A FLEXIBLE MODEL AND IMPLEMENTATION OF COMPONENT CONTROLLERS

Françoise Baude, Denis Caromel, Ludovic Henrio and Paul Naoumenko
INRIA Sophia - I3S - CNRS - Université de Nice Sophia Antipolis
{fbaude,dcaromel,lhenrio, pnaoumen}@sophia.inria.fr

Abstract The GCM (Grid Component Model) is a component model that is being defined by the CoreGRID institute on Programming Models; it is based on the Fractal component model. It is intended at overcoming the insufficiencies of the existing component systems when it comes to Grid computing. Its main characteristics are: hierarchical composition, structured communications with support for asynchrony, support for deployment, functional and non-functional (NF) adaptivity, and autonomicity. As in the Fractal component model, the GCM distinguishes *controllers* which implement NF concerns and are gathered in a *membrane* from the functional content of the component.

This article presents a refinement of the Fractal/GCM model and an API for adopting a component design of the component membranes, as suggested by the GCM specification. The objective of this framework is to provide support for both adaptivity and autonomicity of the component control part. In the design of the model refinement and the API for NF components, we also take into account hierarchical composition and distribution of the membrane, which is crucial in the GCM. Our approach is flexible because it allows "classical" controllers implemented by usual objects to coexist with highly dynamic and reconfigurable controllers implemented as components.

Keywords: GCM, component control, separation of concerns, autonomicity.

1. Introduction

Components running in dynamically changing execution environments need to adapt to these environments. In Fractal [3] and GCM (Grid Component Model) [4] component models, adaptation mechanisms are triggered by the non-functional (NF) part of the components. This NF part, called the *membrane*, is composed of *controllers* that implement NF concerns. Interactions with execution environments may require complex relationships between controllers. In this work we focus on the adaptability of the *membrane*. Examples include changing communication protocols, updating security policies, or taking into account new runtime environments in case of mobile components. Adaptability

implies that evolutions of the execution environments have to be detected and acted upon, and may also imply interactions with the environment and with other components for realizing the adaptation.

We want to provide tools for adapting controllers. This means that these tools have to manage (re)configuration of controllers inside the membrane and the interactions of the membrane with membranes of other components. For this, we provide a model and an implementation, using a standard component-oriented approach for both the application (functional) level and the control (NF) level. Having a component-oriented approach for the non-functional aspects also allows them to benefit from the structure, hierarchy and encapsulation provided by a component-oriented approach.

In this paper, we propose to design NF concerns as compositions of components as suggested in the GCM proposal. Our general objective is to allow controllers implemented as components to be directly plugged in a component membrane. These controllers take advantage of the properties of component systems like *reconfigurability*, i.e. changing of the contained components and their bindings. This allows components to be dynamically adapted in order to suit changing environmental conditions. Indeed, among others, we aim at a component platform appropriate for *autonomic Grid applications*; those applications aim to ensure some quality of services and other NF features without being geared by an external entity.

In this paper we provide a twofold contribution: first, refinements of the Fractal/GCM model concerning the structure of a membrane; second, a definition and an implementation of an API that allows GCM membranes to be themselves composed of components, possibly distributed. Both for efficiency and for flexibility reasons, we provide an implementation where controllers can either be classical objects or full components that could even be distributed. We believe that this high level of flexibility is a great advantage of this approach over the existing ones [8, 7]. Our model refinements also provide a better structure for the membrane and a better decoupling between the membrane and its externals. Finally, our approach gives the necessary tools for membrane reconfiguration, providing flexibility and evolution abilities. The API we present can be split in three parts:

- Methods dedicated to component instantiation: they allow the specification of a NF type of a component, and the instantiation of NF components;
- Methods for the *management of the membrane*: they consist in managing the content, introspecting , and managing the life-cycle of the membrane. Those methods are proposed as an extension of the Fractal component model, and consequently of the GCM;
- An optional set of methods allowing *direct operations on the components that compose the membrane*: they allow introspection, bindings and life-cycle management of the components inside the membrane, as would be

possible using the Fractal API extended with the previously mentioned methods. They take into consideration the distributed nature of the GCM.

This paper is organized as follows. Section 2 presents refinements of the Fractal/GCM model and the API for (re)configuring the membrane; then Section 3 presents the implementation of the API, using GCM/ProActive; Section 4 presents the related work; finally Section 5 concludes.

2. Componentizing Component Controllers

After an example motivating our approach, this section describes the structure of the membrane and primitives for creating and manipulating NF components. Indeed, the purpose of our approach is to design the management of the application as a component system; that is why we want to adopt a GCM design for the NF part of a component. Consequently, like any GCM component, the ones inside the membrane can be distributed. Thanks to such a design, NF requests can be triggered by external (NF) components in a much more structured way. For autonomicity purposes, reconfigurations can be triggered by controllers belonging to the membrane itself.

2.1 Motivating Example

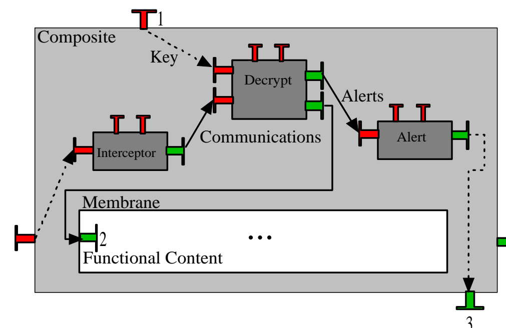

Figure 1. Example: architecture of a naive solution for secure communications

Here we present a simple example that shows the advantages of componentizing controllers of GCM components. In our example, we are considering a naive solution for securing communications of a composite component. As described in Figure 1, secure communications are implemented by three components inside the membrane: Interceptor, Decrypt, and Alert. The scenario of the example is the following: the composite component receives encrypted messages on its server functional interface. The goal is to decrypt those messages. First, the incoming messages are intercepted by the Interceptor component. It forwards all the intercepted communications to Decrypt, which can be an

off-the-shelf component (written by cryptography specialists) implementing a specific decryption algorithm. The Decrypt component receives a key for decryption through the non-functional server interface of the composite (interface number 1 on the figure). If it successfully decrypts the message, the Decrypt component sends it to the internal functional components, using the functional internal client interface (2). If a problem during decryption occurs, the Decrypt component sends a message to the Alert component. The Alert component is charge to decide on how to react when a decryption fails. For example, it can contact the sender (using the non-functional client interface – 3) and ask it to send the message again. Another security policy would be to contact a "trust and reputation" authority to signal a suspicious behaviour of the sender. The Alert component is implemented by a developer who knows the security policy of the system. In this example, we have three well-identified components, with clear functionalities and connected through well-defined interfaces. Thus, we can dynamically replace the Decrypt component by another one, implementing a different decryption algorithm. Also, for changing the security policy of the system, we can dynamically replace the Alert component and change its connexions. Compared to a classical implementation of secure communications (for example with objects), using components brings to the membrane a better structure and reconfiguration possibilities. To summarize, componentizing the membrane in this example provides dynamic adaptability and reconfiguration; but also re-usability and composition from off-the-shelf components.

2.2 A Structure for Componentized Membranes

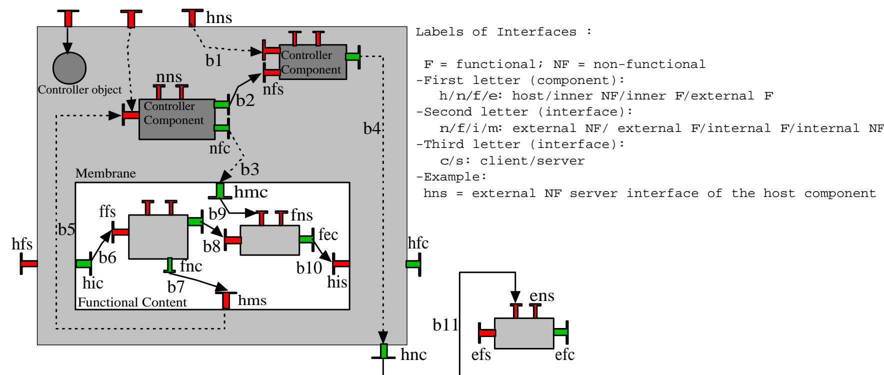

Figure 2. New structure for the membrane of Fractal/GCM components

Figure 2 shows the structure we suggest for the component membrane. The membrane (in gray) consists of one object controller and two component controllers, the component controllers are connected together and with the outside

of the membrane by different bindings. For the moment, we do not specify whether components are localized with the membrane, or distributed.

Before defining an API for managing components inside the membrane, the definition of the membrane given by the GCM specification needs some refinements. Those refinements, discussed in this section, provide more details about the structure a membrane can adopt. Figure 2 represents the structure of a membrane and gives a summary of the different kinds of interface roles and bindings a GCM component can provide. As stated in the GCM specification, NF interfaces are not only those specified in the Fractal specification, which are only external server ones. Indeed, in order to be able to compose NF aspects, the GCM requires the NF interfaces to share the same specification as the functional ones: role, cardinality, and contingency. For example, in GCM, client NF interfaces allow for the composition of NF aspects and reconfigurations at the NF level. Our model is also flexible, as all server NF interfaces can be implemented by both objects or components controllers.

All the interfaces showed in Figure 2 give the membrane a better structure and enforce decoupling between the membrane and its externals. For example, to connect *nfc* with *fns*, our model adds an additional stage: we have first to perform binding *b3*, and then binding *b9*. This avoids *nfc* to be strongly coupled with *fns*: to connect *nfc* to another *fns*, only binding *b9* has to be changed.

In Figure 2, some of the links are represented with dashed arrows. Those links are not real bindings but "alias" bindings (e.g. b3); the source interface is the alias and it is "merged" with the destination interface. These bindings are similar to the export/import bindings existing in Fractal (b6, b10) except that no interception of the communications on these bindings is allowed.

Performance Issues While componentizing the membrane clearly improves its programmability and its capacity to evolve, one can wonder what happens to performance. First, as our design choice allows object controllers, one can always keep the efficiency of crucial controllers by keeping them as objects. Second, the overhead for using components instead of objects is very low if the controllers components are local, and are negligible compared to the communication time, for example. Finally, if controllers components are distributed, then there can be a significant overhead induced by the remote communications, but if communications are asynchronous, and the component can run in parallel with the membrane, this method can also induce a significant speedup, and a better availability of the membrane. To summarize, controllers invoked frequently and performing very short treatments, would be more efficiently implemented by local objects or local components. For controllers called less frequently or which involve long computations, making them distributed would improve performances and availability of the membrane.

2.3 An API for (Re)configuring Non-functional Aspects

2.3.1 Non-functional Type and Non-functional Components. To type-check bindings between membranes, we have to extend the GCM model with a new concept: the *non-functional type* of a component. This type is defined as the union of the types of NF interfaces the membrane exposes. To specify the NF type of a component, we propose to overload the Fractal `newFcInstance` method (the one to create functional components) as follows:

```
public Component newFcInstance(Type fType,Type nfType, any contentDesc, any
    controllerDesc);
```

In this method, `nfType` represents the NF type of the component; it can be specified by hand. Of course the standard Fractal type factory has to be extended in order to support all possible roles of NF interfaces.

The NF type can also be specified within a configuration file: the controller descriptor argument (`controllerDesc`) can be a file written in Architecture Description Language (ADL) containing the whole description of the NF system as we will suggest in Section 3.2.

Components inside the membrane are *non-functional components*. They are similar to functional ones. However, their purpose is different because they deal with NF aspects of the *host component*. Thus, in order to enforce separation of concerns, we restrict the interactions between functional and NF components. For example, a NF component cannot be included inside the functional content of a composite. Inversely, a functional component cannot be added inside a membrane. As a consequence, direct bindings between functional interfaces of NF and functional components are forbidden.

To create NF components, we extend the common Fractal factories (generic factory and ADL factory). For generic factory, we add a method named `newNFcInstance` that creates this new kind of components:

```
public Component newNFcInstance(Type fType,Type nfType, any contentDesc, any
    controllerDesc);
```

Parameters of this method are identical to its functional equivalent and NF components are created the same way as functional ones. To create NF components using Fractal ADL[2], developers need to modify or add some of the modules within the factory. These modules depend on the implementation of the newNFcInstance method and on the Fractal/GCM implementation.

2.3.2 General Purpose API. To manipulate components inside membranes, we introduce primitives to perform basic operations like adding, removing or getting a reference on a NF component. We also need to perform calls on well-known Fractal controllers (*life-cycle controller*, *binding controller*, . . .) of these components. So, we extend Fractal/GCM specification by adding a new controller called *membrane controller*. As we want it to manage all

```
public void addNFSubComponent(Component component) throws
   IllegalContentException;
public void removeNFSubComponent(Component component) throws
   IllegalContentException, IllegalLifeCycleException, NoSuchComponentException;
public Component[] getNFcSubComponents();
public Component getNFcSubComponent(string name) throws NoSuchComponentException;
public void setControllerObject(string itf, any controllerclass) throws
     NoSuchInterfaceException;
public void startMembrane() throws IllegalLifeCycleException;
public void stopMembrane() throws IllegalLifeCycleException;
```

Figure 3. General purpose methods defined in MembraneController interface

the controllers, it is the only mandatory controller that has to belong to any membrane. It allows the manual composition of membranes by adding the desired controllers. The methods presented in Figure 3 are included in the *MembraneController* interface; they are the core of the API and are sufficient to perform all the basic manipulations inside the membrane. They add, remove, or get a reference on a NF component. They also allow the management of object controllers and membrane's life-cycle. Referring to Fractal, this core API implements a subset of the behavior of the life-cycle and content controllers specific to the membrane. This core API can be included in any Fractal/GCM implementation. Reconfigurations of NF components inside the membrane are performed by calling standard Fractal controllers. The general purpose API defines the following methods:

- `addNFSubComponent(Component component)`: adds the NF component given as argument to the membrane;
- `removeNFSubComponent(Component component)`: removes the specified component from the membrane;
- `getNFcSubComponents()`: returns an array containing all the NF components;
- `getNFcSubComponent(string name)`: returns the specified NF component, the string argument is the name of the component;
- `setControllerObject(string itf, any controllerclass)`: sets or replaces an existing controller object inside the membrane. `Itf` specifies the name of the control interface which has to be implemented by the controller class, given as second parameter. Replacing a controller object at runtime provides a very basic adaptivity of the membrane;
- `startMembrane()`: starts the membrane, i.e. allows NF calls on the host component to be served. This method can adopt a recursive behavior, by starting the life-cycle of each NF component inside the membrane;
- `stopMembrane()`: Stops the membrane, i.e. prevents NF calls on the host component from being served except the ones on the membrane controller. This method can adopt a recursive behavior, by stopping the life-cycle of each NF component.

```
public void bindNFc(String clientItf, String serverItf) throws
   NoSuchInterfaceException, IllegalLifeCycleException,
   IllegalBindingException, NoSuchComponentException;
public void bindNFc(String clientItf, Object serverItf) throws
   NoSuchInterfaceException, IllegalLifeCycleException,
   IllegalBindingException, NoSuchComponentException;
public void unbindNFc(String clientItf) throws NoSuchInterfaceException,
   IllegalLifeCycleException, llegalBindingException, NoSuchComponentException;
public String[] listNFc(String component) throws NoSuchComponentException;
public Object lookupNFc(String itfname) throws NoSuchInterfaceException,
   NoSuchComponentException;
public void startNFc(String component) throws IllegalLifeCycleException,
   NoSuchComponentException;
public void stopNFc(String component) throws IllegalLifeCycleException,
     NoSuchComponentException;
public String getNFcState(String component) throws NoSuchComponentException;
```

Figure 4. Distribution specific methods implemented by MembraneController

2.3.3 Distribution-specific API. Considering the distribution aspect of the GCM, we provide an extension to the core API. As usual in distributed programming paradigms, GCM objects/components can be accessed locally or remotely. Remote references are accessible everywhere, while local references are accessible only in a restricted address space. When returning a local object/component outside its address space, there are two alternatives: create a remote reference on this entity; or make a copy of it. When considering a copy of a NF local component, the NF calls are not consistent. If an invocation on `getNFcSubComponent(string name)` returns a copy of the specified NF component, calls performed on this copy will not be performed on the "real" NF component inside the membrane. Figure 4 defines a set of methods that solves this problem. As copies of local components result in inconsistent behavior, the alternative we adopt is to address NF components by their names instead of their references. These methods allow to make calls on the binding controller and on the life-cycle controller of NF components that are hosted by the component membrane. Currently, they don't take into account the hierarchical aspect of local NF components.

Somehow this new API can be considered as higher level operations compared to the API of Figure 3. Indeed, they address the NF components and call their controllers at once. For example, here is the Java code that binds two components inside the membrane using the general purpose API. It binds the interface "i1" of the component "nfComp1" inside the membrane to the interface "i2" of the component "nfComp2". Suppose `mc` is a reference to the *MembraneController* of the host component.

```
Component nfComp1=mc.getNFcSubComponent("nfComp1");
Component nfComp2=mc.getNFcSubComponent("nfComp2");
Fractal.getBindingController(nfComp1).bindFc("i1",nfComp2.getFcInterface("i2"));
```

But, if the code above is executed by an entity outside the membrane and "nfComp1" is a passive component; then it is not the component "nfComp1" inside the membrane that is bound to "nfComp2" but a copy of it. Using the API of Figure 4, this binding can be realized by the following code, that binds the component "nfComp1" correctly, regardless of whether it is active or passive

```
mc.bindNFc("nfComp1.i1","nfComp2.i2");
```

Similarly to the example above, all the methods of Figure 4 result in calls on well-known Fractal controllers. Interfaces are represented as strings of the form *component.interface*, where *component* is the name of the inner component and *interface* is the name of its client or server interface. We use the name "membrane" to represent the membrane of the host component, e.g. `membrane.i1` is the NF interface i1 of the host component; in this case *interface* is the name of an interface from the NF type. For example, `bindNFc(string, string)` allows to perform the bindings: *b1*, *b2*, *b4*, *b3*, *b9*, *b7* and *b5* of Figure 2.

The two parts of our API (Figures 3 and 4) can be included in two separate interfaces. Then developers can choose to implement one or both of these interfaces inside each component.

3. Implementation and Ongoing Work

3.1 Context

The ProActive library is a middleware for Grid computing, written in Java, based on activities, with asynchronous communications and providing a deployment framework. A GCM/ProActive component is instantiated by an activity, i.e. an active object, some passive objects, together with a request queue and a single thread. Because the smallest unit of composition is an activity, GCM/ProActive may be considered as a *coarse-grained* implementation of the Fractal/GCM component model w.r.t. the Julia or the AOKell implementations, where the smallest unit of composition is a (passive) object. A reference implementation of the GCM is being implemented over ProActive, it follows its programming model, especially concerning remote objects/components (called *active objects/components*) and local objects/components (called *passive objects/components*).

3.2 Current Limitations and Future Work

Currently, we have implemented in GCM/ProActive the structure proposed in the previous sections with most of the suggested interfaces and API. Our `MembraneController` is able to manage NF active components and passive objects as controllers. One of the strong points of our implementation is that membranes can consist of both passive objects and active components.

We review below the current status of the implementation and the main limitations that have to be addressed in the future.
Support Passive Components We do not support passive components for the moment. We will investigate on strategies to include also passive NF components. The idea consists in reusing components from existing frameworks (e.g. Julia or AOKell).
Describing the membrane by an ADL For the moment, the only way to instantiate NF components inside the membrane is programmatically: first create a NF component with the `newNFcInstance` method, second add and bind the NF components thanks to invocations on the membrane controller. These operations can be performed either by an external entity (e.g. another component or the framework that instantiates the host component) or by an autonomic controller inside the membrane. In addition to this manual method, we want the developer to be able to describe whole membrane in a separate file, given as last argument of the `newFcInstance` method. This way, the membrane can be designed separately from the functional content.

Considering the membrane as a composite component eases its description with its set of interfaces, objects and internal components; it also allows us to describe the membrane in an (extended) ADL language. External functional interfaces of this composite correspond to both internal and external NF interfaces of the host component. Then membrane's description can be referenced as the controllers description inside the functional ADL. This ADL "composite view" is only necessary at design time: when the membrane is actually created, the composite component should be "dissolved" inside its host component, i.e. the host component will be the parent of the functional and NF components. This avoids unnecessary intermediate interfaces, and having to deal with the "membrane's membrane". In Figure 5, the membrane is a composite drawn with a dashed border line which does not exist at runtime.

4. Related Work

First of all, our approach for adaptivity has the great advantage to allow the usage of many related works. Indeed, by using a component-based approach for the design and implementation of controllers, we can also apply existing knowledge on self-adaptativity at the application level (e.g., [5]) to the NF level.

Other research teams have also proposed to provide a component-based model for the implementation of membranes. Control microcomponents from the Asbaco project [7] are specific components using a control API different from Fractal's, and requiring a specific ADL language (because microcomponents are injected following an aspect oriented approach).

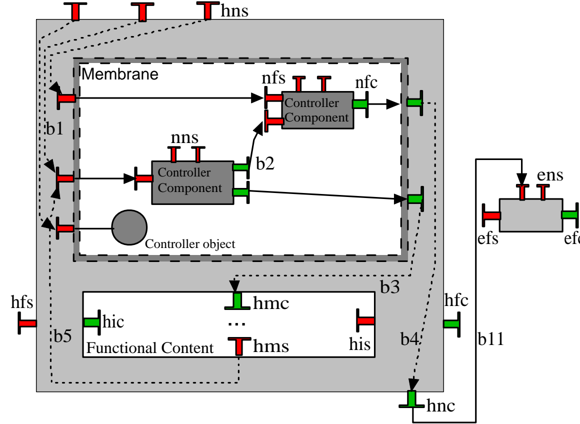

Figure 5. The membrane is designed as a composite component

AOKell [8–9] proposes a component-based approach using the Fractal API for engineering membranes, but their controller components cannot be distributed, neither collaborate with object controllers. Moreover, the membrane is necessarily designed and executed as a composite component entailing one additional level of indirection for requests toward the membrane components. Recently, Julia's Fractal implementation moved towards components in the membrane; their approach is similar to the one of AOKell.

In [6], the authors also advocate the componentization of the membranes in order to enable the dynamic choice between various implementations of the same technical service according to the runtime context of a Fractal component.

Our approach has some differences with the ones cited above. We provide more details about the structure of a membrane. The first useful concept is the NF type: it helps to decide at runtime which NF interfaces will be exposed by the membrane and to type-check NF bindings. Moreover, new roles have been introduced for NF interfaces. This helps developers to have more control over NF bindings and to design the membrane separately from the functional content with well defined communication points. Finally, our approach is more flexible, because component and object controllers can coexist in the same membrane, partially thanks to the absence of an intermediate component for the membrane.

5. Conclusion

In this paper, we provide refinements of the GCM component model, an API, and a partial implementation allowing distributed components and local objects to coexist as controllers of the same membrane. Model refinements provide a better structure for the membrane. The API includes the specification of component NF type, the creation of NF components, and membrane management

with the *MembraneController*. The flexible and adaptable membrane design presented in this article provides a basis for easing the dynamic management of interactions between controllers of the same or distinct components. Including the future work presented in Section 3.2, we plan to experiment some autonomic adaptation scenarios. Thanks to these scenarios, we will evaluate the capability of the membrane to orchestrate two kinds of reconfigurations: reconfiguration of the functional inner component system, following the idea of hierarchical autonomic decision paths [1]; and reconfiguration of the membrane itself when the adaptation is related to NF properties of the host component.

References

[1] M. Aldinucci, C. Bertolli, S. Campa, M. Coppola, M. Vanneschi, and C. Zoccolo. Autonomic Grid Components: the GCM Proposal and Self-optimising ASSIST Components. In *Joint Workshop on HPC Grid programming Environments and Components and Component and Framework Technology in High-Performance and Scientific Computing at HPDC'15*, June 2006.

[2] E. Bruneton. Fractal ADL Tutorial `http://fractal.objectweb.org/tutorials/adl/index.html`. Technical report, ObjectWeb Consortium, March 2004.

[3] E. Bruneton, T.Coupaye, and J. Stefani. The Fractal Component Model `http://fractal.objectweb.org/specification/index.html`. Technical report, ObjectWeb Consortium, February 2004.

[4] CoreGRID Programming Model Virtual Institute. Basic features of the grid component model (assessed), 2006. Deliverable D.PM.04, CoreGRID, Programming Model Institute.

[5] P. David and T. Ledoux. Towards a framework for self-adaptive component-based applications. In J.-B. Stefani, I. Demeure, and D. Hagimont, editors, *Proceedings of Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6.1 International Conference, DAIS 2003*, volume 2893 of *Lecture Notes in Computer Science*, pages 1–14, Paris, Nov. 2003. Federated Conferences, Springer-Verlag.

[6] C. Herault, S. Nemchenko, and S. Lecomte. A Component-Based Transactional Service, Including Advanced Transactional Models. In *Advanced Distributed Systems: 5th International School and Symposium, ISSADS 2005, Revised Selected Papers*, number 3563 in LNCS, 2005.

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

[8] L. Seinturier, N. Pessemier, and T. Coupaye. AOKell: an Aspect-Oriented Implementation of the Fractal Specifications, 2005. `http://www.lifl.fr/~seinturi/aokell/javadoc/overview.html`.

[9] L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A component model engineered with components and aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, Lecture Notes in Computer Science. Springer, June 2006.