging standard is being developed by the Fractal community.

ADL with support for deployment The purpose of an ADL for the GCM is twofold:

- to provide component structure as defined by a Standard ADL, e.g. Fractal;
- to define virtual computing topology for component deployment, cf. Proactive Virtual Nodes or ASSIST Virtual Processes.

These two aspects are partially independent: the possible virtual computing topologies depend on the structure of the component.

Then the mapping from virtual topology to the real one can be defined by a deployment descriptor provided independently. Of course, this mapping can also be dynamically discovered, in the way proposed in [23].

Grid applications would benefit from the availability of extensible packaging. First, extra information can be provided with a package, for example concerning an adaptation policy. Second, meta-information on executable code would be necessary to handle legacy code efficiently.

6 Proposal for the GCM Definition

We define in this section the Grid Component Model. This component model relies on the Fractal specification [17]. We first give an abstract view of the component model we propose, then we recall briefly the most useful characteristics of the Fractal component model. Finally, we focus on the technical aspects on which the GCM extends the Fractal component model. This extension allows, starting from the Fractal component model, the fulfillment of the objectives set out in Section 5.

Except from the Fractal specification section (Section 6.2), and the packaging, which is left undefined as the Fractal community is currently working on this aspect (Section 6.8), 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.

6.1 Abstract Component Model and Architecture

First, we present the basis for defining an *abstract view* of the Grid Component Model. 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 joint 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 schema or DTD
2. Run-Time API defined in several languages
3. Packaging described as an XML schema

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 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.

6.1.1 Component Specification as an XML Schema

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, calls for defining interfaces (various kind of ports), and the binding between those ports.

With respect to dealing with 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 definitions.

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

- 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 3: XML Component Specification

Figure 3 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.

6.1.2 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
- 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, etc.

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.

6.1.3 Packaging described as an XML Schema

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 6.6.2), like cost functions or descriptions of the component's functional behaviour at different levels of abstraction.
- etc.

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

6.2 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 [17]. Section 5 explained the main particularities related to Grid computing, and the main choices or extensions that GCM makes in relation to Fractal.

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:* 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.

Figure 4 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.

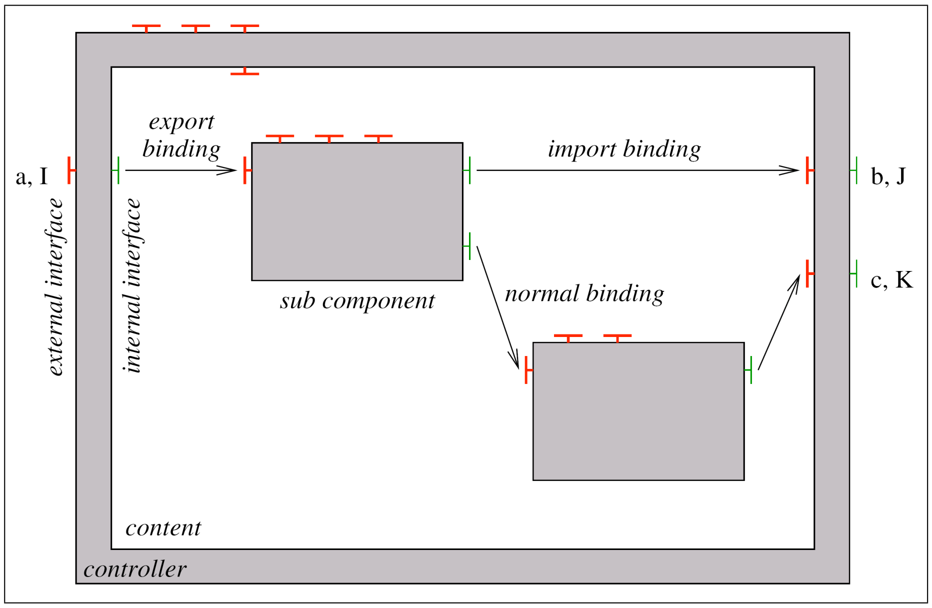

Figure 4: A composite component as defined in Fractal Specification

  - 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 A gives the Fractal API, as defined in the Fractal specification.

The DTD for the standard Fractal ADL is presented in Appendix B.

6.3 Communication Semantics

As explained in Section 5.1, we aim to support several kinds of communication in the GCM. 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 two different tags in the ADL in order to specify streaming communication, with some additional requirements:

- *Streaming push* requiring that each method of an interface implementing such a service has no return value. For instance, in Java such an interface would look like:

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

- *Streaming pull* requiring that each method of an interface implementing such a service takes no argument. For instance, in Java such an interface would look like

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

It is obviously outside the scope of the GCM to specify how streaming is implemented.

6.4 Virtual Nodes

6.4.1 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, that is 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.

We could also include general constraints, at the level of the component itself, for instance allowing to specify constraints between Virtual Nodes.

6.4.2 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 cardinality multiple.

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.

6.4.3 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 syntax for the virtual node definition, and their composition.

6.5 Multicast and Gathercast Interfaces

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*, *gathercast* and *gather-multicast*. 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.

Solutions to the problem of data distribution have been proposed within PaCO++/GridCCM [13]; 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*.

6.5.1 Multicast Interfaces

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

Definitions The following definitions characterize external interfaces (which define the type of a component).

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

On the 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 (notably the fact that results are actually lists of results), and forwards a single invocation either to a complementary multicast client interface in the case of a composite component (Fig. 5.d), or to a contractually defined implementation code in the case of a primitive component (Fig. 5.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. 5.c), or to bound server interfaces of internal components (Fig. 5.d).

> A *multicast client interface* transforms each single invocation coming from either implementation code of a primitive component (Fig. 5.e) or from an internal component (Fig. 5.f) into a set of invocations to bound server interfaces of external components.

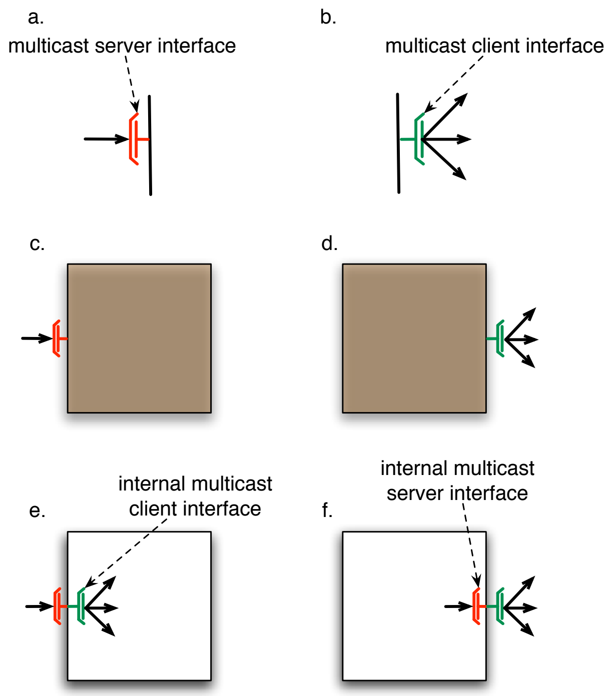
a.
multicast server interface
b.
multicast client interface
c.
d.
e.
internal multicast client interface
f.
internal multicast server interface

Figure 5: Multicast interfaces for primitive and composite 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 ([25, 30]).

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();
}
```

An extension of this proposal should include a reduction mechanism that would return one (or several) 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.

Distribution of invocation parameters If some of the parameters - of a multicast interface - 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*.

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.

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 $f \subseteq [1..k_1] \times [1..k_2] ... \times [1..k_n]$ of the combination of parameters, where f is the (multi)set of the combinations of parameters, n is the number of parameters of the invoked method which are lists of values, and $k_i, 1 \leq i \leq n$ the number of values for each list parameter. Of course, 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. f also gives the number of invocations which are generated and which depends on the configuration of the distribution of the parameters.

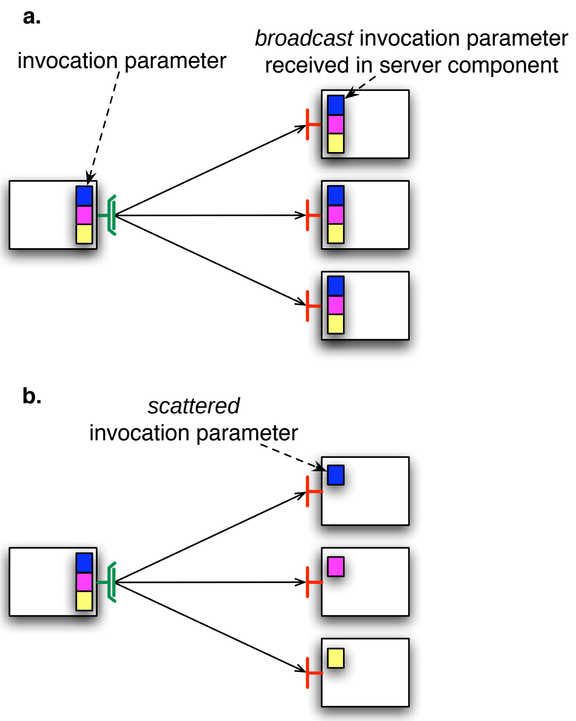
a.
invocation parameter
broadcast invocation parameter received in server component
b.
scattered invocation parameter

Figure 6: Broadcast and scatter of invocations parameters

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 to participate in the multicast.

In addition, this framework of course allows us to express naturally the case where each of the connected interfaces has to receive exactly one invocation, in a deterministic way.

6.5.2 Gathercast interfaces

Gathercast interfaces provide abstractions for many-to-one communications. Gathercast interface and multicast interface definitions and behaviors are symmetrical. [6]

Definition The following definition characterizes external interfaces.

> **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. Invocation return values are also redistributed to the invoking components.

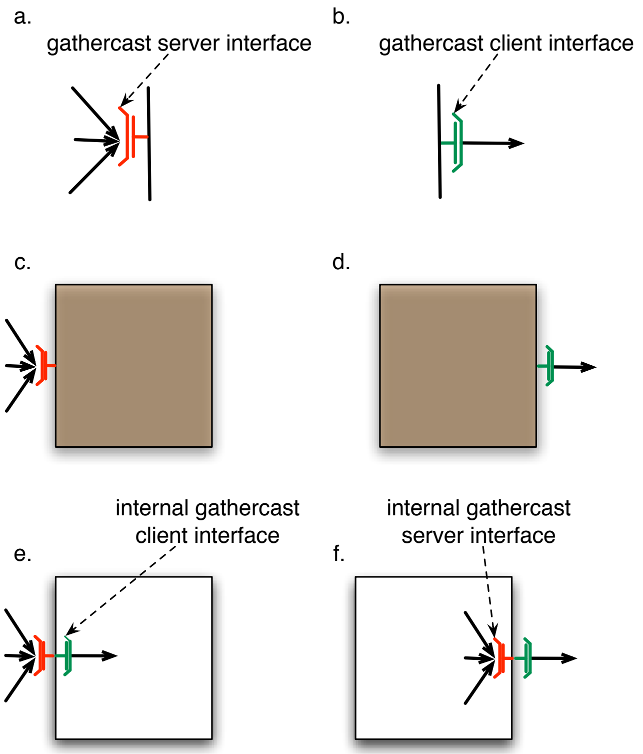
a.
gathercast server interface
b.
gathercast client interface
c.
d.
e.
internal gathercast client interface
f.
internal gathercast server interface

Figure 7: Gathercast server interfaces for primitive and composite components

Gathercast server interfaces gather invocations from multiple client interfaces (Fig. 7), 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 client interface* transforms a set of invocations coming from client interfaces of inner components (Fig. 7.f) or from the implementation code of the component (Fig.7.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. 7.d), or to the implementation code in case of a primitive component (Fig. 7.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.

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 timeout - for all its clients to perform a given invocation on it before forwarding the invocations. It is also possible to define more complex or specific message-based synchronizations, based on the content of the messages or based on temporal conditions, and it is possible to combine these different kinds of synchronizations.

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. 8).

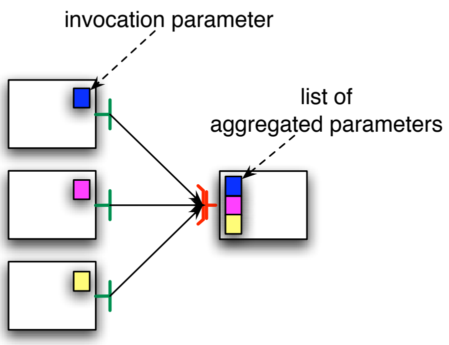

Figure 8: Aggregation of invocation parameters for a gathercast interface

Symmetrically with the case of multicast results, gathered parameters may be reduced, relaxing the constraint of having lists as parameters.

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 first question, where to specify the redistribution of results, is answered in a similar fashion 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 specified also follows reasoning 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, and is configurable through its collective interface controller.

6.5.3 Gather-multicast interfaces

> **A gather-multicast interface transforms a set of invocations into another set of invocations, redistributing both invocation parameters and results from the invocations.**

A gather-multicast interface combines the capabilities of gathercast and multicast interfaces, and behaves like a gathercast interface immediately followed by a multicast interface.

6.5.4 Collective bindings

We name a binding to or from a collective interface a *collective binding*. A collective binding is a special kind of binding. It does not use explicit binding components, which simplifies the design of component models, ensures type compatibility (and possible type conversions), and allows the designer to focus on the business logic (typed relationships between components). The multicast, gathercast and gather-multicast interfaces are configurable, which allows these interfaces to suit various situations. 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.

6.5.5 Gathercast to multicast bindings

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.

We intend to provide a specific binding process in the context of the MxN problem. This specific MxN binding process will address the two following issues:

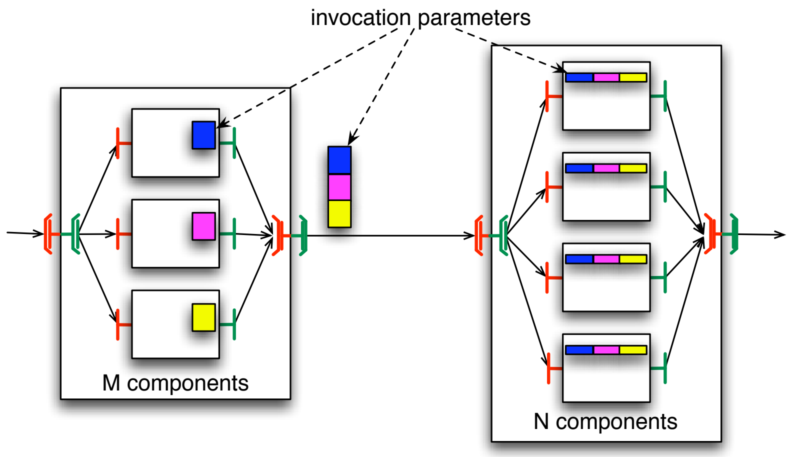

Figure 9: MxN data redistribution

1. the redistribution of the parameters (and results) of the invocations: from which components to which components? Which redistribution scheme? Fig. 9 shows an example of data redistribution.

2. the communication mode in the case of intermediate composite components: are component-to-component communications direct communications? Are communications still routed through intermediate composite components?

6.5.6 Summary: Collective Interfaces and Collective Controllers

Overall, this section presented a proposal for introducing collective communications in the Fractal model. First, Fractal proposes only collection cardinalities, thus we proposed the introduction of a new cardinality for interfaces: *collective*.

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. On the client side, 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). On the server side a multicast interface is a classical Fractal interface, but with an associated collective controller. Symmetric arguments apply to gathercast interfaces.

The constraint of using lists as parameters for gathercast invocations and lists as results for multicast invocations may be relaxed by providing a reduction mechanism which could easily be integrated in the specification of collective interfaces.

Globally, this section specified the behavior of the collective interfaces and their controllers.

6.6 Dynamic Controllers

6.6.1 Principles

Figure 10 shows the representation we suggest for implementing dynamic 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

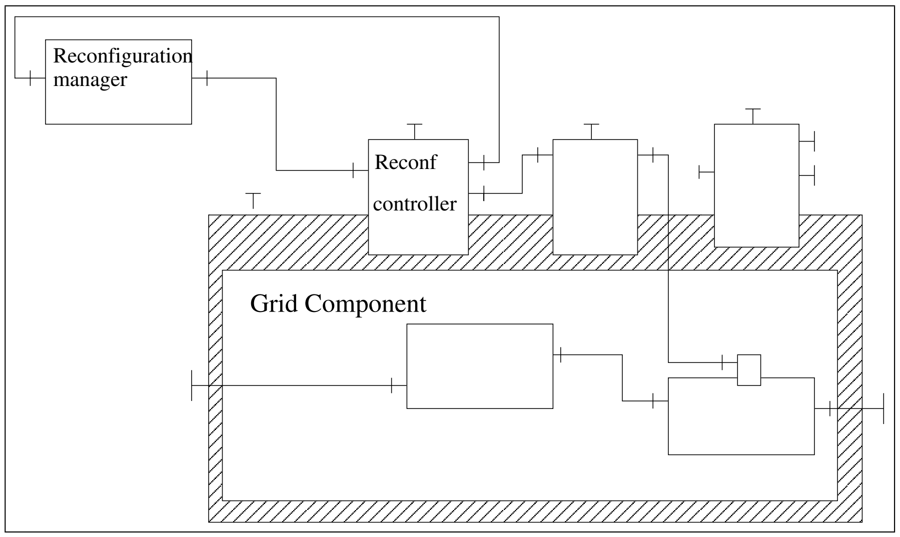

Figure 10: A composite with pluggable controllers

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 sub-component 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.

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 parallel) 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".

Recall that this approach is somewhat similar to previous work on aspect-oriented programming for Fractal [11, 18, 31].

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 [33], 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 [3].

This raises a technical question regarding Fractal and the model proposed here: in Fractal, invocation on controller interfaces *must* be enabled when a component is *stopped*; here, if controllers are implemented by components, one must specify which of these controller components have to be stopped.

6.6.2 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 or bind them to the component they control, thus achieving a high level of dynamicity and adaptivity. Note that this is not sufficient for having real higher-order components since only some *controller* components can receive components, and components can only be manipulated at the meta-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.

6.6.3 Summary: Dynamic 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

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

a binding and a life-cycle controller), and thus can be implemented in a lightweight manner, for example, by a particular kind of object.

Finally, the content controller should be extended in order to allow dynamic addition of a controller component to the membrane. This entails extending the Fractal API.

6.7 Autonomic Components

6.7.1 Introduction

To achieve better adaptivity as proposed in Section 5.6, we refine the definition of the component for autonomic computation [20, 38, 10]. We propose an approach based on the autonomic computing paradigm, that will drive us in the definition of (non-functional) interfaces exposed by a component, and their implementation.

The autonomic computing paradigm is defined by four aspects that should be implemented by each component:

- Self-Configuring: 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 controller and the content 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-Optimising: a component is self-optimising 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-Protecting: A component is self-protecting if it is able to predict, prevent, detect and identify attacks, and to protect itself against them.

  We also consider it interesting to design a framework in which some components explicitly deal with possibly malicious (sub)components, autonomically enforcing correct interaction protocols and other self-protection policies.

These interfaces are bound to (possibly distinct) dynamic controllers that implement them. However, a single controller can implement several interfaces. If a component does not provide one or more of the autonomic functionalities, then the implementations of the respective interfaces are void.

6.7.2 Autonomic Controllers

In this section we give a precise interface for each of the "four selves" introduced in the previous section. These interfaces are non-functional and exposed by each component. Consequently they correspond to Fractal *controllers*.

We consider different compliance levels also with respect to autonomicity: the highest level of compliance is that of fully autonomic components, which implement the whole set of autonomic interfaces. At lower levels of compliance, a component can implement a subset of the autonomic controllers.

Having an external control interface means that an external controller (i.e. the controller of an enclosing component, in the composition hierarchy) can implement autonomic behaviour from non-autonomic components, or it can override the autonomic mechanism of controlled subcomponents.

Each autonomic interface takes as input some information formatted in some structured way (e.g. XML), stating a goal the autonomic controller will try to reach. We refer to this abstract goal as a *contract* describing a specific aspect of the component's QoS. Information in contracts can be qualitative or quantitative. We will see examples of them in the following. In the future, the GCM will define the structure for defining autonomic goals, but the content and the nature of this goals will be left unspecified in order to keep the GCM specification generic: implementations of the autonomic controllers should choose the kind of the autonomic goal they can deal with.

More generally, we use the term *manager* to represent a component pluggable into a composite component, that implements autonomic management with respect to a non-functional aspect.

We thus have a spectrum of different implementations of autonomicity, from high-level, contract-based ones to low-level, low complexity ones which directly deal with execution parameters or reconfiguration directives.

As a consequence, we purposely do not specify the type of the argument of the methods for setting adaptivity goals. Indeed, depending on the level of autonomicity, these arguments can be either very precise if the implementation does not provide fully autonomic actions (e.g. resources to be are added/removed); or much higher-level if the system is highly autonomic (e.g. a computation bandwidth, which the components try to provide autonomically). Configuration can also mean recursively adding/decreasing resources of contained components. In the following we illustrate autonomic controllers by examples taken from a highly autonomic context.

Self-Configuring Management Self-configuration, provided by a `SelfConfigurationController`, has the following interface:

```
interface SelfConfigurationController {
  void setConfigurationGoal(any goalDescription);
}
```

The description of a goal can be of any type; it provides a description of a reconfiguration to be ensured.

For instance, the description of a goal could be a piece of code that should be executed by the component in some way; or some semantics that the component should provide or some "standard" behaviour previously stored in the component as a policy. One could also imagine to provide, as a goal, a high-level reconfiguration request (e.g., in a dedicated script language) for describing a reconfiguration to be realized.

Self-Optimisation Management A self-optimising component should provide a `SelfOptimizationController` providing a `setPerformanceGoal` method on its interface:

```
interface SelfOptimizationController {
  void setPerformanceGoal(any PerformanceContract);
}
```

The performance contract contains information that specifies the required performance for the component.

For instance, we can pass a performance contract that requires the component to offer a service time under a specified threshold (quantitative), or that requires the component to offer the best completion time possible (qualitative).

Self-Healing Management A self-healing component exposes a `SelfHealingController`. Such a controller must implement the following interface:

```
interface SelfHealingController {
  void setResiliencyGoal(any resiliencyContract);
}
```

The information passed to the self-healing related interface specifies the requirements on the fault tolerance of the component.

It can be given in a probabilistic fashion. For instance, it can require a failure-free execution probability of 99%, and in the event of failure (1%), it can require a restart time of at most 1 hour (quantitative), or the fastest restart time possible (qualitative).

Self-Protection Management A self-protecting component exposes a `SelfProtectionController`, implementing the interface:

```
interface SelfProtectionController {
  void setProtectionLevel(any protectionContract);
}
```

The protection contract specifies the security level required by the component. For instance a quantitative measure of protection is the estimation of computation time needed to break the encryption code which ensures privacy of exchanged data.

As stated earlier, there are different degrees of compliance w.r.t. autonomicity, and the choice of externally exposing control interfaces instead of fully autonomic ones is independent for each aspect. When a single controller implements several autonomic aspects, it should export them as several different non-functional interfaces.

6.7.3 Hierarchy and Autonomicity

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

For example, hierarchical autonomic optimization can be realized as follows:

- Sub-component resource allocation is directly modified by the container component, if the sub-components have a low level of autonomicity.
- A computation bandwidth is specified by the container, which the contained components try to provide autonomically, if the sub-components are highly autonomic.

Using dynamic controllers presented in the preceding section helps structuring hierarchically the autonomic controllers. Indeed, in a hierarchical context, autonomic controllers of the sub-components should be connected to the autonomic controller of their container, either directly or by the intermediate of manager components.

6.7.4 Autonomic Component Controllers

By allowing autonomic managers, which interface to Fractal controllers, to be replaced at run-time as component controllers, we introduce the notion of *autonomic* *component controllers*. This provides high-level structuring, dynamicity, and autonomicity. Autonomic controllers implemented using components can be dynamically instantiated and are pluggable.

The implementation of controllers by means of Fractal components allows the dynamic addition or removal of controllers. Applied to autonomic component controllers, this means that one could dynamically change the autonomic policy, or add an autonomic controller that was not planned upon the instantiation of a component (however, the interface of this controller must be known by the entity that will use it).

Autonomic component controllers can also manage new components that are dynamically added to a composite, in fact they can be dynamically bound to the added components' controllers. Moreover they can be plugged to, or plug managed controllers to, (external) manager components that belong to the execution platform and are dedicated to management of a given autonomic aspect, when platform-specific knowledge is required.

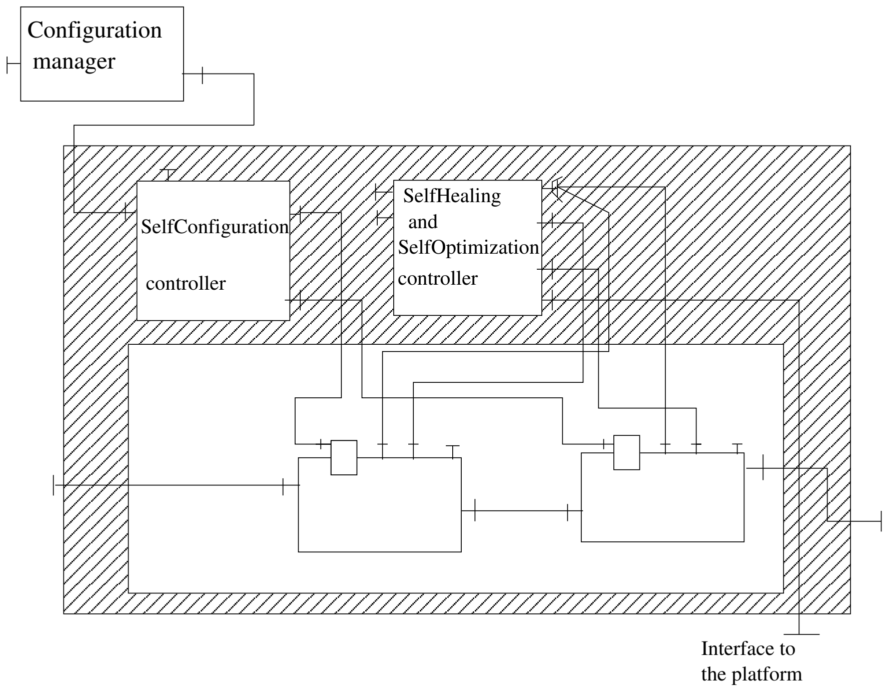
Configuration manager
SelfConfiguration controller
SelfHealing and SelfOptimization controller
Interface to the platform

Figure 11: Autonomic component controllers in the GCM

Three different scenarios can be implemented for the cooperation of managers and controllers:

1. a centralized manager implements the policies and the component controllers have a lower degree of autonomicity;
2. component controllers have full autonomy in implementing the policies;
3. several components share the responsibility of implementing autonomic policies.

We show some examples to clarify those points in the context of autonomic controllers. Figure 11 exemplifies the first scenario (configuration manager and configuration controller), and the second scenario (self-healing and self-optimisation controller). It is quite complete and shows the following choices:

- Autonomic controllers have been implemented in the form of two controller components;
- One of these controller components implement both the self-optimization and the self-healing interfaces;
- The self-configuration controller is less autonomic and receives orders from an external configuration manager (however the degree of autonomicity depends rather on the kind of information the configuration manager would send);
- The self-healing controller uses a multicast interface to send self-healing goals to the two sub-components, whereas the self-configuration and the self-optimization controllers send explicitly (possibly different) requests to the sub-controllers.
- At the sub-level, only the self-configuration controllers are implemented by a component controller, others are simple controllers;
- The composite component does not implement any self-protection controller.
- The self-healing, self-optimizing controller defines a client interface to the platform, this interface can be used either to get information from the platform, e.g. available machines, or to send information, e.g. logging, sending local information about achieved performance, etc. As with every client interface, this interface cannot be used by the platform to send requests to the controllers.

Of course the example of the figure can be generalized by:

- Allowing managers (e.g. the configuration manager) to be implemented as several components, or a composite one.
- Adding manager component(s) handling several autonomic aspects, for example an external manager could handle both the self-configuration and the self-healing controller.
- Repeating the same organization at each level of hierarchy (as suggested roughly in the small sub-components of the Figure 11).
- Adding other non-functional client interfaces.

It is rather improbable that such different choices would be adopted at the same time on the same platform. Indeed one could expect a given component platform to adopt a consistent choice of implementation and degree of autonomicity for all its components, and for all its autonomic controllers. However such highly heterogeneous configurations are possible and very different levels of implementations for autonomicity can interoperate in the GCM.

Figure 12 focuses on the hierarchical implementation of autonomic controllers: it shows a broader view and the dependencies between autonomic controllers of several levels. This figure particularly illustrates the third scenario described above: implementing an autonomic aspect as a hierarchy of managers and controllers. Such a scenario generally relies on the fact that, like on the figure, at each level of composition, the non functional interfaces are the same.

Autonomicity is hierarchically organized as follows:

- The composite component provides a set of non functional interfaces bound to the controllers of the component itself.
- The controllers of the composite are bound to the controllers of the subcomponents by means of subcomponent non-functional interfaces.

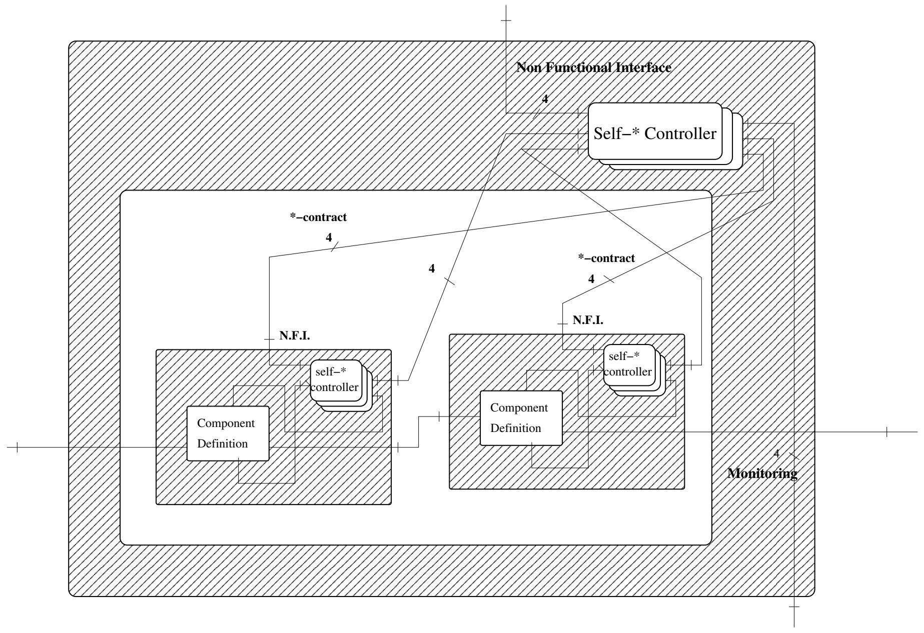

Figure 12: Hierarchical organization of autonomic component controllers

- Each sub-component controller directly monitors the subcomponent itself, and, when requested, provides this information to the composite component controller.
- Each sub-component controller directly manages the subcomponent.

6.7.5 Summary

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 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.

The organization of autonomic controllers presented here enjoys the following properties.

similarity and interoperability – All components (either simple or composite) expose the same set of interfaces, and the managers of the composites require exactly that set of interfaces from its sub-components, this allows the construction of arbitrarily deep hierarchies and the interoperability of different implementations of the GCM. This shows again the importance of specifying an API for the controllers.

Support for locality – Choices can be taken locally whenever possible, thus allowing to implement highly autonomic components.

Support for coordination High-level managers can coordinate sub-components by assigning new contracts, exploiting monitoring information they provide.

Autonomous controller activity We do not precise here if autonomic controllers have their own activity. It seems reasonable to allow these highly independent controllers to run a particular thread, but such an implementation choice depends on the component platform. In any case, if such components have their own activity, the implementation should ensure a clean synchronization with the managed component in order to avoid incoherent states. In other words, some synchronization seems necessary to guarantee that the actions triggered by the autonomic controllers do not affect the coherence of the state of the component. For instance, it seems necessary to synchronize an autonomic controller with the life-cycle controllers upon any action affecting the component structure.

This also applies for any kind of controller, but is crucial for the autonomic ones as, naturally, one would like to give them as much autonomy as possible.

6.8 Packaging

An extension to the Fractal specification for packaging is being defined by the Fractal community. We plan to rely on this specification, and extend it if necessary in order to define packaging for the GCM.

7 Existing Platforms and Implementations

This section shows how some of the features described in Section 6 can be implemented in existing platforms, or for existing applications.

- *Distributed Fractal implementation:* Several implementations of Fractal support distribution, and ProActive [7] is particularly geared at Grid computing. It is important to note that several investigations are being performed by CoreGrid WP3 partners in order to make Fractal compliant several other existing platforms.

- *Virtual Nodes:* this concept has already been implemented in ASSIST (called Virtual Processes) and ProActive, of course, some improvements to those implementations are needed in order to make Virtual Node specification more geared at Grid and components.

- *Multicast and Gathercast interfaces:* Such interfaces are being implemented over the ProActive Component framework.

- *"MxN" communications:* An optimized implementation of MxN communications have already been performed over CCM [13].

- *Autonomic components:* autonomic aspects have been implemented above ASSIST.

- *Component controllers:* Composition of non-functional aspects have been partially studied in AOKell, which should be taken into account in the design of component controllers.

- *Interoperability via the exportation of Web-Service interface:* The integration of ProActive Fractal implementation with HOCs was proposed by WWU and INRIA [15]. Further than the classical integration of the two frameworks, this work also intends to show how to automatically provide a Web-Service interface for a ProActive-Fractal component interface.

All these separate implementations can be seen as proof that GCM features can be implemented inside a Grid environment. Of course, those features have not been put together yet, and have not been implemented over a GCM component framework. The Programming Model Institute expects that those existing implementations will be used when reference implementations of the GCM are designed.

Concerning application and use-cases, we expect to benefit from a lot of use-cases implemented by CoreGrid partners, both in WP3, and in other Virtual Institutes. As an example, a component version of a Grid application is developed over ProActive components during a CoreGrid Research fellow. This application is called JEM3D, it is an object-oriented time domain finite volume solver for the 3D Maxwell Equations.

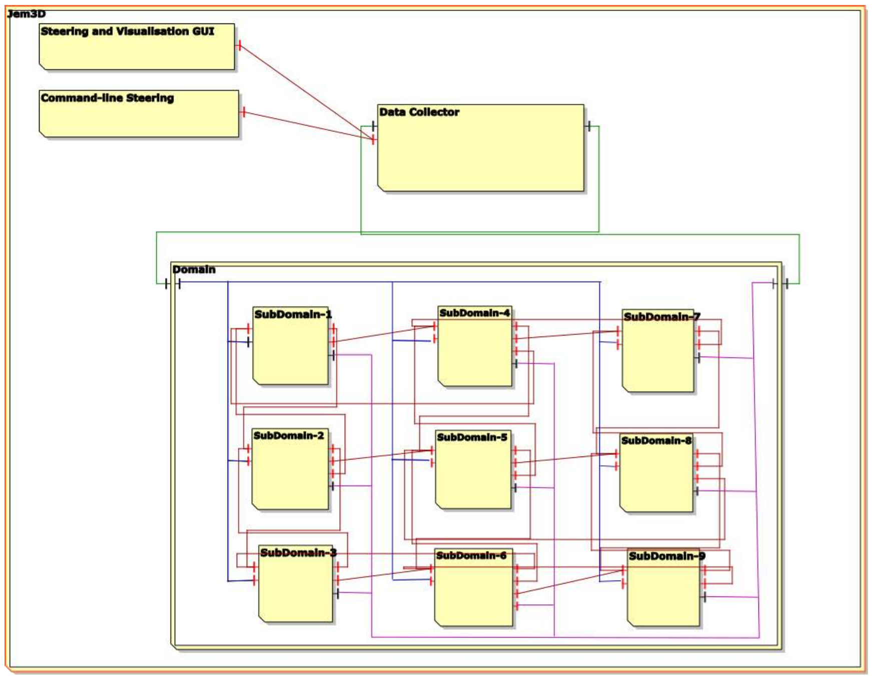

Figure 13: A use-case for the GCM: a Grid component version of JEM3D

Figure 13 shows the component system of this application as it can be visualized by the Fractal GUI: a graphical tool to edit Fractal component configurations. This example also shows how the graphical representation of components used in Fractal can be used to visualize and design component systems.

This application should also be used in the future to illustrate and make experimentations on the Multicast and Gathercast interfaces.

8 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.

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 6.1
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	Globally due to dynamic controllers Dynamic controllers Fractal's content and binding controllers

• 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	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 6.6 play a crucial role in this perspective. Future works on the GCM will provide implementation strategies for the GCM.

References

[1] Dream communication framework. http://dream.objectweb.org.

[2] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for high performance Grid programming in Grid.it. In V. Getov and T. Kielmann, editors, *Proc. of the Workshop on Component Models and Systems for Grid Applications*, CoreGRID series. January 2005.

[3] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. Assist as a research framework for high-performance grid programming environments. In Jose C. Cunha and Omer F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer-Verlag, 2004.

[4] Marco Aldinucci, Françoise André, Jeremy Buisson, Sonia Campa, Massimo Coppola, Marco Danelutto, and Corrado Zoccolo. Parallel program/component adaptivity management. PARCO 2005, Malaga, Spain, to appear, 2005.

[5] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the 1999 Conference on High Performance Distributed Computing*, 1999.

[6] 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.

[7] Francoise Baude, Denis Caromel, and Matthieu Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November*, volume 2888, pages 1226–1242, Springer Verlag, 2003. Lecture Notes in Computer Science, LNCS.

[8] Felipe Bertrand, Randall Bramley, Kostadin B. Damevski, James A. Kohl, David E. Bernholdt, Jay W. Larson, and Alan Sussman. Data redistribution and remote method invocation in parallel component architectures. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium: IPDPS*, 2005.

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

[10] M. Danelutto, M. Vanneschi, C. Zoccolo, N. Tonellotto, R. Baraglia, T. Fagni, D. Laforenza, and A. Paccosi. HPC Application Execution on Grids. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGrid series. Springer, 2006. Dagstuhl Seminar 04451 – November 2004.

[11] 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.

[12] Thierry Delaitre, Tamas Kiss, Ariel Goyeneche, Gabor Terstyanszky, Stephen Winter, and Peter Kacsuk. Gemlca: Running legacy code applications as grid services. *Journal of Grid Computing*, 3(1 – 2):75 – 90, 2005/06.

[13] 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.

[14] 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.

[15] Jan Dünnweber, Sergei Gorlatch, Françoise Baude, Virginie Legrand, and Nikos Parlavantzas. Towards automatic creation of web services for grid component composition. In Vladimir Getov, editor, *Proceedings of the Grids@Work Plugtest, Sophia-Antipolis, France*, October 2005.

[16] Bruneton E., Coupaye T., and Stefani J.B. Recursive and dynamic software composition with sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, 2002.

[17] 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.

[18] 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.

[19] CCA forum. The Common Component Architecture (CCA) Forum home page, 2005. http://www.cca-forum.org/.

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

[21] D. Gannon. Programming the grid: Distributed software components, 2002.

[22] D. Gannon, S. Krishnan, A. Slominski, and G. Kanadaswamy. Building applications from a web service based component architecture. In V. Getov and T. Kielmann, editors, *Component Models and Systems fro Grid Applications*, 1st volume of the CoreGRID series, pages 3–18. Springer, 2995. Invited contribution.

[23] Sébastien Lacour, Christian Pérez, and Thierry Priol. Generic application description model: Toward automatic deployment of applications on computational grids. In *6th IEEE/ACM International Workshop on Grid Computing (Grid2005)*, Seattle, WA, USA, November 2005. Springer-Verlag.

[24] Maciej Malawski, Dawid Kurzyniec, and Vaidy Sunderam. MOCCA – towards a distributed CCA framework for metacomputing. In *Proceedings of the 10th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS2005)*, 2005.

[25] 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.

[26] Vladimir Mencl and Tomas Bures. Microcomponent-based component controllers: A foundation for component aspects. In *APSEC*. IEEE Computer Society, Dec. 2005.

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

[28] Sun Microsystems. Enterprise Java Beans. http://java.sun.com/products/ejb, 2005.

[29] omg.org team. CORBA Component Model, V3.0. http://www.omg.org/technology/documents/formal/components.htm, 2005.

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

[31] 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.

[32] The Component Technologies Project. The Babel home page, 2005. http://www.llnl.gov/CASC/components/babel.html.

[33] 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.

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

[35] Clemens Szyperski. *Component software: beyond object-oriented programming.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[36] The CoreGRID Programming Model Virtual Institute. Roadmap version 1 on programming model. Technical Report D.PM.01, CoreGRID, Programming Model Virtual Institute, Feb 2005.

[37] J. Thiyagalingam, S. Isaiadis, and V. Getov. Towards building a generic grid services platform: a component-oriented approach. In V. Getov and T. Kielmann, editors, *Component Models and Systems for Grid Applications, CMSGA workshop at ICS 2004*. Springer Verlag, 2004.

[38] 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 Fractal API

A.1 Java API

```
  package org.objectweb.naming;
public interface Name {
  NamingContext getNamingContext ();
  byte[] encode () throws NamingException;
}
public interface NamingContext {
  Name export (Object o, Object hints) throws NamingException;
  Name decode (byte[] b) throws NamingException;
}
public interface Binder extends NamingContext {
  Object bind (Name n, Object hints) throws NamingException;
}
public class NamingException extends Exception {
  public NamingException (String msg) { super(msg); }
}
package org.objectweb.fractal.api;
import org.objectweb.fractal.api.factory.InstantiationException;
public interface Component {
  Type getFcType ();
  Object[] getFcInterfaces ();
  Object getFcInterface (String interfaceName) throws NoSuchInterfaceException;
}
public interface Interface {
  Component getFcItfOwner ();
  String 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 Exception {
  public NoSuchInterfaceException (String itfName) { super(itfName); }
}
package org.objectweb.fractal.api.control;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.api.NoSuchInterfaceException;
public interface AttributeController { }
public interface BindingController {
  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 {
  Object[] getFcInternalInterfaces ();
  Object getFcInternalInterface (String interfaceName) throws NoSuchInterfaceException;
  Component[] getFcSubComponents ();
  void addFcSubComponent (Component subComponent)
    throws IllegalContentException, IllegalLifeCycleException;
  void removeFcSubComponent (Component subComponent)
    throws IllegalContentException, IllegalLifeCycleException;
}
public interface SuperController {
  Component[] getFcSuperComponents ();
}
public interface LifeCycleController {
  String getFcState ();
  void startFc () throws IllegalLifeCycleException;
  void stopFc () throws IllegalLifeCycleException;
}
public interface NameController {
  String getFcName ();
  void setFcName (String name);
}
public class IllegalBindingException extends Exception {
  public IllegalBindingException (String msg) { super(msg); }
}
public class IllegalContentException extends Exception {
  public IllegalContentException (String msg) { super(msg); }
}
public class IllegalLifeCycleException extends Exception {
  public IllegalLifeCycleException (String msg) { super(msg); }
}
package org.objectweb.fractal.api.factory;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.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 Exception {
  public InstantiationException (String msg) { super(msg); }
}
package org.objectweb.fractal.api.type;
import org.objectweb.fractal.api.NoSuchInterfaceException;
public interface ComponentType extends org.objectweb.fractal.api.Type {
  InterfaceType[] getFcInterfaceTypes ();
  InterfaceType getFcInterfaceType (String name) throws NoSuchInterfaceException;
}
public interface InterfaceType extends org.objectweb.fractal.api.Type {
  String getFcItfName ();
  String getFcItfSignature ();
  boolean isFcClientItf ();
  boolean isFcOptionalItf ();
  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;
}
```

A.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;
```

A.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);
  };
};
```

B Fractal ADL

B.1 standard.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" ?>
<?add ast="templateController" itf="org.objectweb.fractal.adl.implementations.TemplateController" ?>
<?add ast="definition"     itf="org.objectweb.fractal.adl.implementations.TemplateControllerContainer" ?>
<?add ast="component"      itf="org.objectweb.fractal.adl.implementations.TemplateControllerContainer" ?>

<!-- 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="org.objectweb.fractal.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" ?>

<!-- ********************************************************************** -->
<!--                  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="template-controller"      ast="templateController" ?>
<?map xml="template-controller.desc" ast="templateController.descriptor" ?>

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

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

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

<!ELEMENT component (comment*,interface*,component*,binding*,content?,attributes?,
                 controller?,template-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) #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 template-controller (comment*) >
<!ATTLIST template-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
>

<!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
>
```