AOCI: An Aspect-Oriented Component Infrastructure

Guido Söldner  
Dept. of Comp. Sciences  
Informatik 4  
University of Erlangen-Nürnberg  
Germany  
guido.soeldner@cs.fau.de

Rüdiger Kapitza  
Dept. Of Comp. Sciences  
Informatik 4  
University of Erlangen-Nürnberg  
Germany  
rrkapitz@cs.fau.de

Abstract
Component-based middleware provides already mature support for non-functional properties (e.g. load balancing, transaction support or persistence) offered by the component container. However, to support context-aware applications enabling a flexible dynamic reconfiguration and adaptation, further techniques are required.

This paper proposes an Aspect-Oriented Component Infrastructure (AOCI) that allows the dynamic weaving according to application and environment demands while preserving component integrity. This is achieved by offering a greybox component concept exporting potential points for extension together with an ontology-based metadata description. In contrary to related systems the proposed solution promises a better support for runtime adaptation and eases the development of adaptable components.

Introduction & Goals
Software components [17] are usually considered as black boxes that offer specific services provided by interfaces. Furthermore, components can be easily combined with each other, thereby forming complex applications. However, to support component-based and context-aware applications enabling dynamic reconfiguration and adaptation, the component model and the infrastructure should further ease the development. To tackle these issues, recent systems [14,12,4] support the use of Aspect-Oriented Programming (AOP) [13]. AOP aids the programmer in the separation of concerns, specifically cross-cutting concerns, as an advance in modularization.

A crosscutting functionality represents code that cannot be located in one file but is scattered throughout many different files of an application. The concept of aspects improves modularity by locating these scattered elements in one place. Implementing an aspect consists of providing advice code and a pointcut. While the advice code defines the behavior of an aspect, the pointcut specifies a set of joinpoints where this behavior is to be applied in the application.

In this paper, we explore how to enhance Enterprise Java Beans (EJB) [16] by means of AOP and ontology-based metadata, thus enabling to provide components that can be dynamically adapted at runtime by the environment.

Traditionally, black box components and AOP do not match together, as components inherently forbid changing their implementation or otherwise loose major benefits like encapsulation or information hiding. In the past, several systems [15,11] have been proposed to overcome this issue by exporting additional points for adaptation. However, they do not sufficiently address scenarios, where dynamic reconfiguration and context-awareness is demanded. We propose AOCI, a context-aware aspect-oriented component infrastructure, which is built upon EJB. Developers using AOCI have the possibility to export potential points for extension matching extended
pointcuts of a component for use with AOP, while enforcing encapsulation and preserving the concept of a black box to a certain degree. The extended pointcuts represent the developer’s domain-specific knowledge of the component, which is expressed by means of the Resource Description Framework (RDF) [9]. Thus, our system supports a greybox component [20] approach, allowing a dynamic adaptation of components at runtime on behalf of the exported pointcut information together with additionally attached metadata. We call this combination a greybox annotation. Based on the provided metadata and environment specific rules the infrastructure is able to integrate non-functional properties, provided by advices, during run-time. This is achieved by dynamically weaving the program elements, which match the rule configuration of the context, together with the component.

Compared to standard component-based solutions [10] we see several advantages. As our solution is based on AOP, components can separate business logic from non-functional properties and need not to address issues related to dynamic adaptation. Therefore, neither interfaces have to be implemented, nor is a custom code generation tool needed. This reduces significantly the complexity of developing components. Additionally, adaptations can happen without the explicit knowledge of the source code, as the component developer provided the relevant information. Finally, our solution is compatible with every container, which supports the EJB 3.0 specification.

The remainder of this paper is organized as follows. Section 2 gives an example scenario as a motivation for our architecture. In Section 3 we describe the proposed solutions along with the technologies which are needed for the implementation. An overview of related work can be found in Section 4, followed by the conclusion in Section 5.

**Motivation**
In our scenario, an instant messaging software for a company is deployed within the AOCI component infrastructure. At first, the plain instant messaging component is used without applying any constraints. Clients can dynamically discover the service, log on with their credentials and use it. After a certain time, the administrator of the software discovers, that conversations between users are sniffed from time to time. Therefore, it is decided that all the messages sent by the instant messaging software should be encrypted. This is possible, because the component developer has exported the program elements where adaptation can happen, together with the metadata offering information where security related issues could be applied. The administrator changes the rules of the AOCI-infrastructure, defining that the network traffic is secured by using the Secure Sockets Layer (SSL). Afterwards, the infrastructure intercepts all the calls of the component to a plain socket and replaces the socket by a secure socket (e.g., java.net.ssl.Socket) thus enabling a secure communication. After a while, the executive board of the company decides that no private conversations may be held within the company. To guarantee this, all employees at the company are informed that the communication is being logged within the company. Again the administrator changes the rules to apply logging for each message being sent within the company. If clients use the instant messenger at home, the container will recognize the context switch and turn logging off.

As outlined, AOCI-aware components can dynamically deployed without knowing in advance which non-functional constraints are demanded at a specific target location. The application developer concentrates only on the business logic, while the constraints are imposed by execution context at runtime.

**Proposed Solution**
Our AOCI architecture proposal is based on EJB. Therefore we suggest an AOP implementation that is tightly integrated into EJB and allows dynamic weaving of aspects. For our prototype implementation, we use the JBOSS [7] application server as it already provides basic support for AOP, allows the dynamic weaving of aspects and is available as Open Source. Figure 4 shows the
overall architecture of the AOCI infrastructure. The main components are the deployed component with its greybox annotations, the rules, the Code Repository, which stores the non-functional advices, and the core component provided by the AOCI-adapter. In the following we will shortly discuss the underlying technology RDF, followed by different architectural elements of the solution.

Figure 4: The AOCI architecture

**RDF-Ontology**

In our approach the developers provide means for later adaptability by exporting metadata about the component and thus providing the component as a greybox. This information allows the infrastructure to dynamically weave code to the component. To enable the dynamic binding between an extended component with context-sensitive code from the infrastructure we use RDF, which is a family of World Wide Web Consortium (W3C) specifications originally designed as a metadata model using XML. The underlying structure of a RDF expression is a triple in the form of subject-predicate-object. A collection of RDF statements intrinsically represents a labeled, directed pseudo-graph. The benefits of RDF are its simple data model and the ability to model disparate, abstract concepts. Using this technology we will be able to define an underlying ontology that represents a set of concepts within a domain and the relationships between those concepts. The metadata in the rules and in the components corresponds to the nodes in the ontology. This allows the infrastructure to have a common understanding of the greybox components, the rules and the advices, which are stored in the Code Repository, thereby enhancing the infrastructure to dynamically adapt the software in a context-sensitive environment. To outline the proposed mechanism we present a partial definition of a domain-specific taxonomy targeting security as an example for a non-functional requirement (Figure 1.).

**Extended Pointcuts**

There are two ways to advise elements of a component. Either a central XML-
configuration file, where all possible modifications can be specified, is used or Java annotations as a hint for the infrastructure. In the following, the annotation-based approach will be outlined (Figure 2), where the function `sendMessage` is marked.

```java
public class InstantMessenger {
    @AOCI(SecurityMechanism.SecurityProtocol.NetSecurity)
    @AOCI(SecurityMechanism.Auditing)
    public void sendMessage(String text, String receiver){ ... }
}
```

With this information the AOCI infrastructure knows, that this function is a place, where encryption like SSL or auditing can be applied.

To determine how the component should be dynamically adapted, defining the extended pointcuts is not enough. The infrastructure must know which code must be added or replaced. Therefore rules are needed on behalf the infrastructure can match existing extended pointcuts with possible code fragments. The notion of the rules is very similar to the extended pointcuts and also based on RDF. The following snipped (see Figure 3) determines that communication must be encrypted with the help of SSL.

```xml
<xml-version='1.0' encoding='UTF-8'>
<AOCID>
    <entry>
        <value> SSL </value>
    </entry>
    ...
</AOCID>
```

Figure 1: Security ontology

Figure 2: Annotation-based extended sample pointcut

Figure 3: Example rule defining a security constraint
**AOCI Adapter**

The AOCI-Adapter is the part of the framework which analyzes and adapts the components. It has to fulfill several steps (Figure 5). First of all, the extended pointcuts of the greybox are read. It then checks, if the syntax of the extended pointcuts is right and corresponds to a node in the RDF tree. On success an appropriate implementation is searched. If no such is found, there are different possibilities how to proceed. Either the infrastructure can throw an error, can forget about the missing implementations and just proceed or can take a standard implementation for the extended pointcut. The infrastructure then dynamically builds an AOP fragment with the implementation. Within the last step the point of the extended pointcut is taken as a hook and the infrastructure weaves the newly created fragment with the current implementation.

![AOCI-Adapter Diagram](image)

**Figure 5: The AOCI-Adapter**

While RDF is a flexible and extensible way to represent information about the resources on a platform, there is still need for a standardized way to retrieve these data. Therefore we propose the use of SPARQL [19], which consists of the syntax and semantics for asking and answering against RDF graphs. Furthermore it is possible to query by triple patterns, conjunctions, disjunctions, and optional patterns. Constraining queries by source RDF graph and extensible value testing is supported as well as the possibility that results of SPARQL queries can be ordered, limited and offset in number. As a framework for RDF and SPARQL we propose Jena [8], which allows to dynamically compose and evaluate these queries.

**Related Work**

Recently, several infrastructures targeting the adaptation of component-based software have been developed. Most of them concentrated on providing mechanisms to adapt components, only few use the benefits of AOP and make use of an ontology-based metadata description.

Pessemier et al. [11] show that AOP can safely be supported by Component-Oriented-Programming (COP). In their paper, they explain which problems they have encountered when using AOP with components. To overcome these issues they have introduced the greybox, which marks a compromise between modularity and openness. However there is no support for building context-aware applications using the support of an ontology.

In their work, Rho et al. [12] describe that services in a Service Oriented Architecture (SOA) are general in their nature, which allows them to fit the needs of as many clients as possible. On the other hand, these systems cannot address each single requirement. Therefore they introduced the concept of context-sensitive service aspects which can be changed dynamically at runtime. Their framework is called Ditrios, which is based on OSGI, allowing a context-sensitive weaving. Although there is support for AOP, there is no concept like a greybox and there is no support for components.

Duclos et al. [4] outline that both, component-based applications and AOP address the same “separation of concerns” issue. While containers are only proposing a fixed set of services, AOP
is working at object level. Additionally they present an approach to get both the advantages of AOP and component based systems. Furthermore, they provide a simple language for using aspects on applications. However, the software allows the adaption of components, but no context-awareness is supported.

The Rainbow framework [5] provides a software architecture and infrastructure to support the self-adaption of software systems. To fulfill these issues, the architecture uses an abstract model to monitor the system’s runtime properties, and provide means to react upon events and adapt the software. However Rainbow neither addresses the concept of greyboxes nor AOP. Additionally no ontology is supported.

Another component-based approach is presented in the K-Component-Model [3]. The work is based on architectural reflection. To guarantee integrity and a safe dynamic software evolution a graph transformation is proposed. Furthermore adaption-specific code is separated from functional code by encapsulating it in reflective programs called adaption contracts. K-Component does not address AOP.

CAMidO [1] is an architecture for supporting development and execution of context-aware component applications. It provides an ontology metamodel, which facilitates the description of contextinformation. Additionally the platform provides the possibility to write interpretation and adaption rules, which are used by the CAMidO-compiler. However, there is no support for either AOP or ontology.

Toa Gu et al. present a Service Oriented Context-Aware Middleware (SOCAM) [6], which provides efficient support for acquiring, discovering, interpreting and accessing various contexts to build context-aware services. A formal context model is introduced, which is based on ontology using Web ontology Language to address issues including semantic representation, context reasoning, context classification and dependency. The infrastructure has no support for components.

Mügge et. al. [15] present a solution for an adaptive middleware in the area of ubiquitous computing and describe a new approach combining aspect-oriented programming with structural metadata. However, they do not address context-aware applications.

Table 1 shows an overview of the capabilities of the presented related work. Our approach is the only one supporting context-awareness, AOP, components, the greybox concept and semantic ontology. As we want to weave different aspects and components together, which do not know each other until run-time, we attach importance to the fact that the infrastructure supports both the concept of a greybox and is capable to deal with an ontology.

<table>
<thead>
<tr>
<th>System</th>
<th>greybox</th>
<th>AOP</th>
<th>Context-aware</th>
<th>Component-based</th>
<th>Adaptive</th>
<th>Ontology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pessemier et. al</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Rho et. al.</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Duclos et al.</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Rainbow</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>K-Component</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Camido</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SOCAM</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Mügge et al.</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>AOCI</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 1: Overview of related approaches
Conclusion
In this paper, we presented an architecture based on the EJB component model, which allows the dynamic and automated adaptation of components. Because this usually breaks the modularity of components, we have introduced the greybox-concept, which allows annotating certain program elements to be adapted by the infrastructure.

At the moment, we are working on a prototype of the described infrastructure. As already mentioned, it is based on JBoss as an EJB container and uses the AOP support provided by JBoss. In the near future we will define an ontology, which contains preferably most functionality needed for effective adapting components. This will demand for extended case studies to figure out the right balance between generality and practical usability of the terms defined by the ontology. Additionally, we consider enhancing our infrastructure by using OSGi [18] for flexible loading and unloading of aspects as well as components.

References


