A new factory for Fractal/GCM

Luc Hogie

October 3, 2010

Contents

The aim of the document is to provide a documentation for the new 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 a 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 gather a large community of Researcher

1.1.2 ADL

The ADL used by Fractal allows the description of a hierarchy of components.

A basic tutorial for ADL syntax can be found at the following web URL:

`http://fractal.ow2.org/tutorials/adl/index.html`

This description is written in a dialect of XML. Even though the authors of Fractal claim the independance of Fractal from XML, no parser for other syntax is available.

Fractal incorrectly refers to ADL descriptions as ADL files. In fact, the ADL description is not stored in a file: it is stored as a Java resource. A Java resource can be either a plain file or an entry within a ZIP or JAR container.

1.1.3 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/`

1.1.4 GCM

1.1.5 The component factory

The component factory of Fractal is a piece of code within Fractal whose the role is to build a component out of its XML description. This description is called an ADL (Architecture Description Language).

GCM comes with its own factory which is an extension of Fractal's one.

Althought the objectives of the factory are to merely perform a number of verifications on the description and instantiate a set of components out of it, its implementation is cumbersome: it consists of nothing less than about 150 Java classes and configuration files. In order to implement is extensions, GCM adds about 40 classes and description files.

1.2 Description of the problem

The Fractal factory (and its derivative, in a worse manner) suffer from a number of problems:

- its design involves way more concepts than necessary;
- its design presents many imprecisions in the concepts it introduces;
- its design is made of numerous anti-intuitive constructs;
- its implementation code is full of tricks;
- its comes with absolutely zero documentation, be it at the level of the code or at the level of the conception.

The objective of the new factory is then to make a factory that is *much* shorter, cleaner, easier to understand, faster than the original Fractal/GCM factory. It particular, we intend to use as many concept as they are stricly necessary, in order to avoid confusion. 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.

2 Design of the factory

The new factory is object-oriented. Its design objective is to use an adequate tradeoff between *modularity* (which leads to large architectures) and *simplicity* (which often leads to a lack of modularity)

2.1 Definitions

2.2 Input format

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

2.2.1 ADL

The architecture of the component system must be written in a dialect of XML.

2.2.2 Argument values

The arguments values are described in a file as a set of $key = value$ pairs. The format of the argument file is then:

$$\begin{gathered} k_1 = v_1 \\ k_2 = v_2 \\ \ldots \\ k_n = v_n \end{gathered}$$

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

The name of the argument files is passed to the factory `newComponent()` method.

2.2.3 Deployment descriptor

2.3 The new factory

2.3.1 Lexical analysis: 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 lighweight tree structure whose the basic signature is:

```
class XMLNode
{
        String getName();
        Map<String, String> getAttributes();
        List<XMLNode> getChildrenNodes();
}
```

which is much simpler to understand and you use than the general purpose DOM data structure.

2.3.2 Semantic analysis: from DOM to semantic description

Each XML element type into the ADL description is represented by a `Description` class. Then at runtime, each XML element is represented at an instance of this class: a `Description` object. The description models which attributes and sub-elements (sub-descriptions) the element allows.

In pratise, Fractal ADL defines the following elements:

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`.

2.3.3 Component creation: 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.

3 Enhancement of the ADL

The ADL as it comes from the Fractal specifications 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 is no longer required

All the verifications are done in the semantic analyser.

This simplifies the writing of the XML file.

The DTD can still be useful for documentation purposes.

3.2 Storing the ADL in files rather than in Java resources

CGM is implemented 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 Java system themselves, which break the rule that Fractal is a specification intended to be implemented in any language.

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

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 probably comes from the fact, reinforced by the use of a DTD, that the *definition* element allows extra attributes that the *component* element does not. The *arguments* attribute might be of these.

3.4 the *arguments* attribute

Suppressing the *arguments* attribute into component description element.

3.5 the *content* XML element

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

```
<component name="boh">
        ...
        <content class="java.util.ArrayList" />
        ...
</component>
```

Instead, the content information for a component should be expressed as an attribute in the `component` description.

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

4 How to

4.1 Perform a new verification

Verifications are all performed in the `check()` method of the `Description` class. Depending on the description you want to perform the verification (component, interface, etc), you will have to look into the corresponding description class (respectively `ComponentDescription`, `IntefaceDescription`, etc).

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

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

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

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

4.2 Add a new attribute to an existing XML element

As described in Section **??**, each element type is represented by a `description` class. In turn, each attribute of the element is represented as a field into its corresponding `description` class. Adding a new attribute consists then in adding a new field in this class.

In order to make the new attribute need to be considered by the semantic analyser, you need to add a line of code into the corresponding method.

4.3 Add a new XML element

XML elements have a semantic corresponding description. To add a new XML element, you need to define such a description.

- 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 explicity given by the user.

4.3.1 Example: the interface element

Mandatory attributes must be provided at construction time:

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

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

```
if (n.getAttributes().get("contigency") != null)
{
        id.setContingency(n.getAttributes().get("contigency").equals("mandatory")
                ? Contingency.MANDATORY : Contingency.OPTIONAL);
}
```

4.4 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 supported

The following features are already supported by the new factory:

- component interfaces (singleton);

- component attributes;
- interface bindings;
- sub-components;
- ADL inheritance;
- arguments and variables (plus extra features);

6 Future works

- support for shared components;
- support for collection interfaces;
- support for bindings to web-services;
- support for collection components (Amine Rouini's topic);
- support for componentized membranes (Paul Naoumenko's topic);