Microcomponent-Based Component Controllers: A Foundation for Component Aspects [*]

Vladimir Mencl[1]

[1]*Charles University*
Faculty of Mathematics and Physics
Department of Software Engineering
Prague, Czech Republic
{mencl,bures}@nenya.ms.mff.cuni.cz
http://nenya.ms.mff.cuni.cz/

Tomas Bures[1,2]

[2]*Academy of Sciences of the Czech Republic*
Institute of Computer Science
Prague, Czech Republic
bures@cs.cas.cz
http://www.cs.cas.cz/

Abstract

In most component models, a software component consists of a functional part and a controller part. The controller part may be extensible; however, existing component models provide no means to capture the structure of the controller part, and therefore neither to specify the controller part extensions.

In this paper, we introduce a minimalist component model to capture the structure of the controller part, coining the term microcomponent *for the controller part elements. We further introduce the concept of a* component aspect *as a consistent set of controller part extensions. Within this framework, it is possible to seamlessly integrate controller part extensions, applying them to the components selected in the application's* launch configuration. *We have evaluated these concepts in a prototype implementation.*

Keywords: component controllers, component models

1. Introduction & Motivations

1.1. Background

Component models [2, 5, 18, 8, 10, 15] have been in the research focus for a number of years; a part of the general consensus is that components communicate via a set of server and client interfaces (also called provided and required); many of the component models are hierarchical, permitting a component to be composed of subcomponents. A component typically consists of a functional part and a *controller part* – e.g., in the Fractal component model [2, 3] and in SOFA [15]. Functional part is also called *business part*, e.g., in EJB [18]. The controller part (also *controller* for short) may take the form of a fixed set of API functions specific to the concrete component model.

We focus on Fractal since it features explicit control interfaces, navigable via introspection in the same fashion as functional interfaces. The control interfaces expose the control features of the controller part of a component, allowing for easily extending the component model with new features. In the same vein, these extensions may introduce additional control interfaces to access the new control features.

1.2. Motivations

The possible component model extensions may range from a simple controller providing only a collection of getter/setter methods (e.g., to store the specification of the component behavior) through controller part elements interacting with other controller elements, to complex collections of interacting objects, intercepting calls on possibly both functional and control interfaces of the component. In specific cases, the extension functionality may have a complex structure which would be the best captured in a standalone *control* component. For example, interceptors monitoring behavior on the functional interfaces might interact with a central logging control component.

While the Fractal component model permits the controller part to be extended, it provides no means to specify such extensions. From our experience with developing controller extensions, we have identified the following properties of the extension mechanism as important: (1) support to easily combine different con-

---

troller extensions, (2) means to select individual component instances to which the extensions will be applied, (3) encapsulation of complex control functionality in standalone control components, specified in separate ADL descriptions; even such extensions should not affect the functional part of the application's architecture, nonetheless, (4) the extension mechanism should also capture the ties (bindings) between the controller part extensions and the control components.

At the abstract level of the Fractal component model, there are no means to capture the structure of the control part. Consequently, there is no unified and transparent way to specify controller part extensions. While this may be addressed by a Fractal implementation, none of them (in particular, Julia [13], AOKell [17], and SOFA [15]) does so in a satisfactory way.

Julia [13], the reference implementation of the Fractal component model, features a mechanism for specifying the contents of the controller part. However, this mechanism uses a notation which is hard to manage, and does not provide important aspects of the controller part structure, omitting the dependencies of the individual controller objects. Some of their requirements are specified in plain English in the documentation generated from the source code; some are not documented at all. Consequently, it is not possible to verify the consistency of a particular controller specification.

In addition, the Julia approach is heavily based on mixins, leading to several disadvantages: (i) extensions to existing controllers have to be developed as mixins, and consequently, (ii) extension developer must learn to implement mixins, (iii) the extensions are hard to debug (a merged generated class is executed instead of its constituent mixin classes), and (iv) it is hard to combine controller extensions, due to the notation used to specify the controller part contents.

AOKell [17], another implementation of Fractal, employs AspectJ [9] to weave the component content together with its controller; this mechanism may also be used to extend the controller with extensions developed as AspectJ aspects. However, this approach permits to select components to be extended only by component type and not by a particular instance, and does not address the other issues we have identified, such as the need for control components. The SOFA component model [15], also an implementation of Fractal, does not provide means to extend the component controller, and neither features a mechanism to capture the controller part structure.

1.3. Goals & Structure of the Paper

The goals of this paper are twofold: first, to introduce a component-based model for explicitly capturing the structure of the component controller part, and afterwards, based on this model, to introduce an aspect-based model for specifying controller part extensions; the extension model we present is designed to fulfill the properties (1-4) described in the previous section.

We describe results achieved in the *Asbaco* project (*Aspect-Based Controllers*). The microcomponent model and the extension mechanism we have developed within this project are applicable to the Fractal component model in general, not specific to a particular implementation, and are possibly applicable also to other component models.

The paper is structured as follows: in Sect. 2, we introduce the Asbaco microcomponent model for capturing the internal structure of the component controller part. Next, in Sect. 3, we introduce an aspect-based model for specifying controller extensions. We evaluate our results and discuss a case study in Sect. 4. The Sections 5 and 6 compare our approach to related work, and draw a conclusion.

2. Asbaco Microcomponent Model

In the Asbaco project, we use the component concept to capture the structure of the component controller; for this task, we employ a minimalist component model. We coin the term *microcomponent* to refer to a component forming a part of the component controller; we reserve the term *component* to refer to the host component (or another component in the host component model).

In this section, we introduce the key features of the microcomponent model. We postpone introducing the XML-based microcomponent architecture description language until Sect. 3, focused on specifying consistent controller extensions. Within this section, we use a diagrammatic notation to describe controller part configurations (figures 1 and 3 described later in this section).

The microcomponent model is flat; a microcomponent may not contain other microcomponents. To avoid undesired recursion in the component model definition, microcomponents do not have a controller part, and control interfaces of a microcomponent are implemented by its content part. As the microcomponent model is a minimalist component model, there are only few interfaces to be implemented; the only control features to be implemented by a microcomponent are the interfaces required for establishing bindings. In particular, a microcomponent does not provide interfaces for introspection (e.g., interface `Component` in Fractal).
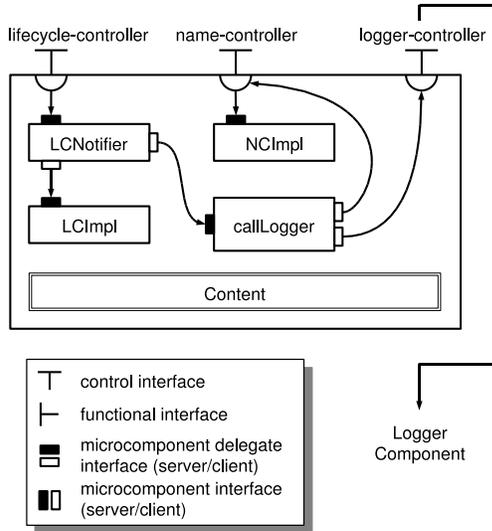
Figure 1. Fragment of microcomponent-based controller part.

2.1. Basic Microcomponent Features

In this section, we introduce the basic features of the microcomponent model; we illustrate them on the fragment of a sample controller part shown in Fig. 1. Each microcomponent has a set of *client* and *server* interfaces. A server interface represents the services offered by the microcomponent. A client interface must be bound either to a server interface of another microcomponent, or to an external control interface of the host component (may be either client or server). A server control interface of the host component may *delegate* to a microcomponent, while a client control interface should delegate to a functional interface of a standalone control component (control components will be described in Sect. 2.3). To capture the common pattern of a *delegation chain*, a microcomponent may designate one server and one client interface as delegate interfaces.

Figure 1 illustrates most of the concepts on a hypothetical controller extension monitoring the activity on the `lifecycle-controller` external control interface via a standalone control component `Logger`. The `lifecycle-controller` interface is delegated to the `LCNotifier` microcomponent featuring a server delegate, a client delegate, and a regular client interface. `LCNotifier` further delegates the incoming calls to the original lifecycle implementation, microcomponent `LCImpl`. The client interface of `LCNotifier` is bound to the only server interface of the standalone microcomponent `callLogger`.

The `callLogger` microcomponent features also two client interfaces. The first one is bound to the server control interface `name-controller`. While the effect is similar to establishing a binding directly to the microcomponent `NCImpl` implementing this control interface, binding to a control interface of the host component assures that the binding will be always established to the very start of the delegation chain, even when a new microcomponent is inserted there. The second client interface of `callLogger` is bound to the external client control interface `logger-controller`, which is bound to the standalone control component `logger`.

Technically, each microcomponent interface must specify its name, type (signature) and role (`client`, `server`, `delegateclient`, or `delegateserver`). As the delegate interfaces are bound implicitly, they are not required to specify a name.

2.2. Intercepting Functional Interfaces

We now demonstrate how the microcomponent model supports intercepting calls on interfaces of the host component. Intercepting a control interface is rather straightforward: an intercepting microcomponent can be created specifically for the type of the intercepted interface and inserted into the delegation chain. However, a more complex solution is required in the case of functional interfaces, where the interface type is not known at the time the microcomponent is developed.

To support interception on functional interfaces, the microcomponent model also introduces *dynamic microcomponents*, generated for each concrete interface type intercepted. Such microcomponents may feature dynamic delegate interfaces: a *dynamic delegate interface* does not specify its type, instead, the type of the interface is derived from the setting the microcomponent is being instantiated in.

Such interception microcomponents have to be generated for each type of functional interface encountered. A possible approach would be to manually develop an interceptor generator for each new controller extension. To ease the development of controller extensions, the Asbaco framework introduces a predefined generic interception interface (shown in Fig. 2 as a Java interface) and a generator constructing the interception microcomponents. These generic microcomponents can be inserted into the delegation chain of the concrete functional interface and interact with the extension-specific microcomponents via the predefined generic interface; the extension developer has only to provide the interface type independent microcomponents.

```
public interface InterceptorNotify {
  Object preInvoke( String interfaceTypeName,
        String interfaceName, String methodName,
        String methodSignature, Object parameters []);
  void postInvoke( String interfaceTypeName,
        String interfaceName, String methodName,
        String methodSignature, Object parameters [],
        Object context, Object retval, Exception exception );
}
```

Figure 2. Predefined interception interface.
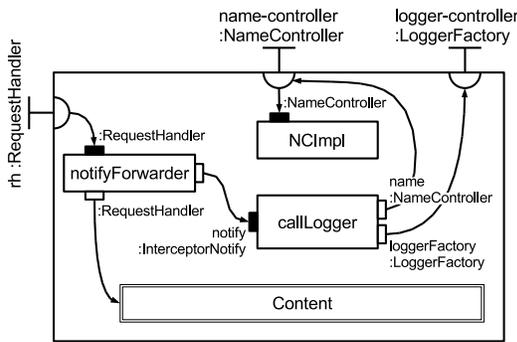
Figure 3. Fragment of controller part intercepting a functional interface.

Figure 3 demonstrates this mechanism on a hypothetical extension logging requests via a *log4j* logger. The extension is applied on the `rh` external interface of type `RequestHandler`. The `notifyForwarder` microcomponent is generated by the Asbaco framework. Besides the delegate-server and delegate-client interfaces, the microcomponent has a `notify` interface of the predefined type `InterceptorNotify` (fig. 2). The microcomponent `callLogger` is the only component to be created by the extension developer; the sole task of this microcomponent is to gather information about the intercepted calls via the `notify` interface and emit log messages on the `loggerFactory` interface.

Note that this simple interception mechanism only provides notifications about the call being performed, and does not permit to alter the call. There are a number of features which might be possibly supported by the interception mechanism, e.g., the extension-specific microcomponent might be permitted to alter the parameters or to possibly block the call. This area is subject of further research in the Asbaco project. The bottom line is that while the Asbaco framework provides a predefined interface and a microcomponent generator to handle the most common situation, the microcompo-

nent model permits to employ custom generators constructing interception microcomponents tailored to the particular needs.

2.3. Control Components and Client Control Interfaces

The functionality of a controller extension may include a part that has itself complex structure which would be the best captured in a software component. Such a part may also serve as a "backend" for the controller extensions, and it may be desirable that it exists in only one singleton instance. Therefore, we introduce the concept of a *control component*. A control component is specified in the same way as the top-level component of an application (i.e., using the architecture description language). As the need to instantiate the control component comes from the controller extension and not the application, a control component is not instantiated inside the application top-level component, but forms a separate orthogonal hierarchy.

The controller extension specifies the control components to be instantiated for this extension (by referring to ADL specifications of these components). The controller extension also specifies the bindings to be established between functional interfaces of the control component and control interfaces of the component to which the extension is applied. As control components may have client and server functional interfaces, we introduce *client control interfaces* to permit binding to a functional server interface of a control component. A client control interface is a connection point for microcomponents to access the functionality offered by the control component. The binding of the client control interface to the control component is established at the time the controller part is created; a client control interface may be accessed by all microcomponents in the controller part. Please note that client control interface is an extension with respect to the current Fractal specification [3].

2.4. Concrete Model: Mapping Asbaco Microcomponents to Java

We introduce a concrete mapping of the microcomponent model to the Java programming language; this mapping is also used in our prototype implementation. This mapping defines the concrete interfaces to be implemented by a microcomponent. The interfaces, inspired by Fractal `BindingController`, are `ClientBindings` and `DelegateBindings` (fig. 4). Due to the specific nature of the delegate interfaces, we have introduced separate interfaces for

```
public interface ClientBindings {
  public void bindMc(String itf , Object o);
  public void unbindMc(String itfName );
}
public interface DelegateBindings {
  public void bindMcDelegate(Object ref );
  public void unbindMcDelegate();
}
public interface Configurable {
  public void configureMc(Map parameters);
}
```

Figure 4. Microcomponent control interfaces.

regular bindings and for delegate bindings. One particular issue to resolve was the signature of these interfaces. We have considered reusing the Fractal `BindingController` interface, however, this would clash with the actual interfaces of a microcomponent situated in the delegation chain of the host component's binding controller. To avoid this clash, we have decided to use a different signature. In addition to the binding interfaces, the `Configurable` interface is used to provide configuration parameters to a microcomponent (in the form of a Java `Hashtable`).

A microcomponent is implemented by a single Java class. For simplicity, reference to an instance of the class serves also as reference to all provided interfaces of the microcomponent. A microcomponent is specified by the list of its interfaces and either the name of the implementing class, or, in the case of a dynamic microcomponent, the name of a microcomponent generator.

3. Extending Controllers via Aspects

Based on Asbaco microcomponent model presented in the previous section, we now introduce the concept of a *component aspect*. Following the general idea of aspect-oriented programming [9], a component aspect defines a consistent set of controller part extensions. The extensions introduced by a component aspect may include control components to be instantiated at the global scope, control interfaces (client and server) to extend the affected components, and microcomponents to be instantiated in their controller part.

3.1. Defining an Aspect

In the Asbaco framework, an aspect is specified in an XML language developed as an extension of the Fractal ADL language [11]. We demonstrate this language in Fig. 5 on the definition of the aspect `logging`, which enforces logging of all calls on functional interfaces via an external control component. This extension has already been used to illustrate the microcomponent model in Sect. 2; Fig. 3 shows the configuration of microcomponents for the case of a single interface.

An aspect definition starts with defining frame add-ons and microcomponent types. A *frame add-on* is a collection of control interfaces to be introduced into a component: the frame add-on `logger-itfs` consists of only a single client control interface `logger-controller`. A *microcomponent definition* defines a microcomponent type in terms of its interfaces and content. The microcomponent `callLogger` features a client and a server interface; its content is specified by the name of the implementing class. The *dynamic* microcomponent `notifyForwarder` features, besides the `notify` client interface, also a dynamic delegate-server and a delegate-client interface. The content of a dynamic microcomponent is specified by referring to a *microcomponent generator*, `InterceptorNotifyGenerator` in this case.

An aspect also specifies control components to be instantiated when the aspect is loaded by referring to their ADL definitions. In our example, the `logger` component is instantiated based on its definition in the file `logger-adl.fractal`.

The core of an aspect is the specification of how actually should components be extended; such specifications are enclosed in component and interface selectors. A *component selector* selects components by their type: either `composite`, `primitive`, or `any`.

The extensions to be applied to the selected component may include introducing new control interfaces (by referring to a frame add-on definition), microcomponents to be instantiated, bindings to be established either among microcomponents or between a control interface and a control component. In our example, all components (`type=any`) are extended with the `logger-controller` interface defined in the `logger-itfs` frame add-on. This interface is bound to the `logFactory` interface of the `logger` control component.

An *interface selector* selects one or more external interfaces (i.e., interfaces of the host component) based on their name and type (either `functional`, `control`, or `any`). The specified extensions are applied to each of the interfaces selected; the extensions may include microcomponent instantiation and establishing bindings. A microcomponent specified in an interface selector is instantiated separately for each such interface. Within the context of an interface selector, a microcomponent may be inserted into the external interface's delegation chain. This insertion is controlled via the `flow` at-

tribute of the instantiation instruction. *Passthrough* flow inserts the microcomponent to the head of an existing delegation chain of the external interface, by rebinding the external interface to the delegate-server interface of the new microcomponent, and binds its delegate-client interface to the original head of the delegate chain. *Endpoint* flow establishes a new delegate chain, by binding the external interface to the delegate-server interface of the microcomponent. *Standalone* flow assumes all bindings will be established explicitly, and is the only flow permitted outside the context of an interface selector (for microcomponents instantiated directly in a component selector).

In our example, the dynamic microcomponent `logfwd` is instantiated for each external functional interface and inserted into its delegation chain (passthrough flow). For each such interface, also an instance of the `callLogger` microcomponent is created, and a binding is established between these components.

An aspect may specify *microcomponent bindings* (among two microcomponents or between a microcomponent and a control interface, `<binding>` element) and *component bindings* (between a control interface and a control component, `<component-binding>`). Both of them specify client and server side of the binding, each consisting of *instance name* and *interface name*. Instance name is either name of a control component, name of a microcomponent, or keyword `this` (when referring to a control interface of the host component). Instance name must be visible within the scope of the binding instruction. A microcomponent name is visible inside the selector directly enclosing its instance definition and in its nested selectors. The name of a control component is visible throughout its defining aspect.

Special rules apply to bindings to control interfaces (`this` keyword). Control interfaces are the only point of interaction among aspects: an aspect may not explicitly bind microcomponents to microcomponents instantiated by other aspects, but may only access control interfaces they declare — either to bind a microcomponent to the control interface, or to insert a microcomponent into the delegation chain associated with the control interface. Hence, a binding may via the `this` keyword reference a control interface not defined within the scope of the aspect.

3.2. Launch Configurations

After defining a collection of aspects, the remaining step is to select the aspects to be loaded and applied to the components forming an application (as well as to

```xml
<aspect-definition name="logging" >
  <frame-addon-definition name="logger-itfs" >
    <interface signature="LoggerFactory"
      role="client" name="logger-controller"/>
  </frame-addon-definition>
  <component name="logger"
    definition="logger-adl" />

  <microcomponent-definition
    name="callLogger" >
    <interface signature="InterceptorNotify"
      role="server" name="notify" />
    <interface signature="LoggerFactory"
      role="client" name="loggerFactory" />
    <interface signature="NameController"
      role="client" name="name" />
    <content class="LoggerInterceptor" />
  </microcomponent-definition>

  <microcomponent-definition
    name="notifyForwarder" >
    <interface signature="InterceptorNotify"
      role="client" name="notify" />
    <dynamic-interface role="delegateserver" />
    <dynamic-interface role="delegateclient" />
    <content
      generator="InterceptorNotifyGenerator"/>
  </microcomponent-definition>

  <select-component type="any" >

    <frame-addon definition="logger-itfs" />
    <component-binding client="this.logger-controller"
      server="logger.logFactory" />

    <select-interface name="*" type="functional">

      <microcomponent name="logFwd"
        definition="notifyForwarder"
        flow="passthrough" />
      <microcomponent name="logCalls"
        definition="callLogger" flow="standalone"/>

      <binding client="logCalls.loggerFactory"
        server="this.logger-controller" />
      <binding client="logCalls.name"
        server="this.name-controller" />
      <binding client="logFwd.notify"
        server="logCalls.notify" />

    </select-interface>
  </select-component>
</aspect-definition>
```

Figure 5. Logging aspect ADL specification.

```
<configuration>
  <aspect name="protocols" definition="...protocols"/>
  <aspect name="logging" definition="...logging"/>
    <apply-aspect name="protocols"/>
  </aspect>
  <application definition="examples.hello.Hello">
    <apply-aspect name="protocols"/>
    <apply-aspect name="logging">
      <param name="logger-key" value="auditlog"/>
      <target path="./server"/>
    </apply-aspect>
  </application>
</configuration>
```

Figure 6. Sample launch configuration.

control components instantiated by aspects). Such a selection is captured in a *launch configuration*. Figure 6 shows a sample launch configuration. The configuration loads two aspects, `protocols` and `logging`. The aspect `protocols` permits to associate a component with a behavior specification (a behavior protocol [16]). The `logging` aspect (shown in detail in Fig. 5) logs all calls on functional interfaces of the affected component. An aspect may be applied to another aspect, affecting the control components instantiated by the target aspect. In our example, the `protocols` aspect is applied to the `logging` aspect; consequently, the `logger` control component will feature a controller to associate it with a behavior specification. Note that the relation formed by applying an aspect to another aspect must be acyclic, in order to avoid the possibility of infinite recursion.

After declaring all the aspects to be loaded, the launch configuration eventually specifies the application to be started by referring to its ADL specification, and also specifies the aspects to be applied. When an aspect is applied either to the application or to another aspect (its control components), the aspect may either affect the whole component hierarchy, or target only a fraction of the component tree (to select the target components, we employ a notation based on the Ant FileSet syntax [1]). In our example, the `protocols` aspect is applied to all components, while the `logging` aspect is applied solely to the server subcomponent. The launch configuration may also provide parameters to be passed as key-value pairs to the `configureMc` method of each microcomponent of the aspect; the parameters are specified either systemwide, specific for each explicit application of the aspect (`apply-aspect` instruction), or possibly specific for each target. In Fig. 6, the parameter `logger-key` is provided for the aspect `logging`.

4. Evaluation

Evaluation. To evaluate the microcomponent model and the component-aspect framework proposed in this paper, we have reimplemented several existing controller objects as component aspects and tested the framework on existing Fractal applications. The configuration framework allows us to easily impose control features on selected components without altering the application architecture. This permits to postpone decisions on application configuration and management until deployment time, with very little overhead to carry out these decisions.

The key contribution of the Asbaco project compared to the solutions previously available is the seamless integration of controller extensions achieved via selective application of component aspects to the component hierarchy of an application. The other part of the contribution is the microcomponent model, which permits to capture the structure of the controller part of a component; this framework permits to verify the consistency of the controller configuration prior to launching the application. Neither of these was possible with the alternative approaches available for Julia [13], the reference implementation of the Fractal component model.

Considering the broad spectrum of Fractal implementations (see Sect. 5), the microcomponent model may not be suitable for all of them. As a microcomponent is in a simplified view an object with several services provided and required, the microcomponent model is applicable to Fractal implementations where the controller part consists of "small object-like elements"; there, Asbaco provides a way to capture structure of the controller part. Even for Fractal implementations where this assumption does not hold, the model provides a way to specify controller part extensions.

From the point of view of aspect oriented programming, the join-points in our project are method executions, pointcuts may select component instances and interfaces, and advices are reflected by microcomponents, control interfaces, and control components. An interesting feature of our model is, that while the general part of the point-cut selection is specified in the aspect, the part specific to the application hierarchy is specified separately in the launch configuration.

Implementation. In the Asbaco project, we have developed a prototype implementation of the microcomponent model and the component framework based on Julia. The key part of our implementation is the component factory, responsible for constructing a component — including the controller part. The factory has been integrated into Fractal by replacing the original implementation of the `GenericFactory` interface in

the bootstrap component. This permits to apply aspects also to components created directly via Fractal API calls on the bootstrap components, besides the preferred approach to create components using the Fractal ADL framework based on their ADL description. As our focus was on developing the microcomponent model and the aspect framework, we have refrained from reimplementing the existing Julia controllers; instead, we wrapped them as microcomponents.

Case study. To test the framework, we have constructed several aspects, arising from actual needs in our Fractal-related projects we have developed for our industrial partners. The aspects utilize all the features presented in Sections 2 and 3, including control components, client control interface, and interception on functional interfaces. We have developed the *logging* aspect already used as the illustrative example in Sect. 3. Among those aspects not presented in the previous sections, we have also experimented with the *protocols* aspect, which allows us to associate a component with a behavior specification (behavior protocol [16]) via a simple controller, and with the *runtime checking* aspect, which allows us to monitor the behavior of a component at runtime and to check whether it obeys its protocol. The runtime checking aspect instantiates a *checker backend* control component, and intercepts operation calls on functional interfaces to deliver event notifications to the checker backend component. To monitor the component's lifecycle (to check whether the component's protocol permits to stop at a particular point in its execution), the aspect also inserts a microcomponent into the delegation chain of the lifecycle controller. The aspects can be also weaved together (e.g., the logging aspect may be applied to the checker backend control component to monitor its behavior). We also aim to develop an aspect implementing the Java Management Extensions (JMX) [19] monitoring features developed in the Fractal JMX project [12]; here, the agent component is also a candidate for a control component.

5. Related Work

The approaches related to our work can be basically divided in two major groups — modeling component controllers and using aspects to address crosscutting concerns in components.

Modeling of component controllers is present in Julia [13] and AOKell [17] implementations of the Fractal specification [3]. Julia uses mixins to dynamically create the controller part of a component. The combination of mixins yielding different controller configurations is described in a special file loaded during Julia start-up. Compared to our approach, extending a controller in Ju-

lia is difficult, quite error-prone, and requires a deep insight into Julia internals. Moreover, it is not possible to create new controller configurations at runtime.

AOKell on the other hand uses AspectJ [9] to weave the controller part with the content. Each application in AOKell comes with an AspectJ aspect which assigns a controller configuration to every component. Thanks to the use of aspects, extending a controller in AOKell is easier than in Julia. However, new controller configurations still cannot be created at runtime. Moreover, in AOKell, a controller definition is associated with a component type (as opposed to a component instance); thus, it is not possible to assign different controller configurations to different instances of the same component (which disallows us for example to log the activity only of one particular component instance). An interesting fact about AOKell is that the implementation of the three standard controller configurations (flat, primitive, and composite) is created using a dedicated Fractal-based component model; however, this component model is used only as a tool for generating the controller part implementation and has not been explicitly documented as a component model for capturing the controller part structure. The models for the three controller configurations are hardwired in the implementation giving no possibility to be easily modified or reused by an extension developer.

Important features of our work (not present in either of these approaches) are the concepts of client control interfaces and control components. Control components allow us to introduce new container-wide logic; client control interfaces provide us with a systematic solution to express that a controller requires a certain control component.

Regarding the use of aspects to address crosscutting concerns in components, there are a few approaches aiming at this task. *JAsCo* [20] uses special weaving connectors to associate component gates with aspect beans (which contain the advice and part of the pointcut specification). JAsCo has been implemented for JavaBeans and .NET. *FAC* [14] augments every component with an aspect controller to which advices are registered. The aspect controller intercepts the component's client and server interfaces and calls advices for which a pointcut matches. The framework puts focus on interception of functional interfaces and does not provide means to introduce new control interfaces or to alter the controller's behavior. The aspect components in this approach are compatible with AOP Alliance API, which is an open source initiative to define a common API for AOP frameworks. *FractalAOP* [6] is similar to FAC, however, the weaving logic is separated to a special weaving component which intercepts calls on

the original component via a special Execution Controller. Advices are represented using Advice Components. Please note that Execution Controller is a client control interface, however, [6] does not elaborate it as a generic concept.

As the aspect weaving was not the primary goal of our work, these approaches are more mature providing a richer selection of pointcuts compared to our work. On the other hand, they rely on modifying an application architecture by introducing the advice components. In our work we do not require this (the logic is hidden in the controller part of a component), which in our opinion yields a more systematic solution.

6. Conclusion & Future Work

In this paper, we have presented a microcomponent model to capture the structure of the controller part of a software component. Based on the microcomponent model, we have presented a framework for extending the controller part via component aspects. With the framework, different controller extensions can be seamlessly integrated by applying a selection of aspects to the application component hierarchy. While the results have been demonstrated on the Fractal component model, they are applicable also to other component models featuring explicit controller part and control interfaces. In the Asbaco project, we have developed a prototype implementation of both the microcomponent model and the component aspect framework; the implementation is based on Julia [13], the reference implementation of the Fractal component model [2].

Future work. One of the key goals in our future work is to investigate further needs in interception on functional interfaces, and to possibly introduce additional predefined interfaces. To extend the flexibility in interception, we consider introducing parameterized microcomponent templates to model interface type adaptations [7].

We plan to extend our prototype implementation also to other implementations of the Fractal component model; we would also like to explore the options to create a universal implementation independent of the particular Fractal implementation used. Further, we aim to extend the flexibility in selecting the target components in applying an aspect; for this, we consider employing the FPath language [4].

References

[1] Apache Software Foundation: *Apache Ant User Manual*, version 1.6.5, http://ant.apache.org/manual/, Jul 2005.

[2] Bruneton, E., Coupaye, T., Leclerc, M., Quema, V., Stefani, J-B.: *An Open Component Model and Its Support in Java*, Proceedings of CBSE 2004, May 24-25, 2004, Edinburgh, UK, LNCS 3054, Springer, 2004

[3] Bruneton, E., Coupaye, T., Stefani, J.B.: *The Fractal Component Model*, Draft 2.0-3, Feb. 5, 2004, http://fractal.objectweb.org/specification/

[4] David, P.-C., *Développement de composants Fractal adaptatifs: un langage dédié à l'aspect d'adaptation*, PhD Thesis, École des Mines de Nantes and Université de Nantes, Jul 2005

[5] Ecma International: *Common Language Infrastructure (CLI)*, 2nd edition, Dec 2002

[6] Fakih, H., Bouraqadi, N., Duchien, L.: *Aspects and Software Components: a case study of the Fractal Component Model*, WAOSD 2004, Beijing, China, Sep. 2004

[7] Galik, O., Bures, T.,: *Generating Connectors for Heterogeneous Deployment*, accepted for publication in proceedings of SEM 2005, Lisbon, Portugal, Sep 2005

[8] Jacob, J.: *The OMEGA Component Model*, Electr. Notes Theor. Comput. Sci. 101: 25-49, Nov. 2004

[9] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: *An Overview of AspectJ*, In Proceedings of ECOOP 2001, June 18-22, 2001, Budapest, Hungary, LNCS 2072, Springer, 2001

[10] Object Management Group (OMG): *CORBA Component Model*, v3.0, formal/02-06-65, http://www.omg.org/

[11] ObjectWeb: *Fractal ADL Documentation*, http://fractal.objectweb.org/current/doc/javadoc/fractal-adl/

[12] ObjectWeb: *FractalJMX Documentation*, http://fractal.objectweb.org/current/doc/javadoc/fractal-jmx/

[13] ObjectWeb: *Julia Documentation*, http://fractal.objectweb.org/current/doc/javadoc/julia/

[14] Pessemier, N., Seinturier, L., Duchien, L.: *Components, ADL and AOP: Towards a Common Approach*, In Workshop ECOOP Reflection, AOP and Meta-Data for Software Evolution (RAM-SE04), Jun 2004

[15] Plasil, F., Balek, D., Janecek, R.: *SOFA/DCUP Architecture for Component Trading and Dynamic Updating*, In Proceedings of ICCDS '98, Annapolis, IEEE CS, 1998

[16] Plasil, F., Visnovsky, S.: *Behavior Protocols for Software Components*, IEEE Trans. Software Eng. 28(11), 2002

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

[18] Sun Microsystems, Inc.: *Enterprise JavaBeans Specification*, Version 2.1, Nov 2003.

[19] Sun Microsystems, Inc.: *Java Management Extensions Instrumentation and Agent Specification, v1.2*, Oct 2002

[20] Suvée, D., Vanderperren, W.: *JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development*, In Proceedings of AOSD 2003, March 17-21, 2003, Boston, MA, USA, ACM, 2003