Leveraging Component-Based Software Engineering with Fraclet

Romain Rouvoy, Philippe Merle

Leveraging Component-Based Software Engineering with Fraclet

Romain Rouvoy
University of Oslo, Department of Informatics
P.O.Box 1080 Blindern, 0316 Oslo, Norway
rouvoy@ifi.uio.no

Philippe Merle
INRIA-USTL-CNRS, ADAM Project-Team
Parc Scientifique de la Haute Borne, 40 avenue Halley,
Bât. A, Park Plaza, 59650 Villeneuve d'Ascq, France
philippe.merle@inria.fr

July 15, 2008

Abstract

Component-based software engineering has achieved wide acceptance in the domain of software engineering by improving productivity, reusability and composition. This success has also encouraged the emergence of a plethora of component models. Nevertheless, even if the abstract models of most of lightweight component models are quite similar, their programming models can still differ a lot. This drawback limits the reuse and composition of components implemented using different programming models.

The contribution of this article is to introduce Fraclet as a programming model common to several lightweight component models. This programming model is presented as an annotation framework, which allows the developer to annotate the program code with the elements of the abstract component model. Then, using a generative approach, the annotated program code is completed according to the programming model of the component model to be supported by the component runtime environment. This article shows that this annotation framework provides a significant simplification of the program code by removing all dependencies on the component model interfaces. These benefits are illustrated with the Fractal and OpenCOM component models.

1 Introduction

Component-Based Software Engineering (CBSE) has achieved wide acceptance in the domain of software engineering by improving productivity, reusability, and composition. This success has also encouraged the emergence of a plethora of component models. These component models can now be applied at any software layer, from operating systems (*e.g.*, Think [11]), to middleware (*e.g.*, OpenCOM [6], Fractal [5]), to applications (*e.g.*, Spring [15], EJB [8], CCM [21], SCA [22]). Usually, each of these component models defines their own *abstract* and *programming* models. The abstract model defines the general concepts supported by the component model (*e.g.*, component, port/interface, binding/-connection, composition/assembly). The programming model applies these concepts to a particular programming language, while introducing technical artefacts dedicated to the component model. This means, in particular, that this *technical code* is systematically tangled with the *business code* of the application developed. Furthermore, if the abstract models of existing component models are quite similar, their programming models can differ a lot. This drawback limits the reuse and composition of components implemented with different programming models (*e.g.*, reusing an OpenCOM-based component implementation in a Fractal-based application).

A convenient way to address these issues is to use *Attribute-Oriented Programming* (@op) techniques [23, 31, 10]. @op proposes to mark the program code with additional metadata to clearly separate the business logic from the domain-specific logic (typically, the technical properties). @op is gaining popularity with the recent introduction of annotations in *Java 2 Standard Edition* (J2SE) 5.0 [13] or in XDoclet [32], and attributes in C# [9]. Recently, the *Enterprise JavaBeans* (EJB) 3.0 specification extensively uses annotations to make EJB programming easier [8]. The *Service Component Architecture* (SCA) component implementation model provides also a particular set of annotations that can be placed in the code to mark specific elements of the implementation to be used by the SCA runtime environment [22]. However, these annotations remain tightly specific to each component model. Therefore, annotated EJB code can not be exploited by a SCA runtime environment and vice versa.

In this article, we present an extended version of the Fraclet framework previously introduced [25, 26]. In particular, we introduce an approach that leverages the development of component-based applications and we introduce a programming model common to several lightweight component models. This programming model is reified as an annotation framework, which allows the developer to mark the program code with the elements of the abstract component model. Then, using a generative programming approach, the annotated program code is completed by the programming model that is supported by the desired component runtime environment, such as Fractal or OpenCOM. We show that this annotation framework makes easier the programming of components by removing all dependencies on the component model interfaces. As a consequence, this approach protects the annotated program code from component model evolutions. Finally, the an-

notated code can be mapped to different lightweight component models, as illustrated in this paper with FRACTAL and OPENCOM.

In the remainder of the paper, we first resume the foundations of this work (cf. section 2) and then present our motivations, which are mostly related to the problem of technical and business code tangling (cf. section 3). Based on these descriptions, we present our abstract model and its reification as a set of annotations (cf. section 4). We demonstrate that components developed with this abstract model can be executed either in FRACTAL or OPENCOM runtime environments (cf. section 5), and we discuss the various alternatives for implementing this approach. We finally compare our approach with related work (cf. section 6) before concluding (cf. section 7).

2 Foundations

This section introduces both FRACTAL (cf. subsection 2.1) and OPENCOM (cf. subsection 2.2) component models before presenting the principles of *Attribute-Oriented Programming* (cf. subsection 2.3).

2.1 The Fractal Component Model

FRACTAL is the component model of the OW2 consortium for open-source middleware (see `http://fractal.ow2.org`). The FRACTAL component model uses the usual *component*, *interface*, and *binding* concepts [5]. A component is a runtime entity that conforms to the model. An interface is an interaction point expressing the provided and required by the component. A binding is a communication channel established between a client interface and a compatible server interface. Besides, FRACTAL supports *recursion with sharing* and *reflective control* as added values. The former means that a component can be composed of several sub-components at any level, and a component can be a sub-component of several components. The latter means that an architecture built with Fractal is reified at runtime, and can be dynamically introspected and managed. Finally, FRACTAL provides an *Architecture Description Language* (ADL) [19] to describe and deploy component-based configurations automatically [16].

Figure 1 illustrates the different entities in a typical FRACTAL component architecture. Thick black boxes denote the controller part of a component, while the interior of these boxes correspond to the content part of a component. Arrows correspond to *bindings*, and tau-like structures protruding from black boxes are internal or external interfaces. Internal interfaces are only accessible from the content part of a component. External interfaces appearing at the top of a component represent reflective control interfaces such as the *Lifecycle Controller* (lc), the *Binding Controller* (bc), the *Attribute Controller* (ac) or the *Content Controller* (cc) interfaces. The two dashed boxes (C) represent a shared component.

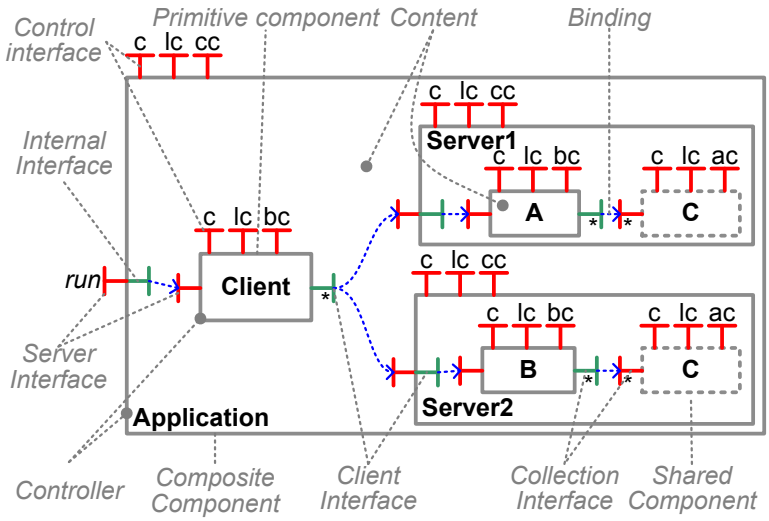

Figure 1: The architecture of a FRACTAL component.

2.2 The OpenCOM Component Model

OPENCOM is the component model developed by the Lancaster University in the context of the OPENORB project. OPENCOM is a lightweight, efficient, reflective component model that uses the core features of Microsoft COM to underpin its implementation; these features include the binary level interoperability standard, Microsoft's IDL, COM's globally unique identifiers and the IUnknown interface [6]. Recently, a Java version of OPENCOM has been developed to provide platform independence, and to ease the developments of applications on top of OPENCOM.

The key concepts of OPENCOM are *interface*, *receptacle* and *connection*. Each component implements a set of interfaces and receptacles, as shown in Figure 2. An interface expresses a unit of service provision, a receptacle describes a unit of service requirement and a connection is the binding between a receptacle and an interface of the same type. Among the possible interfaces provided by an OPENCOM component, the IUknown interface provides the reference of the component and an operation to navigate through the component's interfaces, the IMetaInterface interface provides operations to introspect the component, the IConnections interface is used to connect the component receptacles to interfaces, and the ILifeCycle interface supports the lifecycle of the component. OPENCOM provides a standard runtime substrate per address space that manages the creation and deletion of components, acts upon requests to connect/disconnect components and provides service interfaces for reflective operations. The runtime substrate dynamically maintains a graph of the components currently in use. The maintenance of dynamic dependencies between components is relevant for the introspection and reconfiguration of component configurations. The reflective interfaces of OpenCOM follow three

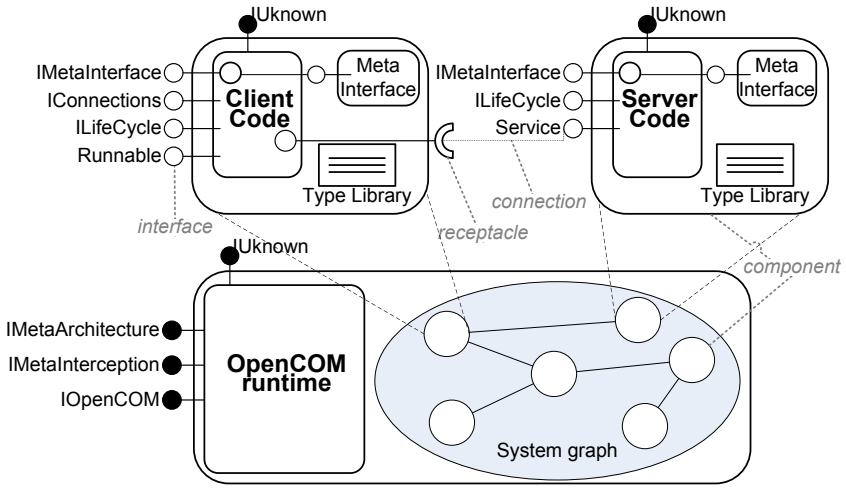

Figure 2: The architecture of an OpenCOM assembly.

of the meta-models proposed by OpenORB—*i.e.*, the Interface meta-model (IMetaInterface), the Architecture meta-model (IMetaArchitecture) and the Behaviour meta-model (IMetaInterception) [6].

2.3 The Principles of Attribute-oriented Programming

Attribute-Oriented Programming (@OP) is a program-level marking technique. Basically, this approach allows developers to mark program elements (*e.g.*, classes, methods, and fields) with *annotations* to indicate that they maintain application-specific or domain-specific concerns [10, 23, 31]. For example, a developer may define a logging annotation and associate it with a method to indicate that the calls to this method need to be logged, or may define a web service annotation and associate it with a class to indicate that the class must implement a Web Service. Annotations separate application's business logic from middleware-specific or domain-specific concerns (*e.g.*, logging and web service functions). By hiding the implementation details of those semantics from program code, annotations increase the level of programming abstraction and reduce programming complexity, resulting in simpler and more readable programs. The program elements associated with annotations are transformed to more detailed program code by a supporting generation engine. For example, a generation engine may insert logging code into the methods associated with a `logging` annotation. Dependencies on the underlying middleware are thus replaced by annotations, acting as weak references—*i.e.*, references that are not mandatory for the application. This means that the evolution of the underlying middleware is taken into account by the generation engine and let the program code unchanged.

@OP can also be used to provide *continuous integration* in CBSE. Continuous integra-

tion allows a developer to generate the middleware artifacts at any step of the component development. Developers concentrate their editing work on only one source file per component. The deployment metadata are continuously integrated without worrying about updating them. When the development of a component consists of several files, @OP allows the developer to maintain only one of them while the other files are generated automatically. Besides, working with only one file per component gives a better overview of the program code to the developer. Therefore, the developer can concentrate on the business logic and reduce the development time drastically.

@OP has been applied in several object-oriented frameworks to ease the process of configuring applications (*e.g.*, Hibernate, Struts, Castor [32]), and it has been applied by several JEE application servers to simplify the configuration of *Enterprise JavaBeans* (EJB) components (*e.g.*, JOnAS, WebSphere, JBoss). Nevertheless, these annotations target only the configuration of the EJB components, and are specific to each application server.

Recently, the EJB 3.0 specification has introduced extensive annotations to make EJB programming easier [8]. The annotations defined in this specification address either EJB component configuration or program code generation concerns. Nevertheless, this specification presents two weaknesses. Firstly, it is dedicated to EJB components. This means that the strengths of this specification are not directly applicable to other component models. Secondly, the EJB specification does not focus on the EJB abstract model but abstracts the EJB programming model. Thus, the annotations defined in the EJB specification are tightly coupled to the EJB programming model.

The @OP approach provides an useful formalism to introduce a higher-level semantics into the artifacts of existing programming models. In particular, @OP can be applied to represent the abstract component model using annotations. These annotations remove all the technical code that is required by a given programming model and that is tangled with the business code. As a side effect, the use of @OP allows the developer to write a program code compliant with several component models. To achieve this, it is necessary to identify the core concepts that are usually defined in the component models. The specificities of existing component models are reified in some extensions of the annotation framework.

3 Motivations

In this section, we motivate the actual limitations of CBSE in terms of dependency to the component model (cf. subsection 3.1), business and technical code tangling (cf. subsection 3.2), and metadata redundancy (cf. subsection 3.3).

3.1 Motivating Component-based Software Engineering

To motivate the limitation of current trends in the development of CBSE, we use the example of the *Comanche* web server. This web server is developed using the FRACTAL component model, and provides the basic features of a web server. *Comanche* is composed of seven primitive components, which are grouped into four composite components (cf. Figure 3). Incoming HTTP requests are received by the **Receiver** and their analysis is systematically scheduled by the **Scheduler**. This analysis is first handled by the **Analyzer**, which logs the request using the **Logger** and before dispatching its interpretation using the **Dispatcher**. The **Dispatcher** delegates successively the HTTP request interpretation to the **File Handler** and then, if it fails, to the **Error Handler** to produce the resulting web page.

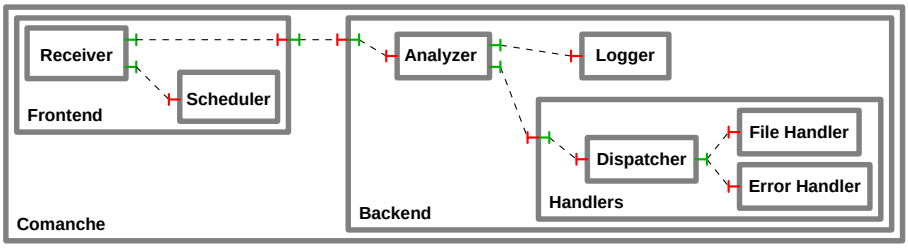

Figure 3: The architecture of the *Comanche* web server.

Listing 1 presents the implementation of the component `Analyzer`, and its associated data structure `Request` and the interface `RequestHandler` using the Java programming model defined by the FRACTAL component model. The data structure `Request` encapsulates a request processed by *Comanche*. The interface `RequestHandler` defines a business method `handleRequest` used to process an incoming request. The interface `AnalyzerAttributes` defines the setter and getter methods required by FRACTAL to handle the component attribute `filter`. Both interfaces `RequestHandler` and `AnalyzerAttributes` are then implemented by the class `Analyzer` to specify the behaviour of the methods. This class also implements the interface `BindingController` to support the connection of the client interfaces `Logger` and `RequestHandler` named `l` and `rh`, respectively.

Listing 2 presents the description of the component using FRACTAL ADL. The definition `comanche.Analyzer` extends the definition `comanche.AnalyzerType`. The former specifies the concrete implementation of the component, while the latter focuses on the definition of the component type—*i.e.*, the provided and required interfaces.

Finally, Listing 3 illustrates the definition of a composite component `comanche.Backend` that composes the definitions of the primitive component `Analyzer` and the components `comanche.Handlers` and `comanche.Logger`.

```java
public class Request {
    public Socket s;
    public Reader in;
    public PrintStream out;
    public String url;
}

public interface RequestHandler {
  void handleRequest (Request r) throws java.io.IOException;
}

public interface AnalyzerAttributes extends AttributeController {
  void setFilter(String value);
  String getFilter();
}

public class Analyzer implements BindingController, AnalyzerAttributes, RequestHandler {
  private RequestHandler rh;
  private Logger l;
  private String filter;
  // BindingController interface implementation
  public String[] listFc () { return new String[] { "l", "rh" }; }
  public Object lookupFc (String itfName) {
    if (itfName.equals("l")) { return this.l; }
    else if (itfName.equals("rh")) { return this.rh; }
    else return null;
  }
  public void bindFc (String itfName, Object itfValue) {
    if (itfName.equals("l")) { this.l = (Logger)itfValue; }
    else if (itfName.equals("rh")) { this.rh = (RequestHandler)itfValue; }
  }
  public void unbindFc (String itfName) {
    if (itfName.equals("l")) { this.l = null; }
    else if (itfName.equals("rh")) { this.rh = null; }
  }
  // AnalyzerAttributes interface implementation
  public void setFilter(String value) { this.filter = value; }
  public String getFilter() { return this.filter; }
  // RequestHandler interface implementation
  public void handleRequest (Request r) throws IOException {
    r.in = new InputStreamReader(r.s.getInputStream());
    r.out = new PrintStream(r.s.getOutputStream());
    String rq = new LineNumberReader(r.in).readLine();
    this.l.log(rq);
    if (rq.startsWith(this.filter)) {
      r.url = rq.substring(this.filter.length+1, rq.indexOf(' ', this.filter.length));
      this.rh.handleRequest(r);
    }
    r.out.close();
    r.s.close();
}  }
```

Listing 1: The Java code of the component **Analyzer**.

```
<definition name="comanche.AnalyzerType">
  <interface name="a"  signature="comanche.RequestHandler" role="server"/>
  <interface name="l"  signature="comanche.Logger"         role="client"/>
  <interface name="rh" signature="comanche.RequestHandler" role="client"/>
</definition>

<definition name="comanche.Analyzer" extends="comanche.AnalyzerType"
       arguments="filter=GET ">
  <content class="comanche.RequestAnalyzer"/>
  <attributes signature="comanche.AnalyzerAttributes">
    <attribute name="filter" value="${filter}"/>
  </attributes>
</definition>
```

Listing 2: The FRACTAL ADL description of the component **Analyzer**.

```
<definition name="comanche.BackendType">
  <interface name="a" signature="comanche.RequestHandler" role="server"/>
</definition>

<definition name="comanche.Backend" extends="comanche.BackendType">
  <component name="ra" definition="comanche.Analyzer"/>
  <component name="rh" definition="comanche.Handlers"/>
  <component name="l"  definition="comanche.Logger"/>
  <binding client="this.a" server="ra.a"/>
  <binding client="ra.l"   server="l.l"/>
  <binding client="ra.rh"  server="rh.rh"/>
</definition>
```

Listing 3: The FRACTAL ADL description of the component **Backend**.

The FRACTAL ADL definition(s) and the associated component implementation represent the elementary information required to implement a component using the FRACTAL component model. However, the code used to develop this component remains tightly coupled to the specificities and constraints of the FRACTAL component model and cannot be easily executed with the runtime associated to the OPENCOM component model.

3.2 Business/Technical Code Tangling

Although CBSE provides more modularity, configurability, and reusability to applications, the use of a given component model introduces also more complexity, verbosity and redundancy in the information expressed by the developer compared to object-oriented programming practices. However, this complexity mostly derives from the underlying programming model used to develop an application. In particular, this programming model maps the abstract model concepts to the programming language artefacts used to develop components. This means that the abstract model concepts are seamlessly drowned in the program code. By introducing dependencies on the component model, the program code is no longer only concerned with business properties, but also with technical properties. The drawbacks that arise from such a tangled component implementation are mainly located in the technical part of the program code, because developing an application using components requires taking into account concerns that are not always related to the business ones. For example, the technical methods required to support bindings or attributes have to respect a behaviour fixed by the component model specification (cf. lines 21–38 in Listing 1). These methods should coherently reflect the structure of the component, but are often error prone due to the repetitive form of the code. This means that any evolution of the structure of the component should be carefully reported either in the body of each technical method (*e.g.*, adding a new client interface requires to update the `BindingController` method bodies) or by adding/removing methods (*e.g.*, removing an attribute requires to remove two methods both in the associated `AttributeController` interface and in the component implementation). Similarly, a slight evolution of the programming model (*e.g.*, re-factoring the signature of the method `bindFc`) will systematically impact all the components developed.

When observing the code of the components, one can notice that these methods are always referring to fields used to store the related information (reference of a binding, value of an attribute), and the declaration of these fields appears to be common to most of the component models that can be used.

3.3 Metadata Redundancy

Architecture Description Languages (ADL), such as FRACTAL ADL, provide a syntax that facilitates the composition and the configuration of component-based architectures. However, the independence between the ADL and the implementation of the described

architecture implies that some metadata is duplicated in both parts. In particular, each component implemented is associated to a description of its structure (cf. Listing 2), but most of the metadata described in the ADL is redundant with some pieces of code of the component. For example, the name and the type of the client interfaces are described in both artefacts (cf. lines 21–38 in Listing 1 and lines 3–4 in Listing 2). This means that any structural evolution of the component should be also reflected in the associated ADL definition (*e.g.*, adding a new client interface in the implementation requires to add a new tag `<interface...>` in the description). The same comment applies to the definition of the component attributes (names and default values), which are also described in the ADL. Thus, keeping coherency between information described in the component implementation and its ADL definition is often prone to errors. Nevertheless, the ADL definition of component compositions exhibits some configuration information that can not be described in the component implementation. In particular, the definition of component compositions, bindings, and attribute values are only present in the ADL definitions.

One can thus observe that most of the information related to the description of primitive components can be easily inferred from the information that are already expressed in the associated component implementation, while the primitive component descriptions are generally shared by the composite component descriptions.

Challenges. To reduce the number of errors related to the technical code development and to improve the coherency of metadata, we propose a lightweight programming model called Fraclet. The objectives of Fraclet is thus to reduce the number of line of code the developer has to write in order to *i)* remove the dependency to a particular component model, *ii)* enforce the evolution support for components, and *iii)* centralised the component metadata.

4 Fraclet: a Component Programming Model

This section introduces the Fraclet component programming model (cf. subsection 4.1) and its mapping to a set of Java5 annotations (cf. subsection 4.2). An illustration of the use of Fraclet is then provided by revisiting the *Comanche* web server (cf. subsection 4.3).

4.1 The Fraclet core model

Most existing component models (*e.g.*, OpenCOM [4], Fractal [5], JavaBean [14], EJB [8], CCM [21]) rely on some common core concepts, as summarized in Figure 4. A **Component** is defined as an entity that **Provides** and **Requires** some interfaces. These interfaces are usually identified by a *name* and a *signature*—*i.e.*, a set of operations. A *cardinality* is also specified to define that a **Component** requires several interfaces of the same type.

A **Component** can additionally define a set of **Attributes** to support configuration. An **Attribute** is initialized with a *value* when the **Component** is loaded. A **Component** can furthermore require some **Services** provided by the runtime structure executing it. These **Services** often provide to the technical properties required by a component (*e.g.*, logging). Finally, the component models define the concept of **Lifecycle**. This concept allows the component to be aware of its current state (*e.g.*, created, started, stopped, destroyed).

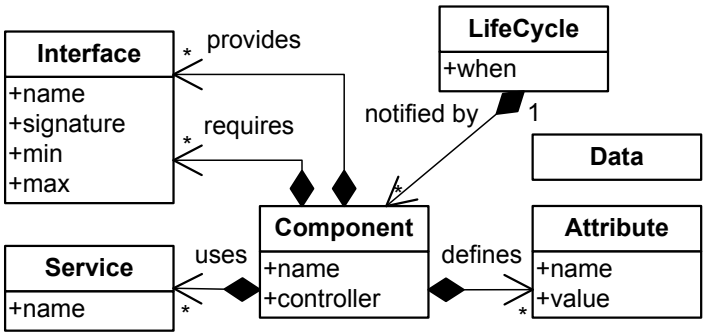

Figure 4: The Fraclet core model.

4.2 A mapping of Fraclet to annotations

Each core component concept previously identified is defined as an annotation applicable to a piece of the program code. The resulting annotations are summarized in Table 1. The annotation `@Data` applies on a class to describe a data structure exchanged between components. The annotation `@Component` refers to a class to specify the *name* of the ADL description associated to a component and possibly a reference to its hosting *controller*[1]. The annotation `@Provides` applies to an interface that is provided by a component. An attribute *name* can be specified when using this annotation. The attribute *signature* is used when the interface signature cannot be inferred from the program code. The annotation `@Requires` applies on the reference of an interface required by a component. The attribute *name* (resp. *signature*) is defined to override the name (resp. the signature) of the field marked by the annotation. The attributes *min* and *max* are useful to indicate whether the field refers to an optional reference or to a collection of references. The annotation `@Attribute` applies to the declaration of a field. If no attribute *value* is defined, the attribute value is required when composing the components together. The annotation `@Service` applies to the reference to a service provided by the component runtime environment (*e.g.*, logging). The attribute *name* refers to the identifier of the service. The annotation `@Lifecyle` applies to a method defined in the component code. This method

[1]The controller, also known as container, is the hosting infrastructure of the component.

defines treatments that should be executed at a given transition of the component life cycle (*e.g.*, from stopped to started) using the attribute *when*.

Annotation	Code Element	Parameter	Description	Contingency	Default Value
@Data	`Class` or `Interface`		data structure exchanged between components		
@Component	`Class`	*name*	component definition name	optional	full class name
		controller	container reference	optional	-
@Provides	`Class` or `Interface`	*name*	provided interface name	optional	interface name
		signature	provided interface signature	optional	interface signature
@Requires	`Field`	*name*	required interface name	optional	field name
		signature	interface signature	optional	field signature
		min	interface minimal cardinality	optional	`1` (mandatory)
		max	interface maximal cardinality	optional	`1` (singleton)
@Attribute	`Field`	*name*	component attribute name	optional	field name
		value	attribute default value	optional	-
@Service	`Field`	*name*	component service name	required	-
@Lifecycle	`Method`	*when*	lifecycle state to handle	required	-

Table 1: The Fraclet annotation-based programming model.

4.3 Revisiting the Motivating Example

This section revisits the *Comanche* web server introduced in subsection 3.1. To illustrate the benefit of @OP, the program code of this application is reengineered to replace all the technical code by some of the previously defined annotations. Listings 4 and 5 show that the original business code is preserved while using the Fraclet programming model. The data structure `Request` is marked with the annotation @Data. The interface `RequestHandler` is marked with the annotation @Provides to define `a` as its default identifier (line 8 of Listing 4). This information was previously defined in the ADL descriptor of the component. The references to the interface `RequestHandler` and `Logger` in the class `Analyzer` are marked with the annotation @Requires (lines 13–14 of Listing 4) to specify the dependency of the component towards required interfaces. Finally, the attribute `filter` is annotated with @Attribute to declare the field as a component attribute and defined its default value to `"GET "` (line 15 of Listing 4). As a consequence, the technical program code related to interface references binding and attributes handling becomes useless. Moreover, the dependency to the FRACTAL component model has been leveraged.

Listing 5 presents a simplified description of the component assembly **Backend**. In particular, this description extends the definition `comanche.RequestHandler`, which is an abstract FRACTAL ADL definition associated to the interface `comanche.RequestHandler`. Thus, the component `comanche.Backend` is defined as a composite component that provides the interface `comanche.RequestHandler`, and contains the components `comanche.Analyzer`, `comanche.Handlers`, and `comanche.Logger`.

4.4 Evaluation

```
1 @Data public class Request {
2     public Socket s;
3     public Reader in;
4     public PrintStream out;
5     public String url;
6 }

8 @Provides(name="a") public interface RequestHandler {
9   void handleRequest (Request r) throws java.io.IOException;
10 }

12 public class Analyzer implements RequestHandler {
13   @Requires private RequestHandler rh;
14   @Requires private Logger l;
15   @Attribute(value="GET ") private String filter;

17   public void handleRequest (Request r) throws IOException {
18     r.in = new InputStreamReader(r.s.getInputStream());
19     r.out = new PrintStream(r.s.getOutputStream());
20     String rq = new LineNumberReader(r.in).readLine();
21     this.l.log(rq);
22     if (rq.startsWith(this.filter)) {
23       r.url = rq.substring(this.filter.length+1, rq.indexOf(' ', this.filter.length));
24       this.rh.handleRequest(r);
25     }
26     r.out.close();
27     r.s.close();
28 }  }
```

Listing 4: The Java code of the component **Analyzer**.

```
1 <definition name="comanche.Backend" extends="comanche.RequestHandler">
2   <component name="ra" definition="comanche.Analyzer"/>
3   <component name="rh" definition="comanche.Handlers"/>
4   <component name="l"  definition="comanche.Logger"/>
5   <binding client="this.a" server="ra.a"/>
6   <binding client="ra.l"   server="l.l"/>
7   <binding client="ra.rh"  server="rh.rh"/>
8 </definition>
```

Listing 5: The FRACTAL ADL description of the component **Backend**.

Table 2 compares the source code written by a developer when developing the *Comanche* web server using only the FRACTAL API (A) or using the Fraclet programming model (B). In particular, this evaluation shows that only half of the source code (56 %) is related to the business concerns of *Comanche*. Thanks to Fraclet, the business concerns are isolated from the technical artifacts imposed by the use of a particular component model. These artifacts are automatically generated by the Fraclet generators, thus ensuring a continuous integration of the component metadata. As a consequence, the minor evolutions of the component model do not impact the components anymore, but only the Fraclet generators.

Metric description	**Metric unit**	FRACTAL A	Fraclet B	**Benefit** G=A-B	**Rate** G/A
Java	Files	13	12	1	8 %
ADL	Files	19	6	13	68 %
Java	Lines	263	189	74	28 %
ADL	Lines	137	47	90	66 %
Java + ADL	Bytes	14 K	8 K	6 K	44 %

Table 2: The empirical measures observed on *Comanche*.

Fraclet is already used by several projects built on top of the FRACTAL component model. The GOTM project [27]—a framework for building transaction services—and the FRACTAL DEPLOYMENT FRAMEWORK [12]—a generic tool for deploying distributed systems and applications— are two examples of projects using the XDoc-based implementation of Fraclet. While the COSMOS project [24]—a framework for processing context information—and PEtALS [18]—an implementation of the *Java Business Integration* (JBI) specification—are using the Java5-based implementation of Fraclet. Besides, other projects have extended Fraclet to include new features, such as Dream [17]—a framework for building message-oriented middleware—and PROACTIVE [2]—a middleware for parallel, distributed and multi-threaded computing. Experiences reported by all these projects show that Fraclet improves the quality of the developments by:

1. *reducing programming errors* related to component developments (*e.g.*, implementation of the FRACTAL lifecycle controller),
2. *reducing the time* spent to develop components,
3. *increasing the robustness* to evolutions of the component model (*e.g.*, changes in the programming API).

In particular, the application source code becomes less prone to errors, and gains in terms of visibility by removing the technical methods that were not related to the business concerns. By removing the technical methods from the source code, we also

remove the code dependency towards a particular component model, and the resulting source code becomes independent of the component model used at runtime. Thus, thanks to Fraclet, the developer spends less time to develop the components because he/she does not need to deal with the specificities of the FRACTAL programming model anymore. As a consequence, Fraclet contributes to reduce the learning curve for adopting CBSE. Developers can apply CBSE principles by extending the general programming language they are used to (*e.g.*, the Java programming language) with domain-specific annotations. Nevertheless, the use of these annotations still implies the manipulation of CBSE concepts, and in some cases the use of reflective capabilities may bind a component implementation to a particular model.

5 Implementation Issues

The Fraclet programming model can be implemented using various technologies. As depicted in Figure 5, the generative approach adopted by Fraclet consists in parsing the developed source code to produce both Java source code and XML descriptions.

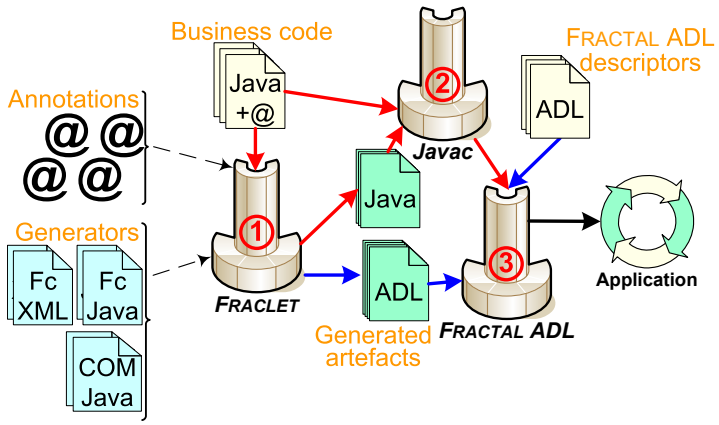

Figure 5: The Fraclet generative approach.

This section first introduces an implementation of the Fraclet framework using source code transformation (cf. subsection 5.1), and then discuss alternative implementations (cf. subsection 5.2).

5.1 Using Source Code Transformation

This section presents a possible implementation of the mapping of Fraclet annotations. This implementation uses a source code transformation engine named Spoon (cf. subsubsection 5.1.1) and allows the developer to produce and deploy either FRACTAL

(cf. subsubsection 5.1.2) or OpenCOM (cf. subsubsection 5.1.3) components developed with Fraclet.

5.1.1 Spoon Transformation Framework

Spoon is a Java 5 open compiler built on top of Eclipse *Java Development Tools* (JDT) Core [23]. Spoon provides the user with a representation of the Java *Abstract Syntax Tree* (AST) in a metamodel, which allows both for reading and writing. Using this metamodel and a specific API, Spoon allows the programmer to process Java 5 programs. This processing is implemented with a visitor pattern that scans each visited program element and can apply some user-defined processing jobs called *processors*.

Taking advantage of Java 5 features, Spoon also natively provides a framework that allows for the definition of code templates in pure Java. By specifying templates in pure Java, programmers can write them in their favorite Java IDE and benefit from all the advantages that come with it (incremental compilation, completion, syntax highlighting, contextual help, re-factoring, wizards, etc.).

5.1.2 Producing the Fractal Artefacts

The generation of the Fractal artefacts relies on the definition of one Spoon template per annotation to support the Java programming model. These templates introduce the references to the technical interfaces specified by the Fractal component model, and implement the associated methods directly in the code of the component. In the particular case of component defining attributes, an attribute controller interface is dynamically generated with Spoon to support the management of the component attributes.

Moreover, another set of Spoon processors is defined to generate the Fractal ADL definitions for primitive components. These processors and templates are then applied by Spoon when parsing the component source code to both modify the component source code and generate its associated Fractal ADL definitions, as illustrated in Figure 5. As a consequence, the resulting Java code and XML descriptions are similar to the examples presented in Listings 1 and 2.

5.1.3 Producing the OpenCOM Artefacts

Similarly to what has been done for the Fractal component model, Spoon templates have been defined for handling each annotation of the Fraclet programming model and generating the associated OpenCOM technical source code. The resulting component is depicted in Listing 6, which introduces the technical code required by the Java programming model of the OpenCOM component model. In particular, the OpenCOM programming model uses an abstract class `OpenCOMComponent` that includes generic source code for OpenCOM components. This implies that the class `Analyzer` has to define a particular constructor accepting a `binder` as parameter and should delegate the initialisation

of the component to the parent class. For each interface required by the component, the annotated field is replaced by a structure `OCM_SingleReceptacle` specific to OpenCOM for handling references. Similarly, the component attributes are stored in a dedicated structure managed by OpenCOM and are accessible via the method `GetAttributeValue`. Thus, accesses to either component dependencies or attributes are also modified in the business source code to reflect the constraints of the OpenCOM programming model. Finally, technical methods are introduced in the source code to manage the dependencies of component (interface `IConnections`) and its life cycle (interface `ILifeCycle`).

Concerning the description of the architecture, we chose to reuse the Fractal ADL processors to associate a Fractal ADL definition to each OpenCOM component. Thus, using the Fractal ADL factory and a specific *backend* supporting the OpenCOM component model [16], OpenCOM components can be easily deployed using the same definition of the composite component `comanche.Backend`. This means that the composition of the OpenCOM components can be handled by the Fractal ADL factory via a dedicated *backend* [16]. Nevertheless, one can also consider the development of Spoon processors that generate ADL descriptions compliant to the Plastik language supported by OpenCOM [3]. In this case, the composition of the Plastik components can be handled by the Fractal ADL factory via a dedicated *frontend* [16]. Furthermore, one can observe that a component developed with the Fraclet programming model can be executed in either a Fractal or OpenCOM runtime environment.

5.2 Discussing Alternative Implementations

In this section, we present and discuss various alternative implementations of the Fraclet programming model. In particular, we discuss the use of Java5 or XDoc annotations (cf. subsubsection 5.2.1), the difference between source code generation and transformation (cf. subsubsection 5.2.2), and finally the manipulation of either Java source code or byte-code (cf. subsubsection 5.2.3).

5.2.1 Java5 Versus XDoc Annotations

The annotations defined by Fraclet can be implemented using either the facility provided by Java5 or using XDoc annotation framework[2]. XDoc annotations are integrated in the Javadoc comments of the source code elements. This means that these annotations comply with any version of the Java specification and the annotated source code can be compiled and executed on small devices using the J2ME environment. However, these annotations are lost at runtime, which means that they have to be interpreted directly in the source code. Besides, the Java5 annotations provide a better typing system to ensure the correctness of the annotation parameters given that the Java5 annotations are integrated in the language.

[2]XDoc annotation framework: `http://xdoclet.codehaus.org/`

```
1public class Analyzer extends OpenCOMComponent
2    implements RequestHandler, IConnections, ILifeCycle {
3  private OCM_SingleReceptacle<RequestHandler> rh ;
4  private OCM_SingleReceptacle<Logger> l ;

6  public Analyzer(IUnknown binder) {
7    super(binder);
8    this.rh = new OCM_SingleReceptacle<RequestHandler>(RequestHandler.class);
9    this.l = new OCM_SingleReceptacle<Logger>(Logger.class);
10 }
11 // IConnections Interface
12 public boolean connect(IUnknown itf, String signature, long id) {
13   if (signature.equalsIgnoreCase("comanche.RequestHandler")) {
14     return this.rh.connectToRecp(itf, signature, id);
15   } else if (signature.equalsIgnoreCase("comanche.Logger")) {
16     return this.l.connectToRecp(itf, signature, id);
17   }
18   return super.connect(itf, signature, id);
19 }
20 public boolean disconnect(String signature, long id) {
21   if (signature.equalsIgnoreCase("comanche.RequestHandler")) {
22     return this.rh.disconnectFromRecp(id);
23   } else if (signature.equalsIgnoreCase("comanche.Logger")) {
24     return this.l.disconnectFromRecp(id);
25   }
26   return super.disconnect(signature, id);
27 }
28 // ILifeCycle Interface
29 public boolean shutdown() { return true; }
30 public boolean startup(Object pIOCM) { return true; }
31 // Business code
32 public void handleRequest (Request r) throws IOException {
33   r.in = new InputStreamReader(r.s.getInputStream());
34   r.out = new PrintStream(r.s.getOutputStream());
35   String rq = new LineNumberReader(r.in).readLine();
36   this.l.m_pIntf.log(rq);
37   String filter = (String)GetAttributeValue("comanche.RequestHandler", "Interface",
38                                             "filter").value;
39   if (rq.startsWith(filter)) {
40     r.url = rq.substring(filter.length+1, rq.indexOf(' ', filter.length));
41     this.rh.m_pIntf.handleRequest(r);
42   }
43   r.out.close();
44   r.s.close();
45} }
```

Listing 6: The Java code of the OpenCOM component **Analyzer**.

Fraclet already provides two implementations that support either XDoc or Java5 annotations.

5.2.2 Transformation Versus Generation Processing

The technical code associated to a particular component model can be woven with the business code in different ways. The first option consists in generating a class that extends the business code of the component class and includes all the technical code dedicated to a particular component model. An alternative to generation consists in modifying the source code written by the developer to introduce the technical code. This approach reduces the memory footprint of component (by reducing the number of classes associated to a component) and supports the validation of the business source code with regards to the component model specification (*e.g.*, hidden communication path detection). However, the uncoupling supported by the generative approach provides the capability to select the component model to use at load-time.

The two implementations of Fraclet already support processing of annotations by either transformation (for Java5 annotations) or generation (for XDoc annotations).

5.2.3 Compile-time Versus Load-time Parsing

The completion of the business code with the technical code can be done either statically at compile-time or more dynamically at load-time. The introduction of the technical artifacts at compile-time generates the source code of the component and its Fractal ADL description, while the introduction at load-time generates bytecode and the component factory internal representation—*i.e.*, an AST for Fractal ADL.

The current Fraclet implementations only support compile-time generation. Nevertheless, a load-time version of Fraclet can be developed. The generation of the bytecode can be achieved using a bytecode transformation framework, such as ASM[3], that visits the annotations included in the business code and generates the technical methods. In Fractal ADL, a load-time integration consists in extending the ADL factory with an Annotation Loader component.

6 Related work

This section compares our work with existing approaches such as Aspect-Oriented Programming and Model-Driven Engineering. We also compare to the existing technologies that use Attribute-Oriented Programming to leverage the management of technical properties.

[3]The ASM framework: `http://asm.ow2.org`

Attribute-Oriented Programming has already been applied in the context of CBSE. The *Enterprise JavaBeans* (EJB) 3.0 [8] and the *Service Component Architecture* (SCA) [22] specifications extensively use annotations to make programming easier but these approaches provide no complete abstraction of their programming model. [28] presents an *a posteriori* approach that extends an ADL to mark components with annotations. Nevertheless, this work is limited to the introduction of additional technical properties, such as the property of *Deny of Service* detection, to legacy components. Our work is an *a priori* approach to leverage CBSE using an annotation-based abstraction of the programming model of the component model.

Aspect-Oriented Programming (AOP) provides a partial solution to the problem of technical code abstraction. In [28], the annotations defined at the architectural level are consumed in the program code by aspects defined with AspectJ. The annotations mark potiential victim interfaces and are consumed to inject the *Deny of Service* detection code. Similarly, the AOKell implementation of the Fractal component model provides an aspect that automatically injects the technical code related to the handling of the component client interfaces [29]. However, AOP is not able to generate additional artifacts such as the attribute controller interface or the component definitions. Our approach provides Java and XML generators to support both program code and ADL definition generation.

Model-Driven Engineering (MDE) promotes the use of Platform-Independent Models (PIM) to define the business concern of an application [30]. The PIM can be transformed into different Platform-Specific Models (PSM) that take into account the specificities of a given platform (*e.g.*, a given component model). Our approach follows the same idea at the program code level rather than at the model level. Indeed, the annotated program code can be considered as a Platform-Independent Code (PIC) composed of the business concern of the application and the annotations related to CBSE. Then, the generators produce the program code compliant with a given programming model as a Platform-Specific Code (PSC). Our approach is a practical application of MDE for the programming level, as a consequence it appears as an interesting solution to provide component model independency. In [31], the authors combined UML stereotypes and tagged values to simulate an annotation mechanism when modeling an application. The stereotype and the tagged values are thus mapped to annotations when generating the application code. This interesting approach brings annotations to the model level but it does not try to abstract the diversity of underlying platforms as proposed in this paper.

7 Conclusion

This article has presented Fraclet, a lightweight component model that leverages CBSE. This model gathers the core concepts of the component programming models manipulated by the developers. Reified as a set of annotations, the developer can write a program code that contains only the business concerns of the application, making it more legible. The

compliance with a given component programming model is ensured by generators that consume the annotations to produce the technical code required by a component model. The generative approach provides an interesting solution to the problem of the evolution of component models because a modification of the programming model of the component model impacts only the generators and no more all the applications. Finally, an annotated program code can be executed on various component-oriented platforms. In this article, the generators for the Fractal and OpenCOM component models are illustrated and demonstrate the benefits of our approach.

Current middleware evaluations show that about half of the source code is related to the business concerns (measures observed on GoTM [27], Fractal Deployment Framework [12], and COSMOS [24] projects). This means that, thanks to the use of Fraclet, the application source code becomes less prone to errors, and gains in terms of visibility by removing the technical methods that were not related to the business concerns. By removing the technical methods from the source code, we also remove the code dependency towards a particular component model, and the resulting source code becomes independent of the component model used at runtime. However, even if Fraclet does not forbid it, the integration of concerns, such as reflection, persistency, or transactions, may restrict the compatible target components, and ultimately introduce a dependency to a particular component model.

Among the possible evolutions of this approach, the support of additional component models is considered (*e.g.*, Spring [15], JavaBean [14], EJB [8], CCM [21], SCA [22]). We believe that the definition of a generic support for programming components requires the definition of a modular meta-model similarly to the xADL [7] approach, which provides an extensible architecture description language based on XML. Another possible extension consists in supporting crosscutting concerns in the programming model (*e.g.*, transaction, persistency). However, the support of additional concepts in the Fraclet core model would reduce the number of target component models supporting all the features, while increasing potential conflicts between annotations. Thus, we are interested in enforcing the validation of the resulting source code to ensure its reliability with regards to the component model used at runtime [20]. This validation can be also extended to the verification of the business code to detect hidden communication paths by analyzing method parameters as done in ArchJava [1].

Availability. Two implementations of Fraclet are already available under the GNU LGPL license and can be downloaded from `http://fractal.ow2.org/fraclet`.

Acknowledgements

The authors wish to thank Denis Conan, Carlos Noguera, Nicolas Pessemier, and Renaud Pawlak for their contributions to the Fraclet programming model and its implementations.

The authors also thank the reviewers for their valuable comments.

References

[1] Aldrich J., Chambers C., Notkin D. ArchJava: Connecting Software Architecture to Implementation. *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*. ACM, Orlando, Florida, USA, p. 187–197, Mai 2002.

[2] Baduel L., Baude F., Caromel D., Contes A., Huet F., Morel M., Quilici R. *Grid Computing: Software Environments and Tools*, Chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[3] Batista T., Joolia A., Coulson G. Managing Dynamic Reconfiguration in Component-Based Systems. *Proceedings of the 2nd European Workshop on Software Architecture (EWSA'05)*, *Lecture Notes in Computer Science*, volume 3527. Springer-Verlag, Pisa, Italy, p. 1–17, June 2005.

[4] Blair G.S., Coulson G., Andersen A., Blair L., Clarke M., Costa F.M., Duran-Limon H.A., Fitzpatrick T., Johnston L., Moreira R.S., Parlavantzas N., Saikoski K.B. The Design and Implementation of Open ORB 2. *IEEE Distributed Systems Online*, **2(6)**, 2001.

[5] Bruneton E., Coupaye T., Leclercq M., Quéma V., Stefani J.B. The Fractal Component Model and its Support in Java. *Software: Practice and Experience – Special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, **36(11-12)**:p. 1257–1284. doi:10.1002/spe.767. John Wiley & Sons, August 2006.

[6] Coulson G., Blair G., Grace P., Taiani F., Joolia A., Lee K., Ueyama J., Sivaharan T. A Generic Component Model for Building Systems Software. *ACM Transaction on Computer Systems*, **26(1)**:p. 1–42. doi:10.1145/1328671.1328672, February 2008.

[7] Dashofy E.M., van der Hoek A., Taylor R.N. An infrastructure for the rapid development of XML-based architecture description languages. *Proceedings of the 22rd International Conference on Software Engineering (ICSE'02)*. ACM, Orlando, Florida, USA, p. 266–276. doi:10.1145/581339.581374, Mai 2002.

[8] DeMichiel L., Keith M. *Enterprise JavaBeans (EJB) Specification*. Sun Microsystems, Inc., Santa Clara, California, U.S.A, 3.0 edition, December 2005.

[9] Ecma International. *C# Language Specification*. Geneva, Switzerland, 3.0 edition, June 2005.

[10] Eichberg M., Schäfer T., Mezini M. Using Annotations to Check Structural Properties of Classes. *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE'05)*, *Lecture Notes in Computer Science*, volume 3442. Springer-Verlag, Edinburgh, UK, p. 237–252, April 2005.

[11] Fassino J.P., Stefani J.B., Lawall J.L., Muller G. Think: A Software Framework for Component-based Operating System Kernels. *USENIX Annual Technical Conference, General Track*. Monterey, California, USA, p. 73–86, June 2002.

[12] Flissi A., Dubus J., Dolet N., Merle P. Deploying on the Grid with DeployWare. *Proceedings of the 8th International Symposium on Cluster Computing and the Grid (CCGrid'08)*. Lyon, France, p. 177–184, Mai 2008.

[13] Gosling J., Joy B., Steele G., Bracha G. *The Java Language Specification, Third Edition*. Addison-Westley Professional Computing, Santa Clara, California, USA, December 2005.

[14] Hamilton G. *JavaBeans Specification*. Sun Microsystems, Inc., San Antonio Road, Palo Alto, CA, 1.01 edition, August 1997.

[15] Johnson R., Hoeller J., Arendsen A., Sampaleanu C., Harrop R., Risberg T., Davison D., Kopylenko D., Pollack M., Templier T., Vervaet E., Tung P., Hale B., Colyer A., Lewis J., Leau C., Evans R. *The Spring Framework - Reference Documentation*, 2.0.6 edition, 2007.

[16] Leclercq M., Özcan A.E., Quéma V., Stefani J.B. Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset. *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. IEEE Computer Society, Minneapolis, USA, p. 209–219. doi:10.1109/ICSE.2007.82, Mai 2007.

[17] Leclercq M., Quéma V., Stefani J.B. DREAM: A Component Framework for Constructing Resource-Aware Configurable Middleware. *IEEE Distributed Systems Online*, **6(9)**, September 2005.

[18] Louis A. Use JBI Components for Integration. JavaWorld.com, July 2006.

[19] Medvidovic N., Taylor R.N. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, **26(1)**:p. 70–93, January 2000.

[20] Noguera C., Duchien L. Annotation Framework Validation using Domain Models. *Proceedings of the 4th European Conference on Model Driven Architecture Foundations and Applications (ECMDA'08)*. Springer-Verlag, Berlin, Germany, June 2008.

[21] OMG. *CORBA Component Model (CCM) Specification*. Needham, MA, USA, 3.0 edition, September 2002.

[22] Open Service Oriented Architecture. *SCA: Service Component Architecture*, 1.0 edition. Java Common Annotations and APIs, March 2007.

[23] Pawlak R. Spoon: Compile-time Annotation Processing for Middleware. *IEEE Distributed Systems Online*, **7(11)**, November 2006.

[24] Rouvoy R., Conan D., Seinturier L. Software Architecture Patterns for a Context-Processing Middleware Framework. *IEEE Distributed Systems Online*, **9(6)**, June 2008.

[25] Rouvoy R., Merle P. Leveraging Component-Oriented Programming with Attribute-Oriented Programming. *Proceedings of the 11th International ECOOP Workshop on Component-Oriented Programming (WCOP'06)*, *Technical Report*, volume 2006–11. Karlsruhe University, Nantes, France, July 2006.

[26] Rouvoy R., Pessemier N., Pawlak R., Merle P. Using Attribute-Oriented Programming to Leverage Fractal-Based Developments. *Proceedings of the 5th International ECOOP Workshop on Fractal Component Model*. Nantes, France, July 2006.

[27] Rouvoy R., Serrano-Alvarado P., Merle P. Towards Context-Aware Transaction Services. *Proceedings of the 6th International IFIP Conference on Distributed Applications and Interoperable Systems (DAIS)*, *Lecture Notes in Computer Science*, volume 4025. Springer-Verlag, Bologna, Italy, p. 272–288, June 2006.

[28] Schiavoni V., Quéma V. *A Posteriori* Defensive Programming: An Annotation Toolkit for DoS-Resistant Component-Based Architectures. *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06)*. ACM, Dijon, France, p. 1734–1738, April 2006.

[29] Seinturier L., Pessemier N., Duchien L., Coupaye T. A Component Model Engineered with Components and Aspects. *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, *Lecture Notes in Computer Science*, volume 4063. Springer-Verlag, Stockholm, Sweden, p. 139–153, June 2006.

[30] Stahl T., Volter M., Stockfleth B.V. *Model-driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, July 2006.

[31] Wada H., Suzuki J. Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming. *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05)*, *Lecture Notes in Computer Science*, volume 3713. Springer-Verlag, Montego Bay, Jamaica, p. 584 – 600, October 2005.

[32] Walls C., Richards N. *XDoclet in Action*. In Actions series, Manning Publications, December 2003.