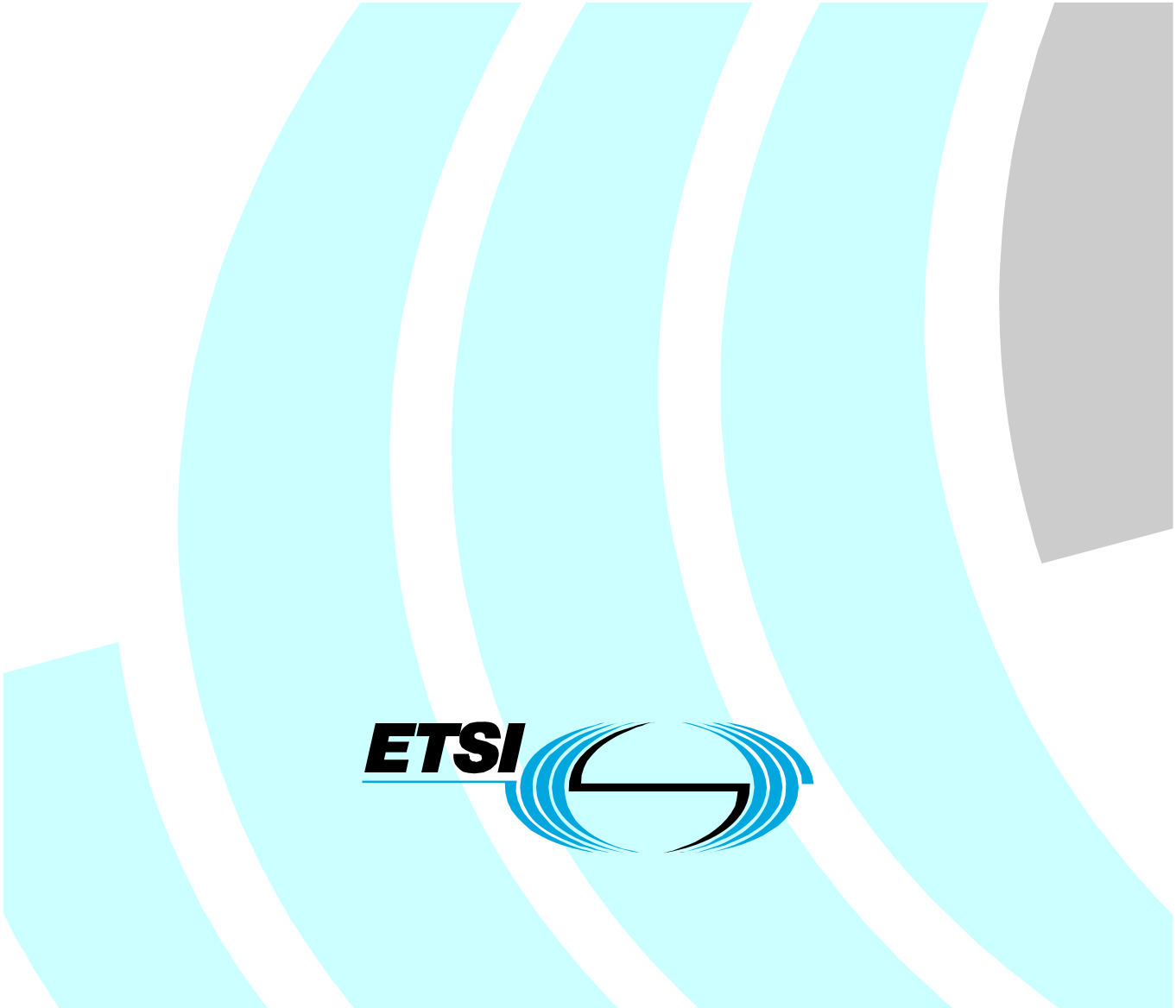
ETSI TS 102 830 V1.1.1 (2010-03)

Technical Specification

GRID;
Grid Component Model (GCM);
GCM Fractal Management API



Reference
DTS/GRID-0004-4 GCM_MgmtAPI

Keywords
architecture, interoperability, network, service

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Contents

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (http://webapp.etsi.org/IPR/home.asp).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Foreword

This Technical Specification (TS) has been produced by ETSI Technical Committee GRID (GRID).

The present document is related to documents 102-650-1 (GCM Interoperability Deployment), 102-650-2 (GCM Interoperability Application Description), and 102-830 (GCM Management API).

Introduction

The GCM has been first defined in the NoE CoreGRID (42 institutions). A reference Open Source implementation has been tested in the 5 previous GRID Plugtests organized from 2004 to 2008 by INRIA and ETSI. The GridCOMP EU project (FP6, started June 2006 to February 2009) is working to further assess and experiment with the specification.

1 Scope

The present document describes standard API to manage GCM components at runtime in an interoperable way. It defines component management in a standard manner to create, introspect, monitor and reconfigure components at execution.

The present document will help enterprises and laboratories to use their large-scale computer and telecom infrastructures with the necessary virtualization.

Its primary audience are developers of distributed software who need to specify complex applications by composing existing software components.

2 References

References are either specific (identified by date of publication and/or edition number or version number) or non-specific.

- For a specific reference, subsequent revisions do not apply.
- Non-specific reference may be made only to a complete document or a part thereof and only in the following cases:
  - if it is accepted that it will be possible to use all future changes of the referenced document for the purposes of the referring document;
  - for informative references.

Referenced documents which are not found to be publicly available in the expected location might be found at http://docbox.etsi.org/Reference.

NOTE: While any hyperlinks included in this clause were valid at the time of publication ETSI cannot guarantee their long term validity.

2.1 Normative references

The following referenced documents are indispensable for the application of the present document. For dated references, only the edition cited applies. For non-specific references, the latest edition of the referenced document (including any amendments) applies.

[1] ETSI TS 102 829: "GRID; Grid Component Model (GCM); GCM Fractal Architecture Description Language (ADL)".

[2] ETSI TS 102 828: "GRID; Grid Component Model (GCM); GCM Application Description".

[3] ETSI TS 102 827: "GRID; Grid Component Model (GCM); GCM Interoperability Deployment"..

2.2 Informative references

The following referenced documents are not essential to the use of the present document but they assist the user with regard to a particular subject area. For non-specific references, the latest version of the referenced document (including any amendments) applies.

[i.1] "The Fractal Component Model".

NOTE: Available at http://fractal.objectweb.org/specification/index.html.

[i.2] CoreGRID NoE (FP6): "Basic Features of the Grid Component Model".

NOTE: Available at http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf.

[i.3] GCM: "A Grid extension to Fractal for Autonomous Distributed Components", F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, C. Perez. Annals of Telecommunications - The Fractal Initiative, 2008.

[i.4] "A formal specification of the Fractal component model".

NOTE: Available at http://hal.inria.fr/inria-00338987/.

3 Definitions and abbreviations

3.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

binding: communication path between a client and a server interface

cardinality: property of an interface type that indicates how many interfaces of this type a given component may have

NOTE: The cardinality is singleton, collection, multicast or gathercast.

client interface: component interface that emits operation invocations

complementary interface: of an interface I, a component interface with the same name, signature, contingency, and cardinality as I, but with opposite role and visibility (external or internal)

component: runtime entity exhibiting a recursive structure and reflexive capabilities

NOTE: A component is composed of a set of controllers and a content. A component has well defined access points called interfaces, and provides introspection and control capabilities (intercession) to other components.

component interface: access point to a component, i.e. a place where operation invocations can be emitted or received

composite component: component that exposes its content (inner structure)

content: one of the two parts of a component, the other one being its controller

NOTE: A content is an abstract entity controlled by a controller. The content of a component is (recursively) made of sub components and bindings.

contingency: property of an interface, indicates if the functionality of this interface is guaranteed to be available or not, while its component is running

NOTE: The contingency is either optional or mandatory.

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

NOTE: By convention, control interfaces are server interfaces whose name ends with -controller, or is equal to component.

controller: one of the two parts of a component, the other one being its content

NOTE: A controller is an abstract entity that embodies the control behaviour 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).

external interface: component interface that is only accessible from outside the component

factory: component that can create other components

NOTE: Generic factories can create several kinds of components, while standard component factories create only one kind of components.

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

intercession: ability of a component (seen as a program) to modify its own execution state; or to alter its own interpretation or semantics

internal interface: component interface that is only accessible from inside the component, i.e. from its sub components

introspection: ability of a component (seen as a program) to observe and reason about its own execution state

language interface: type made of several operation declarations, i.e. a Java interface for the Java API of the GCM

mandatory interface: component interface whose functionality is guaranteed to be available, while the component is running

optional interface: component interface whose functionality is not guaranteed to be available, while the component is running

primitive component: component with some control interfaces, but that does not expose its content

reflection (reflective capabilities): ability of a component (seen as a program) to manipulate as data the entities that represent its execution state during its own execution

NOTE: This manipulation can take two forms: introspection and intercession.

role: property of a component interface, indicates if this interface is a client or server interface

server interface: component interface that receives operation invocations

shared component: component that is contained in several super components

signature: of a component interface, is the name of the language interface type corresponding to this component interface

sub component: component that is contained in another component

super component: relatively to a (sub) component, a component that contains this (sub) component

NOTE: Due to component sharing, a component may have several super components.

template: special kind of factory that creates components that are "isomorphic" to itself

type: set of structural properties common to a set of entities (components and interfaces for instance)

3.2 Abbreviations

For the purposes of the present document, the following abbreviations apply:

ADL	Architecture Description Language
API	Application Programming Interface
GCM	Grid Component Model

4 Overview of the Grid Component Model (GCM)

By enforcing a strict separation between interface and implementation and by making software architecture explicit, component-based programming can facilitate the implementation and maintenance of complex distributed software systems.

Existing component-based frameworks and architecture description languages, however, provide only limited support for extension, adaptation, and distribution. These limitations lead to major drawbacks: they prevent the easy and possibly dynamic introduction of different control facilities for components such as non-functional aspects; they prevent application designers and programmers from making important trade-offs such as degree of configurability vs performance and space consumption; and they can make difficult the use of these frameworks and languages in different environments, including distributed systems.

The Grid Component Model (GCM) extends the Fractal component model. As an extension of Fractal the GCM reuses the Fractal concepts and features: we do not distinguish in the present document what is part of the basic Fractal component model and what is an extension (excepted for their namespaces, see annex B). We specify the whole features making the GCM. The GCM model alleviates the above problems by introducing a notion of component endowed with an open set of control capabilities. In other terms, components in GCM are reflective, and their reflective capabilities are not fixed in the model but can be extended and adapted to fit the programmer's constraints and objectives, Also the GCM handle the component distribution over a set of resources by introducing location information and parallel communication mechanisms.

Main goals of the GCM component model are to implement, deploy and manage (i.e. monitor and dynamically reconfigure) complex and distributed software systems. These goals motivate the main features of the GCM model: parallel communications (predefined and customizable communication patterns), remote communication (fully transparent remote binding: a remote component can be used as a local one), composite components (to have a uniform view of applications at various abstraction levels), introspection capabilities (to monitor a running system), and configuration and reconfiguration capabilities (to deploy and dynamically reconfigure an application). But another goal of the GCM is to be applicable to many software.

The present document is the 4th part of the GCM specification; the 3 previous parts of the standard have been published by ETSI [1], [2] and [3]. More explanations will be found in the following white papers: [i.1], [i.2], [i.3] and [i.4].

The GCM API provides instantiation, reflection, intercession, introspection, configuration capabilities at run time. Not all the controllers exposing this API are mandatory. As a result of this modular and extensible organization (anyone is free to define its own control interfaces, in order to provide new introspection and intercession capabilities) GCM components can be used in very different situations and cover a large set of application needs.

The present document describes the Java API of the GCM. Each clauses contains a part of the normative API, and the whole API is included in the Annex A.

5 Basic GCM component introspection

A GCM component provides introspection functions to introspect its external features, i.e. its boundary. This clause defines more precisely the external features of GCM components, and specifies the interfaces related to the introspection of these features. The interfaces related to the introspection (and reconfiguration) of the internal features of GCM components are specified in the next clause.

5.1 External component structure

Depending on the level of observation, or scale, a GCM component can be seen as a black box or as a white box. When seen as a black box, i.e. when its internal organization is not visible, the only visible elements of a GCM component are some access points to this black box, called its external interfaces (see figure 1). Each interface has a name, in order to distinguish it from the other interfaces of the component. All the external interfaces of a component must have distinct names, but two interfaces in two distinct components may have the same name. One may distinguish two kinds of interfaces: a client (or required) interface emits operation invocations, while a server (or provided) interface receives them.

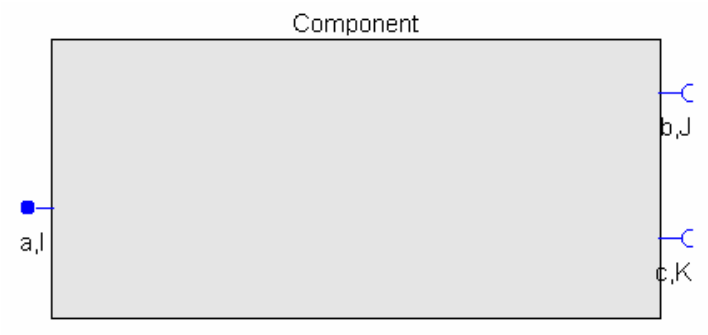


Figure 1: External view of a GCM component

The interfaces of a component can be introspected with two language interfaces, specified in clauses 5.2 and 5.3: one to get the list of interfaces of a component, and one to introspect the interfaces themselves.

5.2 Component introspection API

In order to discover the external interfaces of a component, a component provides an interface that implements the Component interface (see figure 2). This language interface provides two operations named getFcInterfaces and getFcInterface, that can be used to retrieve the interfaces of the component. The first operation takes no arguments, and returns an array containing all the external interfaces, either client or server (which include non functional interfaces), of the component, including the Component interface. The second operation takes the name of an interface as parameter, and returns this interface, if it exists.

```
package org.objectweb.fractal.api;
interface Component {
   Object[] getFcInterfaces ();
   Object getFcInterface (String itfName) throws NoSuchInterfaceException;
   Type getFcType ();
}
interface Type {
   boolean isFcSubTypeOf (Type t);
}
```

Figure 2: Component introspection API

The getFcInterfaces and getFcInterface operations return references that give access to requested interfaces. In other words, the references returned by these operations can be used directly, after an appropriate cast, to invoke operations on the component's server interfaces (no explicit binding is needed). For example, if a component has a server interface named account implementing the language interface Account, then the getBalance operation of this interface can be invoked with a code like ((Account)c.getFcInterface("account")).getBalance(), where c is a reference to the Component interface of the component.

The Component interface also provides a getFcType operation, which returns the type of the component, as a Type reference. This interface defines a minimal notion of type, which actually defines only one operation named isFcSubTypeOf, whose role is to test if a given type is a sub type or not of another type. This interface can be extended to define more useful type systems for components and component interfaces, such as the one defined in clause 8.

The org.objectweb.fractal.api.NoSuchInterfaceException exception must be thrown in the getFcInterface operation when a requested component interface is not found.

A component interface implementing Component must be named component.

5.3 Interface introspection API

By default the references returned by the getFcInterface and getFcInterfaces operations provide access to the requested interfaces, and nothing more. In particular, it is impossible to find the names of these interfaces. In order to provide such interface introspection functions, a component ensures that the references returned by the above operations are castable into the Interface type (see figure 3). This interface specifies four operations to get the name of a component interface, to get its type (as a Type reference), to get the Component interface of the component to which it belongs, and to test if the interface is internal or not (see also clause 6.1).

```
package org.objectweb.fractal.api;
interface Interface {
   String getFcItfName ();
   Type getFcItfType ();
   Component getFcItfOwner ();
   boolean isFcInternalItf ();
}
```

Figure 3: Interface introspection API

Note that the getFcItfOwner operation allows one to discover all the interfaces of a component from any interface of this component, and not only from its interface of type Component. For example, if a is a reference to the Account interface of such component, the Component interface of this component can be retrieved with a code like ((Interface)a).getFcItfOwner(). The result can then be used to get the reference of any other interface of the component.

NOTE 1: Component and Interface have very distinct roles and should not be mixed up. On the one hand, Component is a language interface that is provided by a component just like any other language interface. On the other hand, Interface is a language interface that is implemented by all the component interfaces: any component interface of such a component implements both a specific language interface, such as Account or Component, and Interface.

NOTE 2: A functional interface such as Account is likely to provide a getName or getOwner operation, as the Interface interface. And since a reference of one type should be castable to the other, there is a risk of name conflicts (at least in some languages, such as Java). In order to reduce these risks, the Interface operations follow the pattern verbFcnoun or verbGCMnoun. This pattern has then been generalized to all the Fractal/GCM APIs.

6 GCM component configuration

At the next level of control capability, beyond the "introspection" level where components provide interfaces to introspect their external features, a GCM component can provide control interfaces to introspect and reconfigure its internal features. This clause defines these internal features, and specifies some predefined interfaces to introspect and reconfigure them.

6.1 Internal component structure

A GCM component is formed out of two parts (see figure 4): a membrane and a content: The membrane (the grey part of the Composite component on the figure 4) is made of a set of controllers which expose non-functional interfaces (the green interface on the figure 4). The content of a component is either composed of an object or of (a finite number of) other components, called sub components, which are under the control of the controller of the enclosing component. The GCM model is thus recursive and allows components to be nested (i.e. to appear in the content of enclosing components) at an arbitrary level. A component that exposes its content is called a composite component. A component that does not expose its content, but has at least one control interface (see below), is called a primitive component.

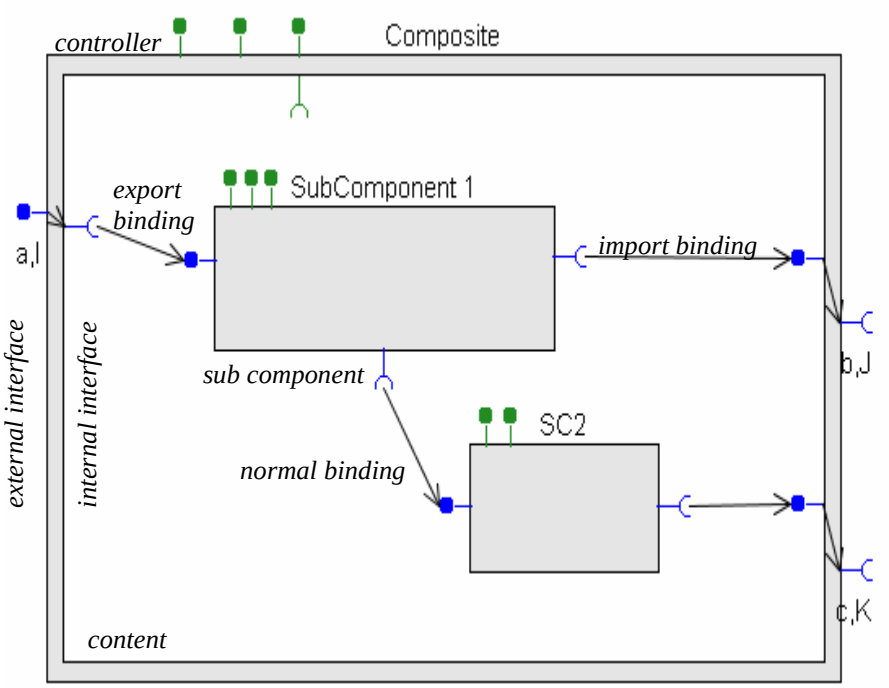


Figure 4: Internal view of a GCM component

The controller of a component can have external and internal interfaces. External interfaces are accessible from outside the component, while internal interfaces shall be accessible only from the component's sub components. All the external interfaces of a component must have distinct names, all its internal interfaces must have distinct names, but a component can have an external and an internal interface of the same name. A functional interface is an interface that corresponds to a provided or required functionality of a component, while a control interface is a server interface that corresponds to a "non functional aspect", such as introspection, configuration or reconfiguration, and so on. By convention, an interface is considered to be a control interface if its name is equal to `component`, or ends with `-controller`. All other interfaces are considered to be functional interfaces.

The controller of a component embodies the control behaviour associated with a particular component. In particular, a component controller can:

- Provide an explicit and causally connected representation of the component's sub components.
- Intercept incoming and outgoing operation invocations targeting or originating from the component's sub components.
- Superpose a control behaviour to the behaviour of the component's sub components, including suspending, check pointing and resuming activities of these sub components.

Each component controller can thus be seen as implementing a particular semantic of composition for the component's sub components. The control capability of a controller is not limited by the model. For instance, it can be mainly interception-based as in industrial component frameworks containers for instance; or it can be void (i.e. no control is exercised - in this case, the controller can still be useful for it can provide a representation of its content and manifest a containment relationship).

A binding is a communication path between one client interface and one server interface, which means that the operation invocations emitted by the client interface should be accepted by the specified server interface. A binding between a client interface c and a server interface s of two components C and S must verify one of the following constraints (see figure 4):

- c and s are external interfaces, and C and S have a direct common enclosing component. Such bindings are called normal bindings.
- c is an internal interface, s is an external interface, and S is a sub component of C. Such bindings are called export bindings.

- c is an external interface, s is an internal interface, and C is a sub component of S. Such bindings are called import bindings.

In addition to these structural constraints, which ensure that bindings cannot "cross" component boundaries except through interfaces, a binding can be established between a client and a server interface only if the server interface can accept at least all the operation invocations that the client interface can emit. In other words, the (language) type of the server interface must be a sub type of the type of the client interface (the two interfaces can of course be of the same type since a sub typing relation must be reflexive). The last constraint is that a client interface can be bound to at most one server interface, while several client interfaces can be bound to the same server interface. This constraint is relaxed in the case of collective interface, in particular with client interface having a multicast cardinality.

Bindings are not concrete GCM elements, and do not appears as such in the APIs. They are abstractions used in the definition of the next clauses.

Clause 6.2 introduce the predefined controllers APIs, namely: Attribute controller, Binding controller, Lifecycle controller, Content controller, Collective controller, Multicast controller, Gathercast controller, Migration controller, Monitoring controller, Priority controller.

6.2 Attribute control API

An attribute is a configurable property of a component, such as the text or color of a button, or the maximum size of a pool or cache component. Attributes are generally of primitive type, and are used to configure the state of components without needing to use bindings (it is possible to configure the text of a button, for example, by binding this button component to a text component; but this is overly complex for what is needed). A component can provide an AttributeController interface to read and write its attributes from outside the component (see figure 5).

```
package org.objectweb.fractal.api.control;
public interface AttributeController { }
```

Figure 5: Attribute control API

In this case, the component must actually provide a sub interface of this interface, since the AttributeController interface is in fact empty. This sub interface must contain one getter and/or setter operation per configurable attribute. For example:

- a component that wants to provide an AttributeController interface for a read only string attribute foo must provide a sub interface of this interface containing the following operation: String getFoo();
- a component that wants to provide an AttributeController interface for a write only string attribute foo must provide a sub interface of this interface containing the following operation: void setFoo(String foo);
- a component that wants to provide an AttributeController interface to configure two String attributes foo and bar must provide a sub interface of this interface containing the following operations: String getFoo(), void setFoo(String foo), String getBar() and void setBar(String bar).

It is a requirement of the present document that setters and getters must follow the lexicographic and typing conventions introduced informally in the example above with respect to names and signatures of setters and getters (these conventions are those of the Java Beans component model).

A component interface implementing AttributeController must be named `attribute-controller`.

6.3 Binding control API

A component can provide the BindingController interface to bind and unbind its client interfaces to other components through bindings (see figure 6).

```
package org.objectweb.fractal.api.control;
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;
}
```

Figure 6: GCM Binding control API

This interface defines the following operations:

- The listFc operation returns the names of the client interfaces including the internal ones of the component. These names are the names that can be passed as first argument to the lookupFc operation.
- The lookupFc operation takes as parameter the name of a client interface of the component, either external or internal, and returns the server interface that is bound to this client interface, or null if there is no such interface. As the component to which the server interface belongs supports the interface introspection (see clause 5.3), the reference returned by this operation can be cast to org.objectweb.fractal.api.Interface. The lookupFc behaviour is undefined if its argument is a multicast interface (see clause 6.7).
- The bindFc operation takes as parameters the name of a client interface of the component, either external or internal, and a server interface of another component, and binds these two interfaces together. As above, the server interface can be cast to org.objectweb.fractal.api.Interface, thanks to the introspection capabilities provided by the server component. An error shall be thrown if the interface is already bound, unless the client interface name refers to a multicast interface.
- The unbindFc operation takes as parameter the name of a client interface of the component, either external or internal, and unbinds this interface. In case of multicast interface, all bindings to this multicast interface are removed.

These operations may throw a org.objectweb.fractal.api.NoSuchInterfaceException exception if a specified client interface does not exist, an org.objectweb.fractal.api.control.IllegalLifeCycleException exception when a component is not in an appropriate state to perform an operation, and an org.objectweb.fractal.api.control.IllegalBindingException exception in case of other errors related to bindings.

A component interface implementing BindingController must be named `binding-controller`.

6.4 Content control API

A component can provide the ContentController interface to add and remove sub components in this component (see figure 7).

```
package org.objectweb.fractal.api.control;
public interface ContentController {
   Object[] getFcInternalInterfaces ();
   Object getFcInternalInterface (String itfName)
      throws NoSuchInterfaceException;
   Component[] getFcSubComponents ();
   void addFcSubComponent (Component c)
      throws IllegalContentException, IllegalLifeCycleException;
   void removeFcSubComponent (Component c)
      throws IllegalContentException, IllegalLifeCycleException;
}
public interface SuperController {
   Component[] getFcSuperComponents ();
}
```

Figure 7: GCM Content control API

This interface defines three operations to get the list of sub components of a component, and to add and remove sub components in a component:

- The getFcSubComponents operation returns the list of sub components of the component, as an array of Component references. The getFcSuperComponents operation, in the SuperController interface (see figure 7), provides the opposite function: it returns the components that contain this component, and which are called its super components.
- The addFcSubComponent operation takes a component as parameter, as a Component reference, and adds this component to the component's content.
- The removeFcSubComponent operation takes a component as parameter, as a Component reference, and removes this component from the component's content.

A given component can be added to several other components. Such a component is said to be shared between these components. Shared components are useful, paradoxically, to preserve component encapsulation. Consider, for example, a menu and a toolbar components, with an "undo" toolbar button corresponding to an "undo" menu item. It is natural to represent the menu items and toolbar buttons as sub components, encapsulated in the menu and toolbar components, respectively. But, without sharing, this solution does not work for the "undo" button and menu item, which must have the same state (enabled or disabled): these components, or an associated state component, must be put outside the menu and toolbar components. With component sharing, the state component can be shared between the menu and toolbar components, in order to preserve component encapsulation. Shared components are also useful to help separate "aspects" in component based applications.

Because of shared components, the structure of a GCM component, in terms of direct and indirect sub components, is not necessarily a tree, but can be a directed acyclic graph (it cannot be an arbitrary graph, because a component cannot be added inside itself or inside one of its direct or indirect sub components). In terms of bindings, this structure can be arbitrary, provided it follows the constraints of clause 6.1. In particular, bindings can form cycles.

The ContentController interface also specifies two operations to get the internal interfaces of the component, which are similar to the getFcInterface and getFcInterfaces operations. These operations are useful to bind the internal interfaces to sub components.

The content controller operations may throw a org.objectweb.fractal.api.NoSuchInterfaceException exception if a specified client interface does not exist, an org.objectweb.fractal.api.control.IllegalLifeCycleException exception when a component is not in an appropriate state to perform an operation and, in case of other errors related to content control, an org.objectweb.fractal.api.control.Illegal ContentException exception.

A component interface implementing ContentController (resp. SuperController) must be named `content-controller` (resp. `super-controller`).

In order to associate local names to the sub components of a component, similar to the local names of the interfaces of a component, a possibility is to ensure that all these sub components provide the NameController interface defined in figure 8.

```
package org.objectweb.fractal.api.control;
public interface NameController {
   String getFcName ();
   void setFcName (String name);
}
```

Figure 8: Name control API

A component interface implementing NameController must be named `name-controller`.

6.5 Life cycle control API

Changing an attribute or a binding, or removing a sub component, with the above control interfaces, and while components are executing, can be dangerous: messages can be lost, the application's state may become inconsistent, or the application may simply crash. In order to provide a minimal support to help implement such dynamic reconfigurations, a component can provide the LifeCycleController interface (see figure 9).

```
package org.objectweb.fractal.api.control;
public interface LifeCycleController {
   String STARTED = "STARTED";
   String STOPPED = "STOPPED"
   String getFcState ();
   void startFc ()
      throws IllegalLifeCycleException;
   void stopFc ()
      throws IllegalLifeCycleException;
}
package org.etsi.uri.gcm.api.control;
public interface GCMLifeCycleController extends LifeCycleController {
   void terminateGCMComponent ()
      throws IllegalLifeCycleException;
}
```

Figure 9: GCM Life cycle control API

The LifeCycleInterface interface provides two operations named startFc and stopFc, to start and stop a component properly. As for the addFcSubComponent and removeFcSubComponent operations, the semantics of these operations is voluntarily as weak as possible, so that many implementations are possible: these operations may or may not be recursive, i.e. starting or stopping a component may or may not automatically start or stop all its direct and indirect sub components. Likewise, the effect of these operations on the component's state is voluntarily not specified (in fact it cannot be specified here, because the APIs defined in the present document do not provide access to this state). In particular, the stopFc operation can be seen as a clean up operation invoked before the component is destroyed, or as a suspend operation. In the first case the component's state will be erased, while in the second case it will be left unchanged.

Another operation, terminateGCMComponent, is available in the GCMLifeCycleController interface and allows users to terminate a GCM component.

In addition to these operations, the LifeCycleController interface also provides a getFcState operation. This operation returns the current state of the component (in a strict sense, i.e. without taking into account its sub components, which can have a different execution state), as a String. The STARTED and STOPPED String objects mean that the component is started or stopped, respectively.

In the STARTED state, i.e. just after successful completion of a call to startFc, a component can emit or accept operation invocations, which are guaranteed to execute "normally". Note that this does not prevent the unbindFc and removeFcSubComponent operations to throw the IllegalLifeCycleException if they are invoked while the component is in this state (in order to prevent a component from being reconfigured while it is in an unstable state).

In the STOPPED state, i.e. right after successful completion of a call to stopFc, a component cannot emit operation invocations, and can accept operation invocations only through control interfaces. Operation invocations to the functional interfaces of a stopped component are accepted and will be executed when the component is restarted.

However, some components may require very different life cycles. Of course, completely arbitrary life cycles can be specified by providing completely new interfaces, distinct from the LifeCycleController interface. More commonly, life cycles can be adapted from the basic one by extending the LifeCycleController interface to introduce new states and transitions or even to change the transitions of the basic life cycle. In this case, it is a requirement of the present document that the semantics associated to the STARTED and STOPPED states should be preserved.

The org.objectweb.fractal.api.control.IllegalLifeCycleException exception may be thrown when a requested transition, in a life cycle automaton, is not valid.

A component interface implementing LifeCycleController or GCMLifeCycleController must be named `lifecycle-controller`.

6.6 Collective interface control API

In order to provide facilities for parallel programming, GCM defines collective interfaces by adding new cardinalities in the specification of interfaces, namely multicast (see clause 6.7) and gathercast (see clause 6.8). 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 and gathering purposes. Collective 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 they expose the collective nature of a given interface. The method signatures of client and server interfaces are different when using collective interfaces. Thus, in order to give a way to ensure the compatibility between such interfaces, the component controller of collective interfaces can extend the CollectiveInterfaceController interface (see figure 10).

```
package org.etsi.uri.gcm.api.control;
public interface CollectiveInterfaceController {
   public void ensureGCMCompatibility (InterfaceType itfType, Interface itf)
      throws IllegalBindingException;
}
```

Figure 10: Collective interface control API

The CollectiveInterfaceController interface provides a single operation named ensureGCMCompatibility. As its name implies, this operation ensures the type compatibility between a collective interface and a component interface which has to be bound to the collective interface. This operation takes as parameters the org.objectweb.fractal.api.type.InterfaceType of the collective interface and the org.objectweb.fractal.api.Interface of the component interface. This operation does not return anything but throws an org.objectweb.fractal.api.control.IllegalBindingException exception when trying to bind two incompatible interfaces.

6.7 Multicast control API

Multicast interfaces provide abstractions for 1-to-n communications, i.e. transform a single invocation into a list of invocations.

A multicast server interface transforms each single invocation into a set of invocations that are forwarded either to implementation code of a primitive component, or to bound server interfaces of internal components. A multicast client interface transforms each single invocation coming either from implementation code of a primitive component or from an internal component into a set of invocations to bound server interfaces of external components.

When a single invocation is transformed into a set of invocations, these invocations are forwarded to a set of connected server interfaces. The semantics of the propagation and the distribution invocation parameters are customizable, and the result of an invocation on a multicast interface - if there is a result - is a list of results by default but can also be customizable. 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, with automatic distribution of invocation parameters.

In order to manage its multicast interfaces, a component can provide the MulticastController interface (see figure 11).

```
package org.etsi.uri.gcm.api.control;
public interface MulticastController extends CollectiveInterfaceController {
   public void unbindGCMMulticast (String multicastItfName, Object serverItf)
      throws NoSuchInterfaceException, IllegalBindingException, IllegalLifeCycleException;
   public boolean isBoundTo (String multicastItfName, Object[] serverItfs)
      throws NoSuchInterfaceException;
   Object[] lookupGCMMulticast (String clientItfName)
      throws NoSuchInterfaceException;
}
```

Figure 11: Multicast control API

In addition to extending the CollectiveInterfaceController interface, the MulticastController interface defines one operation to manage the multicast interface: the unbindGCMMulticast operation. This operation removes a binding between a multicast interface and a server interface. Parameters of this operation are the name of the multicast interface and the reference on the server interface which can be cast to org.objectweb.fractal.api.Interface as previously. This operation may throw an org.objectweb.fractal.api.NoSuchInterfaceException exception if the specified multicast interface does not exist, an org.objectweb.fractal.api.control.IllegalLifeCycleException exception when the component is not in an appropriate state to perform the operation, and an org.objectweb.fractal.api.control.IllegalBindingException exception in case of other errors related to bindings.

Two other operations are also available:

- The isBoundTo operation checks if a multicast interface is bound to one of the given server interfaces. This operation takes as parameters the name of the multicast interface and an array of references on server interfaces which can be cast to org.objectweb.fractal.api.Interface, and returns a boolean indicating if one of these server interfaces is bound on the multicast interface. This operation may be useful to know if the given multicast interface of a component is bound on a component.

- The lookupGCMMulticast operation takes as parameter the name of a multicast interface of the component and returns the server interfaces that are bound to this multicast interface, or null if there is no such interface.

These operations may throw an org.objectweb.fractal.api.NoSuchInterfaceException exception if the specified multicast interface does not exist.

A component interface implementing MulticastController must be named `multicast-controller`.

6.8 Gathercast control API

A gathercast interface is an abstraction for n-to-1 communications, i.e. transforms a list 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. Returned values are redistributed to the invoking components. Gathering of incoming data and redistribution are fully customizable. Both client and server interfaces may have a gathercast cardinality. A gathercast client interface transforms a set of invocations coming from client interfaces of inner components or from the implementation code of the component, into a single invocation. A gathercast server interface transforms a set of invocations coming from client interfaces of external components into a single invocation to one server interface of an inner component, or to the implementation code in case of a primitive component.

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, bindings to gathercast interfaces are bidirectional links. In other words, a gathercast interface is aware of which interfaces are bound to it.

In order to manage its gathercast interfaces, a component can provide the GathercastController interface (see figure 12).

```
package org.etsi.uri.gcm.api.control;
public interface GathercastController extends CollectiveInterfaceController {
   public void notifyAddedGCMBinding (String gathercastItfName, Component owner, String
clientItfName)
      throws NoSuchInterfaceException, IllegalBindingException, IllegalLifeCycleException;
   public void notifyRemovedGCMBinding (String gathercastItfName, Component owner, String
clientItfName)
      throws NoSuchInterfaceException, IllegalBindingException, IllegalLifeCycleException;
   public List<Object> getGCMConnectedClients (String gathercastItfName)
      throws NoSuchInterfaceException;
}
```

Figure 12: Gathercast control API

In addition to extending the CollectiveInterfaceController interface, the GathercastController interface specifies the following operations:

- The notifyAddedGCMBinding operation notifies a component that a binding has been performed on one of its gathercast interfaces. This operation takes as parameters the gathercast interface name, a reference on the component which is bound to this gathercast interface and the client interface name of this component bound to the gathercast interface.
- The notifyRemovedGCMBinding operation notifies a component that a binding has been removed from one of its gathercast interfaces. This operation takes as parameters the gathercast interface name, a reference on the component which has removed its binding to this gathercast interface and the client interface name of this component which was bound to the gathercast interface.
- The getGCMConnectedClients operation returns a list of references on the interfaces connected to a given gathercast interface of a component. The single parameter of this operation is the gathercast interface name.

These operations may throw an org.objectweb.fractal.api.NoSuchInterfaceException exception if the specified gathercast or client interface does not exist. The notifyAddedGCMBinding and notifyRemovedGCMBinding operations may also throw an org.objectweb.fractal.api.control.IllegalLifeCycleException exception when the component is not in an appropriate state to perform the operation and an org.objectweb.fractal.api.control.IllegalBindingException exception in case of other errors related to bindings.

A component interface implementing GathercastController must be named `gathercast-controller`.

6.9 Migration control API

In distributed software systems, it is common to need to migrate some processes or some data from a node to another one. Within the context of the GCM, one may also need to do a migration on a component. In order to do that, a component can provide the MigrationController interface which allows the migration of this component from the node where it has been deployed to another one (see figure 13).

```
package org.etsi.uri.gcm.api.control;
public interface MigrationController {
   public void migrateGCMComponentTo (String nodeURL)
      throws MigrationException;
   public void migrateGCMComponentTo (URL nodeURL)
      throws MigrationException;
   public void migrateGCMComponentTo (Object node)
      throws MigrationException;
}
```

Figure 13: Migration control API

The MigrationController interface defines three operations, all named migrateGCMComponentTo, to do such a migration. The only difference between these operations is the way to address the destination node: the first operation takes as parameter the String representing the URL of the destination node, the second one takes as parameter the java.net.URL object corresponding to the URL of the destination node and at last, the third operation takes as parameter a Java object that references the destination node itself.

The org.etsi.uri.gcm.api.control.MigrationException exception may be thrown if the destination is not valid or when an error occurs during the migration.

A component interface implementing MigrationController must be named `migration-controller`.

6.10 Monitoring control API

Be informed in real time on the Quality of Service (QoS) of a component may be very useful. For instance, if a component has a bad QoS, then the user can decide to reconfigure the application by replacing this component by another one and thus improve the global performance. In term of GCM component, the QoS is defined as a set of various statistics (not specified in the present document) collected for each method of each interface of a component. For example, the collected statistics may contain the last execution time for a request of a given method or the average waiting time for a request for a given method. In order to get such statistics, a component can provide the MonitorController interface (see figure 14).

```
package org.etsi.uri.gcm.api.control;
public interface MonitorController {
   public void startGCMMonitoring ();
   public void stopGCMMonitoring ();
   public void resetGCMMonitoring ();
   public boolean isGCMMonitoringStarted ();
   public Object getGCMStatistics (String itfName, String methodName, Class<?>[] parameterTypes)
      throws NoSuchInterfaceException, NoSuchMethodException;
   public Map<String, Object> getAllGCMStatistics ();
}
```

Figure 14: Monitoring control API

The MonitorController interface provides two kinds of operations: one kind for managing the monitoring and the other one to get statistics.

The monitoring management is done with the operations startGCMMonitoring, stopGCMMonitoring, resetGCMMonitoring and isGCMMonitoringStarted, respectively to start the monitoring, stop it, reset it (i.e. delete all collected statistics) and check if the monitoring is running. These operations do not take parameters and do not return anything except the isGCMMonitoringStarted operation which returns a boolean indicating whether the monitoring is started.

Regarding the access to the statistics, the operation named getGCMStatistics can be used to retrieve the statistics of a method of a component interface. It takes as parameters the interface and the method name and the parameter types of the method. This operation then returns a Java object which contains the statistics corresponding to the requested method. The org.objectweb.fractal.api.NoSuchInterfaceException exception may be thrown when a specified interface does not exist or a java.lang.NoSuchMethodException exception if a specified method does not exist.

In addition of that, another operation, getAllGCMStatistics, provides all the statistics collected by the MonitorController interface. These statistics are returned through a java.util.Map object where the key is a single ID for each method of the component and allows to get a Java object containing the statistics of the method corresponding to this ID.

A component interface implementing MonitorController must be named `monitor-controller`.

6.11 Priority control API

By default, a component treats its requests, functional or non functional, in a FIFO order (First In First Out). Sometimes some non functional requests, like for instance life cycle management requests or reconfiguration requests, need to be executed before others. In order to define different priority levels for non functional requests, a component can provide the PriorityController interface (see figure 15).

```
package org.etsi.uri.gcm.api.control;
public interface PriorityController {
   public enum RequestPriority {
      F, NF1, NF2, NF3;
   }
   public void setGCMPriority (String itfName, String methodName, Class<?>[] parameterTypes,
RequestPriority priority)
      throws NoSuchInterfaceException, NoSuchMethodException;
   public RequestPriority getGCMPriority (String itfName, String methodName, Class<?>[]
parameterTypes)
      throws NoSuchInterfaceException, NoSuchMethodException;
}
```

Figure 15: Priority control API

The PriorityController interface defines an enum type, named RequestPriority, corresponding to the different levels of priority the requests of a method of a component interface may have:

- F, priority for functional requests. Requests always go at the end of the request queue.
- NF1, default priority of non functional requests. Requests also always go at the end of the request queue.
- NF2, priority for prioritized non functional requests. Requests pass the functional requests into the request queue but respect the order of other non functional requests.
- NF3, priority for most prioritized non functional requests. Requests pass all the other requests into the request queue.

This interface also specifies two operations to set and get the priority of requests of a given method.

The operation to set the priority of requests of a given method, named setGCMPriority, takes as parameters the interface name, the method name, the parameter types of the method and the value of the enum type RequestPriority which corresponds to the priority to set to the method requests.This operation does not return anything.

The operation to get the priority of requests of a given method, named getGCMPriority, also take as parameters the interface name, the method name and the parameter types of the method. This operation then returns the value of the enum type RequestPriority of requests of the asked method.

The org.objectweb.fractal.api.NoSuchInterfaceException exception may be thrown when a specified interface does not exist or a java.lang.NoSuchMethodException exception if a specified method does not exist.

A component interface implementing PriorityController must be named `priority-controller`.

7 GCM component runtime instantiation

The frameworks presented in the previous clauses allow one to use, introspect, configure and reconfigure existing components. In order to be useful, they must be completed with a framework to create new components. This clause defines such a framework, based on factories.

7.1 Factories API

In the instantiation framework specified in this clause, components are created by other components called component factories. The GCM model distinguishes between generic component factories, which can create several kinds of components, and standard component factories, which can create only one kind of components, all with the same component type. Generic and standard component factories can provide the GenericFactory interface and the Factory interfaces, respectively (see figure 16 - note that, in accordance with the rule defined in clause 6.1, both interfaces are functional interfaces, and not control interfaces).

```
package org.objectweb.fractal.api.factory;
interface GenericFactory {
   Component newFcInstance (Type t, Object controllerDesc, Object contentDesc)
      throws InstantiationException;
}
interface Factory {
   Type getFcInstanceType ();
   Object getFcControllerDesc ();
   Object getFcContentDesc ();
   Component newFcInstance () throws InstantiationException;
}
```

Figure 16: Instantiation API

The GenericFactory interface provides only one operation named newFcInstance. This operation takes as parameter the type of the component to be created, a descriptor of its controller part, and a descriptor of its content part. This operation creates a component corresponding to the given description, and returns its Component interface.

The Factory interface also provides a newFcInstance operation, but this operation does not take any parameter, which reflects the fact that all the components created by this operation have the same type, and the same controller and content descriptors. This information can be retrieved with the three other operations of this interface, named getFcInstanceType, getFcControllerDesc and getFcContentDesc.

In both interfaces, the component type must describe only the functional interfaces of the components to be created. The control interfaces of the components to be created must indeed be specified in the controller descriptors. The exact semantics of the controller and content descriptors, in both interfaces, is however left unspecified in this version of the GCM component model specification.

Note that, in both interfaces, the newFcInstance operation does not necessarily create a new component instance each time it is invoked. It can also, for example, always return the same instance (this is the singleton pattern). The components created by a factory must be created in the same address space as the factory component. But the exact location of the created components, in this address space, is voluntarily not specified. In particular, it is not ensured that the components created by a factory are automatically added to the parent component(s) of the factory component.

The org.objectweb.fractal.api.factory.InstantiationException exception must be thrown when a component cannot be created, in the newFcInstance operations of the Factory and GenericFactory interfaces.

A component interface implementing GenericFactory must be named `generic-factory`. A component interface implementing Factory must be named `factory`.

7.2 Templates

A template is a special kind of standard factory component that creates components that are quasi "isomorphic" to itself. More precisely, the components created by a template component must have the same functional client and server interfaces as the template component (except for the Factory interface, which is provided by the template, but not necessarily by its instances), but can have arbitrary control interfaces. The components created by a template component also have the same attributes as the template. A template component may contain several sub template components, bound together through bindings. The components created by such a template component are components that contain as many sub components as sub templates in the template, bound together as the sub templates are bound in the template. If some sub templates are shared, the corresponding sub components in the components created by the template will also be shared.

If a generic factory component is able to create template components, then it must be possible to create a template component with a operation invocation, on this generic factory, of the form newFcInstance(type, templateControllerDesc, {controllerDesc, contentDesc}), where type describes the functional client and server interfaces of the components that the template will create, templateControllerDesc is the descriptor of the controller part of the template component to be created, and controllerDesc and contentDesc are the descriptors of the controller and content parts of the components that the template will create (the brackets denote an array).

Template components are useful in only one case, namely when several identical components must be created from a textual representation, such as an Architecture Description Language definition. In this case, instead of parsing the textual representation each time an instance must be created, it can be more efficient to parse the text file(s) and to create a corresponding template only once, and then to instantiate the template each time an instance is needed. In all other cases, using templates is equivalent, but generally less efficient, than not using them.

7.3 Bootstrap

According to the above framework, components are created from component factories. But how are created component factories? They can be created from other component factories, but this leads to an infinite recursion. In order to stop it, a bootstrap component factory, which does not need to be created explicitly, and which is accessible from a "well-known" name, is necessary. This bootstrap component factory must be able to create several kinds of components, including component factories. In other words, it must provide the GenericFactory interface.

This bootstrap component must be accessible from the getBootstrapComponent static method, defined in the org.etsi.uri.gcm.api.GCM class. This method must not take any parameter, and must return the Component interface of the bootstrap component.

8 GCM Typing

This clause defines a simple type system for components and component interfaces. This type system reflects the main characteristics of component interfaces, introduced in clause 5, i.e. their name, their language type, and their role (client or server). It also introduces two new characteristics named contingency and cardinality.

8.1 Component interfaces contingency and cardinality

The contingency of an interface indicates if the functionality corresponding to this interface is guaranteed to be available or not, while the component is running:

- The operations of a mandatory interface are guaranteed to be available when the component is running. This semantic is obvious for a server interface. For a client interface, which does not have a functionality of its own, it means that the interface must be bound, and that it must be bound to a mandatory interface. As a consequence, a component with mandatory client interfaces cannot be started until all these interfaces are bound to other mandatory interfaces.

- The operations of an optional interface are not guaranteed to be available. This can happen, for a server interface, when the complementary internal interface is not bound to a sub component. This can also happen, for a client interface, when this interface is not bound.

The cardinality of an interface type T may be:

- The singleton cardinality which means that a given component must have exactly one interface of type T.

- The collection cardinality which means that a given component may have an arbitrary number of interfaces of type T. All these interfaces must have a name that begins with the name specified in T (see clause 8.2). Since there is a priori an infinite number of such interfaces, these interfaces cannot all be created at the same time: they must be created lazily, during invocations of the getFcInterface and bindFc operations.

  EXAMPLE: If the name specified in T is listener, then an invocation to getFcInterface("listener11") or to bindFc("listener11", s) will create an interface named listener11, if it does not already exist.

  This interface may be removed automatically when it is no longer used by any binding.

- The multicast cardinality which means that a given component must have exactly one interface of type T but transforms each invocation into a set of invocations (see clause 6.7).

- The gathercast cardinality which means that a given component must have exactly one interface of type T but transforms a list of invocations into a single invocation (see clause 6.8).

Mandatory are designed for components that absolutely require other components to work. Optional interfaces are useful for components which may also use other components, if they are present. For example, a parser component absolutely needs a lexer component, but can work with or without a logger component. Collection interfaces are useful for components with a variable number of required components of the same type, such as a menu component and its associated menu item components, a model component and its listener components (in the MVC model), and so on. Multicast interfaces are useful to enable parallel invocations, with automatic distribution of invocation parameters. Gathercast interfaces are useful to define synchronization barriers and to gather incoming data.

8.2 Type system API

In the type system specified here, a component type is just a set of component interface types. A component type is represented by the ComponentType interface (see figure 17). This interface defines a getFcInterfaceTypes operation, which returns the set of component interface types in this component type, as an array. It also defines a getFcInterfaceType operation, which returns the component interface type whose name is given as parameter (this operation must throw the NoSuchInterfaceException if the requested interface type does not exist).

A component interface type is represented by the InterfaceType interface which may be cast to GCMInterfaceType. Such a type is made of a name, a signature, a role, a contingency and a cardinality. The name is the name of component interfaces of this type. The signature is the name of the language interface type that is implemented by component interfaces of this type (for a client interface, an empty signature means that this client interface can be connected to any server interface). The role indicates if component interfaces of this type are client or server interfaces. The contingency indicates if the functionality of interfaces of this type is guaranteed to be available or not. Finally, the cardinality indicates if the interfaces of this type have a singleton, collection, multicast or gathercast cardinality.

Component and component interface types can be created by using a type factory, represented by the TypeFactory interface or the GCMTypeFactory interface. Indeed the TypeFactory interface provides two operations to create component interface types and component types whereas the GCMTypeFactory interface, which extends the TypeFactory interface, provides one more operation to create component interface types. A component interface implementing TypeFactory or GCMTypeFactory must be named `type-factory`.

```
package org.objectweb.fractal.api.type;
interface ComponentType extends Type {
   InterfaceType[] getFcInterfaceTypes ();
   InterfaceType getFcInterfaceType (String itfName) throws NoSuchInterfaceException;
}
interface InterfaceType extends Type {
   String getFcItfName ();
   String getFcItfSignature ();
   boolean isFcClientItf ();
   boolean isFcOptionalItf ();
   boolean isFcCollectionItf ();
}
interface TypeFactory {
   InterfaceType createFcItfType (String name, String signature, boolean isClient,
      boolean isOptional, boolean isCollection) throws InstantiationException;
   ComponentType createFcType (InterfaceType[] itfTypes) throws InstantiationException;
}
package org.etsi.uri.gcm.api.type;
interface GCMInterfaceType extends InterfaceType {
   String getGCMCardinality ();
   boolean isGCMSingletonItf ();
   boolean isGCMCollectionItf ();
   boolean isGCMMulticastItf ();
   boolean isGCMGathercastItf ();
}
interface GCMTypeFactory extends TypeFactory {
   InterfaceType createGCMItfType (String name, String signature, boolean isClient,
      boolean isOptional, String cardinality) throws InstantiationException;
}
```

Figure 17: Typing API

A component of type T must have as many external interfaces as described in T (and, in particular, in the interface cardinalities), and all these interfaces must have the name, language type and role described in the corresponding component interface type. Likewise, if this component also exposes its content, and in particular its internal interfaces, then it must also have, at most, as many internal functional interfaces as described in T, and each of these interfaces must have the name, language type and role described in the corresponding component interface type. This implies that each internal functional interface has a complementary external interface of the same name, signature, contingency and cardinality, and of opposite role (but the converse is not necessarily true). Note that this property is ensured by the type system specified in this clause: in the general case, nothing more than what is explicitly stated in clause 8.1 is ensured (and so an internal interface may not have a complementary external interface).

Note that if, in general, the number of interfaces of a Fractal component may change during its life time, the number of interfaces of a Fractal component that uses the type system presented here cannot change during its lifetime (except for interface collections). Indeed the ComponentType and InterfaceType interfaces do not offer any operations to modify an existing type, and the other interfaces specified in the present document do not offer an operation to change the type of a component or of an interface. But a GCM component may perfectly provide a setFcType operation, if needed, since the GCM model is extensible.

8.3 Sub typing relation

This clause defines a sub typing relation for component and component interface types, based on substitutability. This relation provides a sufficient (but not necessary) condition such that if a component type T1 is a sub type of T2, then a component of type T1 can replace a component of type T2 in any environment, this environment (other components and bindings) being left unchanged, and both components being seen as black boxes.

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

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

A component type T1 is a sub type of a component type T2 if and only if each client interface type defined in T1 is a sub type of an interface type defined in T2, and each server interface type defined in T2 is a super type of an interface type defined in T1.

Annex A (normative): Java API

The GCM Java API is contained in an archive file (ts_102830v010101p0.zip) which accompanies the present document, and containing the Java source files for the API. The archive file is named Fractal-GCM-Management-API.zip

This archive file contains the Fractal GCM Management API expressed in the Java programming language.

It contains:

- LICENSE.txt: The license under which this module is (GNU GPLv2)
- src: The Java source folders of the GCM Management API, organised in two parts:
  - Under org/objectweb/fractal/api/, the original Fractal API specification
  - Under org/etsi/uri/gcm/api/, the GCM extensions to Fractal.

Below is the detail of the packages and Java Interfaces found in the two parts:

- `src/org/objectweb/fractal/api/` which specifies the component and component interface concepts.
  NoSuchInterfaceException.java Type.java
  Interface.java Component.java
- `src/org/objectweb/fractal/api/control/` which specifies some Fractal component interfaces to control components.
  ContentController.java SuperController.java LifeCycleController.java
  NameController.java AttributeController.java BindingController.java
  IllegalBindingException.java IllegalLifeCycleException.java IllegalContentException.java
- `src/org/objectweb/fractal/api/factory/` which specifies some basic component interfaces to instantiate components.
  InstantiationException..java GenericFactory.java Factory.java
- `src/org/objectweb/fractal/api/type/` which specifies a basic type system for components and component interfaces.
  ComponentType.java TypeFactory.java InterfaceType.java
- `src/org/etsi/uri/gcm/api/` which provides a class to get a bootstrap component.
  GCM.java
- `src/org/etsi/uri/gcm/api/control/` which specifies some GCM component interfaces to control components.
  GCMLifeCycleController.java MonitorController.java MulticastController.java
  MigrationException.java MigrationController.java GathercastController.java
  PriorityController.java CollectiveInterfaceController.java
- `src/org/etsi/uri/gcm/api/type/` which defines GCM type system specificities for component interfaces.
  GCMInterfaceType.java GCMTypeFactory.java
- `src/org/etsi/uri/gcm/util/` which provides utilities to facilitate GCM API's use.
  GCM.java

Annex B (informative): Namespaces

This annex describes how the respective namespaces of Fractal and GCM should be used and managed for the present documents and its future versions.

In a JAVA API (Application Programming Interface), namespaces are used to structure the names of the Java objects (interfaces, classes, exceptions, etc.).

B.1 Description

The GCM Management API contains two parts:

- The first part consists in a set of Fractal interface definitions. These interfaces lie in the **org.objectweb.fractal** namespace. They have been previously adopted by the fractal community as the official Fractal specification (that is not a standard). We have reproduced them (in full and unchanged) inside the GCM Management API in order to make the standard self-contained.

- The second part is a set of GCM definitions, lying in the **org.etsi.uri.gcm** namespace. In term of Fractal conformance levels, the GCM definitions are extensions of the Fractal specifications.

Both parts are present in the archive file referred in annex A.

In order to use the GCM Management API, for example for building and compiling an implementation of the GCM, it is sufficient to use the archive file from annex A. There is no need to get any additional interface definitions form Fractal.

But of course, having the Fractal interfaces included in the GCM Management API allows the GCM implementation developers to use existing implementations of the Fractal API.

B.2 Versioning

The definitions lying in the **org.etsi.uri.gcm namespace** should not be extended or modified.

Evolution of the API can happen in two ways:

- ETSI (meaning some TG of ETSI in charge of the GCM standard, being TCGRID or some followers technical group) may decide to issue a new version of the GCM Management API.

  In this case, ETSI will create a new namespace, e.g. **org.etsi.uri.gcm-v2**, and not modify the existing gcm namespace content.

- The Fractal community may issue a new version of the Fractal API specification.

  In this case, ETSI may decide to stay with the current version of the GCM, built on top of the current version of Fractal.

  Or ETSI may decide to follow the Fractal evolution, and build a new version of GCM incorporating the new Fractal version. This new CM will also be set in a new etsi namespace like in the previous case.

Annex C (informative): Examples

This annex shows how a Java GCM platform can be used, in order to illustrate how the APIs defined in the present document can be used to create, assemble and reconfigure component configurations.

The example used throughout this clause is a very simple application made of two primitive components inside a composite component (see figure C.1). The first primitive component is a "server" component that provides an interface s of type S. The other primitive component is a "client" component, bound to the previous server interface.

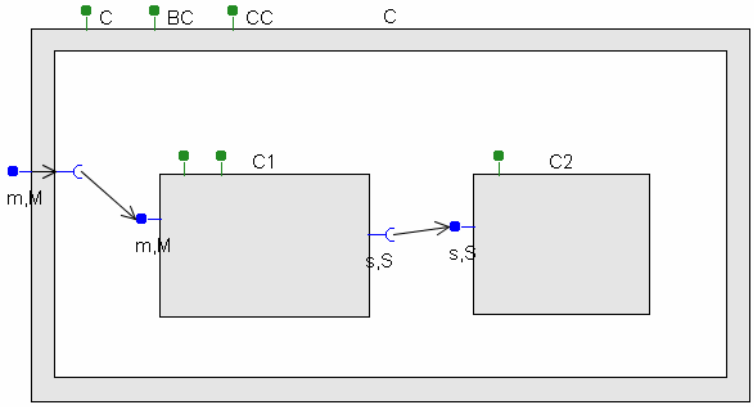


Figure C.1: A sample component based application

C.1 Instantiation

The above components can be instantiated as follows. The first step is to create the component and component interface types. In order to do this, we get a reference to the bootstrap component, and then to its TypeFactory interface:

```
Component boot = GCM.getBootstrapComponent();
TypeFactory tf = (TypeFactory)boot.getFcInterface("type-factory");
We can now create the types of the root, client and server components as follows:
ComponentType rType = tf.createFcType(new InterfaceType[] {
   tf.createFcItfType("m", "M", false, false, false)
});
ComponentType cType = tf.createFcType(new InterfaceType[] {
   tf.createFcItfType("m", "M", false, false, false),
   tf.createFcItfType("s", "S", true, false, false)
});
ComponentType sType = tf.createFcType(new InterfaceType[] {
   tf.createFcItfType("s", "S", false, false, false)
});
```

We could now create the components directly, but we will use intermediate template components here, in order to show how they can be used. These component templates can be created as follows:

```
GenericFactory gf = (GenericFactory)boot.getFcInterface("generic-factory");

Component rTmpl = gf.newFcInstance(
   rType, "compositeTemplate", new Object[] {"composite", null});
Component cTmpl = gf.newFcInstance(
   cType, "template", new Object[] {"primitive", "CImpl"});
Component sTmpl = gf.newFcInstance(
   sType, "template", new Object[] {"primitive", "SImpl"});
```

Here the template (resp. compositeTemplate) descriptor is supposed to describe components with a BindingController interface (resp. with a BindingController and a ContentController interfaces). The primitive and composite descriptors are supposed to describe similar components, but with an additional LifeCycleController interface. Finally, CImpl and SImpl are the names of the Java classes of the GCM components that will be encapsulated in the client and server components. The CImpl class, for example, has the following form:

```
public class CImpl implements M, BindingController {
   private S s;
   public String[] listFc () { return new String[] { "s" }; }
   public Object lookupFc (String name) {
      if (name.equals("s")) return s;
      return null;
   }
   public Object bindFc (String name, Object itf) {
      if (name.equals("s")) s = (S)itf;
   }
   public Object unbindFc (String name) {
      if (name.equals("s")) s = null;
   }
   // ...
}
```

We can then either instantiate each template one by one, put the resulting primitive components inside the composite component, connect all these components, and finally start them. But we can also put the primitive templates inside the composite template, connect these templates together, and then instantiate the whole application by just instantiating the composite template component. This is what we do here.

We begin by putting the primitive template components inside the composite one:

```
ContentController cc = (ContentController)rTmpl.getFcInterface("content-controller");
cc.addFcSubComponent(cTmpl);
cc.addFcSubComponent(sTmpl);
```

We then bind the internal client interface m of the composite template to the server interface m of the client template:

```
((BindingController)rTmpl.getFcInterface("binding-controller"))
   .bindFc("m", cTmpl.getFcInterface("m"));
```

Finally, we bind the client interface s of the client template to the server interface s of the server template:

```
((BindingController)cTmpl.getFcInterface("binding-controller"))
   .bindFc("s", sTmpl.getFcInterface("s"));
```

At this stage the template components are like the components depicted in figure C.1, with just an additional Factory interface. Now that the template components have been created and bound to each other, the application components can be instantiated and bound to each other automatically, by just calling the newFcInstance method on the root template component:

```
Component r = ((Factory)rTmpl.getFcInterface("factory")).newFcInstance();
```

All the application components can now be started automatically by just calling the startFc method on the root application component (here we assume a stronger semantic than the default one for the startFc method, i.e. we assume it to be recursive - see clause 6.5):

```
((LifeCycleController)r.getFcInterface("lifecycle-controller")).startFc();
```

C.2 Reconfiguration

Let us suppose we want to dynamically change the server component. In order to do this, we need to unbind the client component, remove the server component, create a new server component, add the server component in the composite component, and finally bind the client component to the new server. But the binding and component removals cannot be done while the client and the composite component, respectively, are not stopped. So we must first stop these components (here again we assume this method to be recursive; we also assume that it does not change the states of the components, and that method calls to functional interfaces while the components are stopped are only suspended until the components are restarted):

```
((LifeCycleController)r.getFcInterface("lifecycle-controller")).stopFc();
```

We then retrieve the references of the client and server components:

```
Component c = ((Interface)((BindingController)r.
   getFcInterface("binding-controller")).lookupFc("m")).getFcItfOwner();
Component s = ((Interface)((BindingController)c.
   getFcInterface("binding-controller")).lookupFc("s")).getFcItfOwner();
```

We can now unbind the client and server components, and remove the server component from the composite component (we assume a strong semantic for removeFcSubComponent):

```
((BindingController)c.getFcInterface("binding-controller")).unbindFc("s");
((ContentController)r.getFcInterface("content-controller")).removeFcSubComponent(s);
```

We can now create the new server component. Instead of using a template component for doing that, as in clause C.1, we use here the bootstrap generic factory directly:

```
Component newS = gf.newFcInstance(sType, "primitive", "NewSImpl");
```

We can now add this new component in the composite component, bind it to the client component, and finally resume the application's execution (we make the same semantic hypotheses as in clause C.1 for the addFcSubComponent and startFc methods):

```
((ContentController)r.getFcInterface("content-controller")).addFcSubComponent(newS);
((BindingController)c.getFcInterface("binding-controller")).bindFc("s", newS);
((LifeCycleController)r.getFcInterface("lifecycle-controller")).startFc();
```

History

Document history		
V1.1.1	March 2010	Publication