

F-F-F: A new factory for Fractal/GCM

Luc Hogie, Bastien Sauvan, Paul Naoumenko

February 24, 2011

Contents

1	Introduction	3
1.1	Context	3
1.1.1	Fractal	3
1.1.2	ProActive	3
1.1.3	GCM	3
1.1.4	ADL and the component factory	4
1.2	Description of the problem	4
1.3	Compliance with previous work	5
2	Design of the new factory	5
2.1	Execution workflow	5
2.1.1	Lexical analysis (parsing): from XML to DOM	5
2.1.2	Semantic analysis: from DOM to semantic description	5
2.1.3	The backend: from semantic description to components	6
2.2	Input format	6
2.2.1	ADL description	6
2.2.2	Passing arguments	6
2.3	Extensibility	7
2.3.1	Basic factory	7
2.3.2	Addition of multiple inheritance	8
2.3.3	Addition of controller	8
2.3.4	Addition of the support for shared components	9
2.3.5	Addition of arguments	9
2.3.6	Addition of attributes	9
2.3.7	Addition of comments	10
2.3.8	GCM	10
2.4	Files	10
2.5	Practical cases	10
2.5.1	GCM virtual nodes	10
3	Proposals for a better ADL	10
3.1	The DTD declaration is no longer required	11
3.2	Storing the ADL in files rather than in Java resources	11
3.3	the <i>definition</i> XML element	11
3.4	the <i>content</i> XML element	11
4	How to...	11
4.1	Perform a new verification	11
4.2	Add a new attribute to an existing XML element	12
4.3	Add a new XML element	12
4.3.1	Lexical analysis	13
4.3.2	Semantic analysis: the Description	13
4.4	Component creation	13
4.4.1	Example: the interface element	13
4.5	Modify the way attributes values are processed	13

5	Features	14
5.1	Already supported	14
5.2	Under progress	14
5.3	Will be supported in the future	14
6	Conclusion and future works	14

The aim of this document is to provide developers and users with sufficient documentation for the new Fractal/GCM factory.

1 Introduction

1.1 Context

In a few words, this work consists in the design and implementation of a new component factory for the GCM component-based grid architecture, which is an extension of the Fractal specification, and whose ProActive constitutes the only implementation available today.

1.1.1 Fractal

According to its authors, Fractal is a modular, extensible and programming language agnostic component model that can be used to design, implement, deploy and reconfigure systems and applications, from operating systems to middleware platforms and to graphical user interfaces.

<http://fractal.ow2.org/>

Fractal is mostly a Research system which gathers a large community of people working on various aspect of the component-oriented paradigm.

1.1.2 ProActive

According to its authors (OASIS Research team) ProActive is a GRID Java library for parallel, distributed, and concurrent computing, also featuring mobility and security in a uniform framework. ProActive runs on Local Area Network (LAN), on clusters of workstations, or on Internet Grids.

<http://proactive.inria.fr/>

1.1.3 GCM

The Grid Component Model (GCM) is an extension of the Fractal component model.

According to ETSI TS 102 830, 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.

1.1.4 ADL and the component factory

ADL stands for Architecture Description Language. One usually refers to ADL as *architecture description*. ADL is not part of the Fractal specification.

The ADL enables the description of a component system. In the particular case of Fractal, the component system is a tree, hence the adequation of XML (note that Fractal's ADL supposedly does not depend on XML).

The XML dialect introduced with Fractal as its ADL makes use of a variety of elements and attributes. Fractal ADL specification can be found at the following URL:

<http://fractal.ow2.org/tutorials/adl/index.html>

Basically, an ADL description looks like that:

```
1 <definition name="luc.demo.ExampleComponent">
2   <interface name="r" role="server" signature="luc.demo.ExampleComponent"/>
3   <component name="impl">
4     <interface name="s" role="server" signature="luc.demo.ExampleComponent"/>
5     <content class="luc.demo.BasicExampleComponent"/>
6   </component>
7   <binding client="this.r" server="impl.s" />
8 </definition>
```

Fractal assumes that the ADL description is not stored in a file but in a Java resource, whereas it should be stored in a standard file, at least for the sake of independence to Java. In spite of it, Fractal's way of referring to an ADL resource is to call it an *ADL file*.

The component factory of Fractal is a piece of code within Fractal implementations whose the role is to build a component out of its XML description. The reference implementation of GCM (ProActive) comes with its own factory which is an extension of the Fractal factory.

The implementation of the Fractal factory is cumbersome: it consists of about 150 Java classes and configuration files. organized in numerous packages. Besides, in order to implement its own extensions, GCM adds about 40 classes and description files.

A description of the Fractal factory can be found as a chapter of the book "Intergiciel et Construction d'Applications Réparties, ICAR (Ed.) (2006)":

hal.archives-ouvertes.fr/docs/00/15/50/90/PDF/main.pdf

1.2 Description of the problem

The default Fractal factory (and its derivative) suffers from a number of problems, which lead to high frustration when it comes to deal with it.

- for the sake of modularity, its design involves many more concepts than necessary for its purpose, most often these concepts are imprecise;
- its implementation is made of numerous non-intuitive constructs;
- it comes with little documentation.

The objective of the new factory is then to make a factory that is shorter, cleaner, easier to understand/modify, and faster than the original Fractal/GCM factory. In particular, in order to avoid confusion, we intend to use as little concepts as strictly necessary. Also, we plan to document it so as a new user involved in its development/extension will have a minimal amount of stress getting into the code.

1.3 Compliance with previous work

On the one hand, the new factory must be able to load ADL files as they are described by the Fractal and GCM specifications. It must also take into account extensions which are not part of official standard but which are of interest for the Oasis research team. Non-official extensions which are not of interest for Oasis will not be considered, unless the new factory is foreseen as a replacement of the default factory within Fractal. In such case, the authors of these extensions should be asked a contribution to porting their code.

On the other hand, the new factory may introduce constructs which constitute improvements over the GCM ADL in its current version. As such, ADL definitions tailored for the new factory may not be loadable by the default GCM factory.

Also the new factory may expose a different API than the default GCM factory does, considering that the new API suits better its very purpose.

2 Design of the new factory

2.1 Execution workflow

2.1.1 Lexical analysis (parsing): from XML to DOM

XML parsing to DOM is done by the parser built-in into the Java Development Kit. Because DOM structures are not handy to manipulate, the parsing process goes a little further: it converts the DOM structure into a lightweight custom tree structure whose the basic signature is:

```
1 class XMLNode
2 {
3     String getName();
4     Map<String, String> getAttributes();
5     List<XMLNode> getChildrenNodes();
6 }
```

which is more convenient to use than the general purpose DOM data structure.

The parser takes as input an XML/ADL description and turns it into a tree-like object-oriented structure, which is similar to DOM. The parsing involves no other verification than syntactic ones. XML validation is not performed. No DTD is then required.

Note that at this stage, the value of attributes not parsed/checked.

2.1.2 Semantic analysis: from DOM to semantic description

Each XML element type into the ADL description is represented by a **Description** class. Because each XML element declaration refers to a specific concept, each of these concept is represented as a specific sub-class of the **Description** class. Then at runtime, each XML element is represented as an instance of a sub-class of the class **Description**, as follows:

definition and component respectively describe the root component of the component tree; and a sub-component of a given component; they are both represented by the class **ComponentDescription**;

interface describes an interface of a given component, it is represented by the class `InterfaceDescription`;

binding describes an interface binding involving two interfaces of two parent/child components; it is represented by the class `BindingDescription`;

attributes describes attributes in a given components; it is represented by the class `AttributesDescription`;

content describes the content class for the implementation code of a given component. It is *not* represented by any description. Instead it is represented as a field into the class `ComponentDescription`.

The creation of the description is carried on by static methods in the subclass of the class `Description`. Each of these static method is in charge of creating the description object for the host class.

The factory then turns this structure into a high level representation of the description. This conversion makes a number of verification on the content of the XML, including all the verifications performed by a validating XML parser. But the process goes beyond DTD-based XML validation: the content of XML attributes are also parsed/checked at this stage.

2.1.3 The backend: from semantic description to components

The creation of the component out of the semantic description of the component system is done using Fractal and GCM APIs. Because of this, components created by the new factory are of the very same nature than those created by the original factory. They can perfectly interoperate.

All the code is located in the class `Backend`.

The description is finally passed to a backend in charge of manipulating Fractal API so as to create the actual component system.

2.2 Input format

```
Component createComponent(JavaResource adl, Map<String, String> args,
Object deployDescr)
```

The input of the factory comes in the form of files on the disk.

This constitute a difference with the original factory which requests as input an ADL file as well as a *context* aimed at stored unspecified information.

2.2.1 ADL description

The ADL description comes in the form of an XML file.

2.2.2 Passing arguments

Arguments value are passed to the factory as an associative table which link the name of an argument to its value.

For convenience purpose, F-F-F comes with a parser that allows the arguments to be specified in a file made of *key = value* pairs. The format of the argument file is then:

$$\begin{aligned}
k_1 &= v_1 \\
k_2 &= v_2 \\
&\dots \\
k_n &= v_n
\end{aligned}$$

Java programmers will notice that the syntax is the one used for *properties* files.

2.3 Extensibility

The objective is to build a factory which provides an extension mechanism so that the addition of new concepts can be done by addition of code, with no single alteration of the existing one.

The description may be subclassed. A new backend that takes into account the new properties of the descriptions may be implemented.

2.3.1 Basic factory

The basic implementation considers only the concepts of:

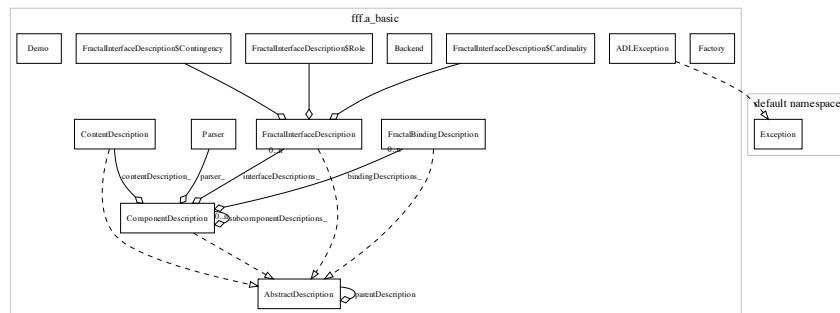
definition ;

sub-components ;

content ;

interfaces ;

bindings .



The basic factory is articulated around about 10 classes.

ADLException is thrown when an error is found in the ADL;

Backend creates a component out of its description;

InterfaceDescription ;

BindingDescription defines that an interface is bound to two components;

ComponentDescription defines that a component has bound interfaces, sub-components or a content;

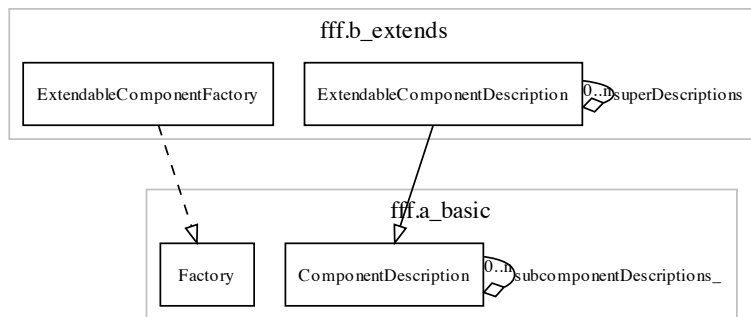
Parser builds a DOM-like structure out of an XML text;

ContentDescription ;

AbstractDescription defines what every description need: a parent description, a way to perform verification, and a way to convert it to XML;

Factory feeds the input of the backend with the output of the parser and return the component.

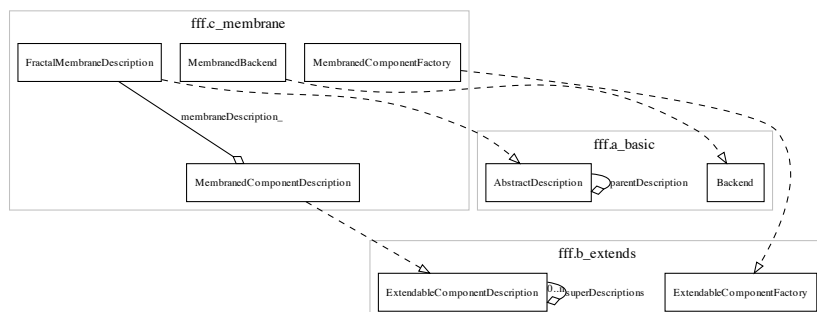
2.3.2 Addition of multiple inheritance



In order to add support for multiple inheritance, the `ComponentDescription` class must be derived so as to define that a component may have super components. The new description class must be declared somewhere, hence the derivation of the factory class.

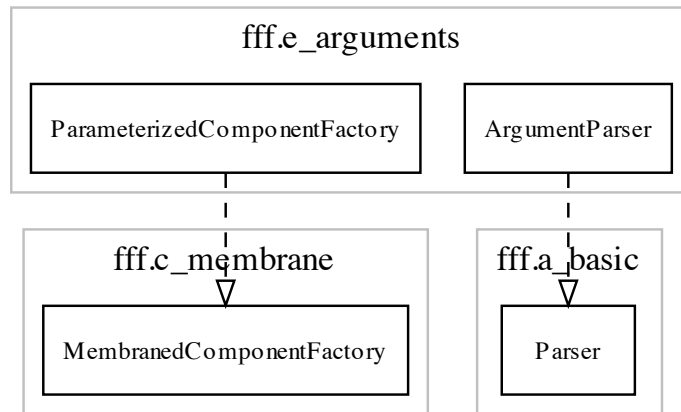
The an extendable component description features a method `resolveInheritance()` which performs a reversed breadth-first search (BFS) and updates the child description with its ancestors' features.

2.3.3 Addition of controller

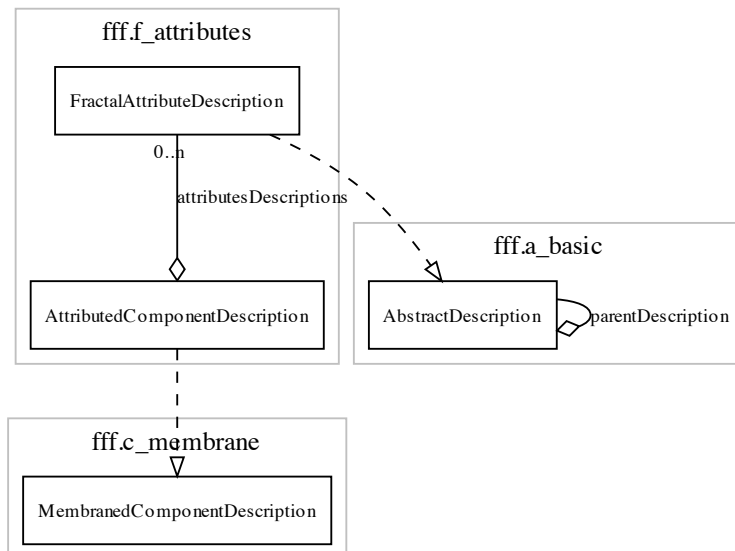


2.3.4 Addition of of the support for shared components

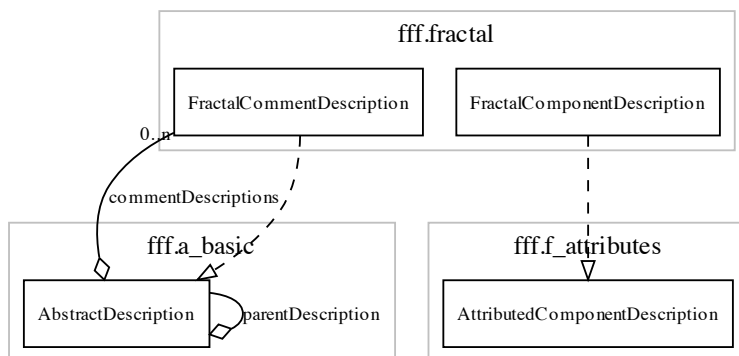
2.3.5 Addition of arguments



2.3.6 Addition of attributes



2.3.7 Addition of comments



Now complies with the Fractal specification.

2.3.8 GCM

Collection interfaces support for multicast/gathercast interfaces support for virtual nodes support for bindings to web-services

2.4 Files

2.5 Practical cases

2.5.1 GCM virtual nodes

```

1 <interface ...>
2     <exportedVirtualNodes>
3         <exportedVirtualNode name='VN1'>
4             <composedFrom>
5                 <composingVirtualNode
6                     component='client '
7                     name='client -node'>
8             </composedFrom>
9         </exportedVirtualNode>
10    </exportedVirtualNodes>
11 </interface>
  
```

3 Proposals for a better ADL

The ADL as it comes from the Fractal specification presents a number of weaknesses and imprecisions. For example, it is not clear that the *definition* element is intrinsiquely different from the *component* one. Also, the usefulness of the *arguments* attribute within a *definition* element is to be discussed. This section addresses these imprecisions and proposes a number of enhancements of the ADL.

3.1 The DTD declaration is no longer required

It is not the responsibility of the XML file to declare which piece of software will be in charge of validating it. Instead the parser is aware of the structure constraints of the XML code. The DTD is dynamically inserted into the XML text.

3.2 Storing the ADL in files rather than in Java resources

CGM reference implementation is written in Java. This said, the implementation language of a given software should remain a detail to the user. In Fractal, the input data (the ADL file) is expected to be a Java resource, making Fractal applications dependant of Java, even at the user-level. This somehow breaks the rule that Fractal is a specification potentially be implemented in any language.

Instead, the input data should be expected to be a file.

For compatibility, this will not be implemented in F-F-F.

3.3 the *definition* XML element

A component is described by the use of the *component* element, except at the root-level where the component is to be referred as a *definition*. This choice comes from the need of the *definition* element to allow two attributes that the *component* element does not: *arguments* and *extends*.

In fact the definition of the argument should not be included into the description of the root component. The arguments should be defined *before* they are of potential use.

3.4 the *content* XML element

In the description of a component, only one *content* element is allowed. Moreover this element accepts only one attribute, the *class* attribute.

```
1 <component name="boh">
2     ...
3     <content class="java.util.ArrayList" />
4     ...
5 </component>
```

Instead, the content information for a component should be expressed as an attribute in the **component** description. This would look like this:

```
1 <component name="boh" content="java.util.ArrayList">
2     ...
3 </component>
```

4 How to...

4.1 Perform a new verification

Verifications are all performed in the `check()` method of the description class and its subclass. The programmer redefining the `check()` method to add new verifications must not forget to include a call to `super.check()`.

Adding a new verification consists in adding a line to the `check()` method or redefining it. Typically, such line is in the form:

```
1  if (!condition)
2      throw new ADLException("condition failed");
```

For convenience purpose, the use of the following construct is encouraged:

```
1  Assertions.ensure(condition, "condition failed");
```

4.2 Add a new attribute to an existing XML element

The concept of XML attribute is not modelled in F-F-F. The instantiation of a description object requires the corresponding XML node. In order to take into consideration a new attribute, you need to extend a description class and add the adequate code in its constructor. What this code does with the new attribute depends on your application.

For example, consider the case of the addition for the support of inheritance. A new attribute **extends** was introduced in the ADL. We need to add new code to the constructor of the component description class so that it will take into account this new attribute. In particular, it will consider the value of the attribute as a sequence of entity names. So it will consider each of this entity name and find/create the corresponding entity. These entities are stored in a field of type `List<ComponentDescription>`.

4.3 Add a new XML element

XML elements have a corresponding semantic description class, which is a subclass of the class `Description`. Thus, to add a new XML element, you need to define new description class.

- Mandatory attributes have no default value. They must be assigned at construction time.
- Optional attributes have a default value. They are reassigned only if they are explicitly given by the user.

The instance of the description is created by the semantic analyzer.

Let us take the method `InterfaceDescription createInterfaceDescription(XMLNode n)` as an example:

```
1  String name = n.getAttributes().get("name");
2  Role role = n.getAttributes().get("role").equals("client")
3      ? Role.CLIENT : Role.SERVER;
4  Class<?> signature =Clazz.findClassOrFail(n.getAttributes().get("signature"));
5  InterfaceDescription id = new InterfaceDescription(name, role, signature);
6
7  if (n.getAttributes().get("contingency") != null)
8  {
9      id.setContingency(n.getAttributes().get("contingency").equals("mandatory")
10         ? Contingency.MANDATORY : Contingency.OPTIONAL);
11 }
12
```

```

13 if (n.getAttributes().get("cardinality") != null)
14 {
15     id.setCardinality(n.getAttributes().get("cardinality").equals("singleton")
16         ? Cardinality.SINGLETON : Cardinality.COLLECTION);
17 }

```

4.3.1 Lexical analysis

Nothing to do at this step.

4.3.2 Semantic analysis: the Description

- fields
- checking
- toXMLNode()

And the way to instantiate the description.

4.4 Component creation

Needs to update the backend.

4.4.1 Example: the interface element

Mandatory attributes must be provided at construction time:

```

1 String name = n.getAttributes().get("name");
2 Role role = n.getAttributes().get("role").equals("client") ? Role.CLIENT : Role.SERVER;
3 Class<?> signature = Clazz.findClassOrFail(n.getAttributes().get("signature"));
4 InterfaceDescription id = new InterfaceDescription(name, role, signature);

```

Optional elements are set right after construction, if the user provided a value of them:

```

1 if (n.getAttributes().get("contingency") != null)
2 {
3     id.setContingency(n.getAttributes().get("contingency").equals("mandatory")
4         ? Contingency.MANDATORY : Contingency.OPTIONAL);
5 }

```

4.5 Modify the way attributes values are processed

XML attribute values are parsed by the method:

`String replaceArgument(String v, Map<String, String> argumentValues)` method in the `Attributes` class. Its default behavior is to replace the pattern `${name}` by the value of the `name` argument by its value found in the associative map `argumentValues`.

Override it to get the behavior you want.

5 Features

5.1 Already supported

The following features are already supported by the new factory:

- component interfaces (singleton);
- component attributes;
- interface bindings;
- sub-components;
- ADL multiple inheritance;
- ADL arguments;
- comment elements (partial).

5.2 Under progress

The following list enumerates the features that are currently under development:

- support for external definitions;
- support for componentized membranes (Paul Naoumenko's topic);
- support for collection interfaces (GCM).

5.3 Will be supported in the future

The following list enumerates the features that are will be developed in the near future, ordered by importance:

1. support for collection components (Amine Rouini's topic);
2. support for multicast/gathercast interfaces (GCM);
3. support for virtual nodes (GCM);
4. support for shared components (Fractal);
5. support for bindings to web-services (GCM).
6. support for loggers (not implemented in the default factory);
7. support for coordinates (not implemented in the default factory).

6 Conclusion and future works