AOCI: Weaving Components in a Distributed Environment

Guido Söldner, Sven Schober, Wolfgang Schröder-Preikschat, and Rüdiger Kapitza

Dept. of Comp. Sciences 4, University of Erlangen-Nürnberg, Germany
{soeldner,schober,wosch,rrkapitz}@cs.fau.de

Abstract. Mobile and embedded devices like PDAs, mobile phones, and all kinds of consumer hardware populate the world we live in. Despite the vision of ubiquitous computing and its idea of spontaneous interaction among these devices more than fifteen years ago, most of them are still isolated and restricted in their interaction capabilities. One reason for this limitation is the poor support for dynamic adaptation and evolution of software in distributed environments.
This paper proposes AOCI, an Aspect-Oriented Component Infrastructure that takes the core ideas of AOP, the separation of concerns and system modularization to make them more adaptable and evolvable, to the domain of component systems. Components are usually considered as black boxes that can be combined to a complex system using their outer interfaces. In the context of our infrastructure, components export possible adaptation points, which are enriched by ontological information. This enables the application of AOP techniques without detailed knowledge about the component's internals, enabling dynamic and distributed adaptation.
Our prototype is based on OSGi and provides a complete infrastructure to weave local as well as remote components. We demonstrate the feasibility of our approach by adapting the RUBiS infrastructure (a web-based bidding system) to support dynamic user-centric error detection.

1 Introduction

More than fifteen years ago, Mark Weiser introduced the term of ubiquitous computing [1], which proposes a post-desktop model of human-computer interaction where information processing is integrated into everyday objects and activities. As opposed to the desktop paradigm, in which a single user engages a single device for a specialized purpose, ubiquitous computing has the vision of small, inexpensive, robust devices, either mobile or embedded in almost any type of object imaginable, including cars, tools, appliances, and various consumer goods – all communicating through increasingly interconnected networks. However, the services these objects will deliver, still have to be adapted to the person, the time, or generally spoken to the context of their use [2]. In such a world, where everything is connected to everything, Ferscha et al. [3] propose a *spontaneous interaction* model, where devices start to interact with each other once they have entered the physical proximity to each other. This behavior is called a *digital aura* [3]. This in turn means that dynamic adaptation of services in such an aura is of great relevance. Some adaptations can be anticipated, but as the whole range of applications within an interaction is difficult to estimate, some of them are not. Thus, there is a need for appropriate software engineering techniques and methodologies enabling systems to handle unexpected situations - or at least to improve their features, enabling spontaneous interaction.

1.1 Aspect-Oriented Programming and Components: Do They Match?

Aspect-oriented programming (AOP) [4] has proven to be a technology that can support software evolution and adaptability. AOP aids the programmer in the separation of concerns, specifically cross-cutting concerns, as an advance in modularization. However, the techniques of aspect-oriented software development that help to improve the modularity of software can restrict the evolvability of that software. The *obliviousness* [5] property of aspect orientation states that aspects are not invoked explicitly, but instead implicitly [6]. However, a developer has to be aware of which advice are applied to his source code to be sure that the aspect does not negatively impact his original intent. In consequence, applying AOP to code written by someone else (especially when the source code is not available) is risky. This is even more the case when using components [7], which usually offer a black box behavior. Thus, at first sight, components and AOP do not match, as components inherently forbid changing their implementation or otherwise lose major benefits like encapsulation or information hiding.

1.2 AOCI: An Aspect-Oriented Component Infrastructure

Based on our preliminary research work [8], we propose a context-aware *Aspect-Oriented Component Infrastructure* (AOCI), which is built upon OSGi [9]. AOCI seperates the identification of possible points for adaptations (pointcuts at source code level) from programming of advice (code that will be woven) by embedding a conceptual metamodel based on an ontology, representing a set of concepts within a domain and the relationships between those concepts. Component developers export potential points for adaptation by attaching ontology-based meta information to the component source code. Thereby, developers do not only restrict the places where adaptation can happen but also provide information on what kind of adaptations might be suitable at these points. We call those extended pointcuts *ontology-based pointcuts*. Aspect-writers describe where advice code should be applied by means of the metamodel. To match the ontology-based pointcuts and the aspects, AOCI uses the Resource Description Framework (RDF) [10] to express the metadata and to enable easy processing of the ontology. Thus, our system supports a *greybox* component [11] approach, allowing adaptation of components at load time by means of the ontology-based pointcuts.

Our prototype implementation is based on the Equinox OSGi platform and utilizes Equinox Aspects [12], which provides basic AOP support for OSGi. At its core, Equinox Aspects integrates the AspectJ [13] load time weaving features enabling the dedicated weaving of OSGi components. In the context of our prototype, we use this support to provide the AOCI-aware components by integrating the proposed ontology-based pointcuts. We show that our approach performs well in a real-world application, the RUBiS bidding system, and outline a use case scenario in a ubiquitous setting.

1.3 Contribution

Compared to common component systems such as plain OSGi or the CORBA component model [14], we see several advantages. As our solution is based on AOP, components can separate functional from non-functional code, and issues related to adaptation do not need to be addressed inside the component. Therefore, neither additonal interfaces have to be implemented, nor is a custom code generation tool needed. This significantly reduces the complexity of developing components and runtime overhead if an adaptation is not necessary. Component developers use their expertise to specify possible points of adaptation without the need to be aware of the actual advice code that might be integrated. Aspect developers can develop advice as usual and specify a pointcut at the metamodel level. Thus, in the ideal case, advice and components can be implemented independent of each other and aspects can be applied at any time. This aids the spontaneous interaction in a ubiquitous setting together with the support of applying aspects at remote locations. The use of ontologies as metamodel bears some advantages. Firstly, they can be used to describe the functional and non-functional concerns of components. Secondly, they allow a convenient and powerful querying capability. In the following sections we explain AOCI, which enables ubiquitious computing and show a proof of concept for this infrastructure.

The remainder of this paper is organized as follows. Section 2 presents the core idea of an AOCI-aware component. Section 3 outlines the technical background. In Section 4, we describe the architecture of the prototype. Section 5 shows our sample proof of concept application. An overview of related work can be found in Section 6, followed by a concluding section.

2 Basic Concept

Component adaptation needs to be taken into account as a design principle when it is not possible at design time to anticipate every use case the software will be involved in during its lifetime. This especially applies to distributed and ubiquitous computing where users and services are dynamic in nature and where the execution context usually demands some form of adaptation. In this section we address the challenge to apply aspect-oriented techniques on components to enable dynamic adaptation.

Components are usually considered as black boxes, which offer specific services provided by interfaces and hide their implementation. This is especially the case when using third-party components that are only available in binary form or when components reside at distant locations. Thus, it is almost impossible to write aspects for adapting traditional black box components. Even if the component code is available, the expressive power of AOP can lead to some unwanted side effects.

AOCI addresses these issues by constraining the power of AOP and introduces *ontology-based pointcuts*. An ontology-based pointcut represents an extended pointcut, hence possible points for adaptation, and is based on a conceptual metamodel. Therefore, in AOCI, aspects are decoupled from the structure of the base program. As a result, the definition of their application is no longer defined in terms of the source code and low-level implementation details, but is expressed based on properties of the metamodel. The metamodel is deployed in an additional abstraction layer, which is layered over the implementation and serves as a contract between the source code of the component and the aspect. Fig. 1 illustrates how ontology-based pointcuts, annotated by the developer of the component, and aspects play together to realize the decoupling between aspects and source-code. The application developer is responsible for the components and defines the places where later adaptation is possible in terms of the ontology. The metamodel is represented as a separate layer and contains the ontology. The ontology is maintained by a domain expert and is global in scope or at least rich enough to cover the target domain. Hence, it forms a domain-based abstract view of the concepts. Aspect developers, however, can write aspects, which are automatically applied to the component by the infrastructure. To do so, aspect developers write their advice in conjunction with the ontology-based pointcuts, so forming the aspect. As a consequence, we decouple the base code from aspects by means of an intermediate abstraction layer. At the lower level the aspect developer still intercepts and adapts code of the woven component. As he has no access to the source code he has to depend on runtime information provided by the AOP engine to actually do the adaptation. However, the locations where his code is applied is well defined by the ontology based pointcut. See Section 4.5 for more information about the programming model.

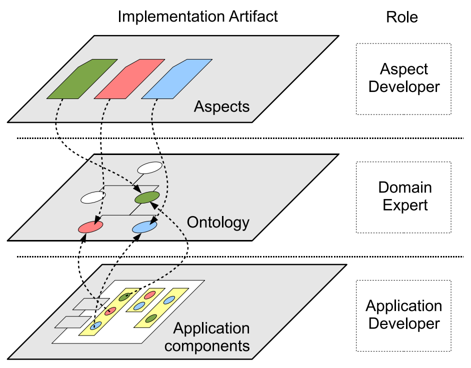

Fig. 1. Conceptual structure of an AOCI-aware component

3 Base Technologies

3.1 RDF Ontology

In our approach the developer provides the 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 at predefined places in the component. To enable the dynamic binding between a greybox component with advice, we use the Resource Description Framework (RDF), which is a family of World Wide Web Consortium (W3C) specifications originally designed as a metadata model using XML. The underlying structure of an 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. Based on RDF, we establish ontologies for the AOCI-infrastructure and its use cases. Ontologies define concepts in certain domains and the way these concepts interact, hence they provide a way to view the world and help to organize information. Furthermore, they facilitate interoperability, as they define a shared vocabulary and meanings with respect to other terms. To implement these ontologies, we used OWL (Web Ontology Language) [15], which is a RDF-based ontology markup language that enables context sharing and reasoning. In AOCI, we use OWL Full [16], a sublanguage of OWL, which contains the complete set of language constructs of OWL. 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 (see Fig. 2). The domain of security is subclassed with *Credential*, *Auditing*, and *Security Protocol* classes, which are either subclassed or have references to implementations of these concepts. Our infrastructure already utilizes such generic ontologies for non-functional properties to reason about potential adaptations within the components. However, it is possible to modify these ontologies or add custom ones for more specific use cases.

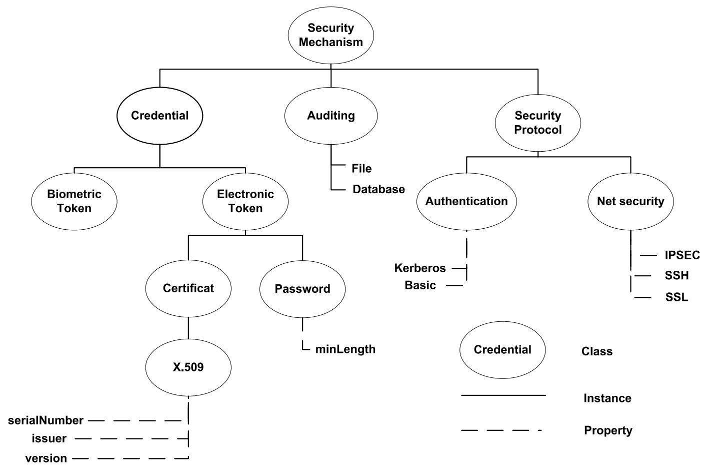

Fig. 2. Security ontology

While RDF is a flexible and extensible way to represent information about resources on a platform, there is still a need for a standardized way to retrieve and process these data, in order to realize the mapping between an ontology-based pointcut and an aspect. Therefore we use SPARQL [17], a query language with similarities to SQL, which has its own syntax and semantics for asking and answering against RDF graphs. Furthermore, it is possible to query by triple patterns, conjunctions, disjunctions, and optional patterns. As a programming framework for RDF and SPARQL we selected Jena [18], as it allows to dynamically compose and evaluate these queries.

3.2 OSGi

Our architecture is based on OSGi, which implements a complete and dynamic component model in the sense of service-oriented computing. In the OSGi terminology, components are called *bundles*. The infrastructure provides bundles with lifecycle management; they can be remotely installed, started, stopped, updated, and removed without restarting the framework.

Bundles act as services, which reduces the coupling of bundles by defining service interfaces. Therefore, if a bundle wants to access services of another bundle, the interfaces have to be published in the *Service Registry*. This registry can also be used to discover and bind services. Each bundle has a bundle manifest, which declares the identity of a component and its exports and imports in terms of code dependencies and services to other bundles. Based on these declarations, OSGi enforces collaboration constraints between bundles.

While there are a number of implementations of the OSGi specification, we decided to use Equinox [19], as it is a well-proven implementation and is very flexible for extensions. Equinox offers the possibility to use hooks regarding the bundle life cycle and the setup of class loaders. This provides the base to integrate load time weaving support as addressed in the following section.

3.3 Equinox Aspects

Equinox Aspects [12] uses the flexibility of the aforementioned hook architecture to support the integration of AspectJ's load time weaving features into Equinox. It has been implemented as OSGi lacks the support for AOP at the bundle level. This is needed to apply AOP in a controlled and component-aware way. Aspects are represented as bundles that can be applied on other bundles. Thus, it is possible to capture inter-bundle relationships through AOP. In addition, aspects can describe patterns of interaction between bundles. Consequently, the major benefits of AOP are available for OSGi based applications.

Within Equinox Aspects, there are two programming models: *Opt-In* and *Co-Opt*. The motivation behind these models is that developers can write their own aspects, use or customize those of a library, or even weave third party code. Therefore, the models help to preserve the developer's intent at runtime using metadata and to resolve introduced dependencies between the bundles. Using the opt-in model means that the component developer decides which aspects will be applied to his bundles. This opt-in model can be implemented by means of the existing manifest headers. On the other hand, the co-opt model allows to decide where aspect bundles should be applied. Equinox Aspects introduces several new bundle manifest headers to implement this functionality. Hence, using the co-opt model means an inversion of dependencies between the bundles, as the aspect bundle complements the application bundle.

Thus using Equinox Aspects AOP techniques can only be applied in the scope of aspect bundle, finer-grained mechanisms are not available. However, for AOCI, this does not mean a disadvantage, as we focus on components and greybox-concepts. On the implementation level, Equinox Aspects uses the hookable adapter architecture; this mechanism is used extensively within the class loading of the aspects. Once a bundle is to be loaded, the class loading is intercepted by Equinox Aspects, which in turn checks if adaptation is required. This is done by integrating AspectJ and its configuration files, especially `aop.xml`, which publishes the pointcuts, where the aspect bundle's advice can be applied. In the context of our implementation we utilize the co-opt model to weave aspects after a semantic match process has been performed (see Section 4).

3.4 R-OSGi

OSGi offers limited support for distribution. This is achieved by means of interfaces for interaction with Jini (for OSGi R3) or UPnP (for OSGi R3 + R4). However, Jini is not a lightweight implementation, as it is based on RMI, which in turn makes it less feasible on mobile devices, and UPnP does not support the richness of OSGi services within a distributed environment. Therefore, AOCI uses R-OSGi [20], a middleware platform that extends the standard OSGi specification to support distributed module management. R-OSGi allows multiple peers with local OSGi frameworks to interact in such a way that they behave like one large framework. On the implementation side, R-OSGi uses proxies, which are created on demand. AOCI is designed for use with mobile devices and spontaneous interactions are a key requirement. These devices do not have a preconfigured knowledge about their environment, instead they have to be context-aware and need to be adapted whenever interaction with other nodes occurs. R-OSGi addresses this requirement by providing basic support for service discovery. To communicate with other frameworks, R-OSGi uses the Service Location Protocol (SLP), which is described in RFC 2608 [21]. By using SLP, it is possible to announce services on a local network and dynamically discover them. Additionally, services can be described by attributes, they can be used to query for this service, using LDAP's query language. While the notion of services in SLP is very close to the ones in OSGi, however, the client cannot use the standard OSGI service interface to communicate with them; instead it has to use the `RemoteOSGiService` interface. Nevertheless, this does not represent a disadvantage as R-OSGi is encapsulated within the AOCI infrastructure.

4 Architecture

In this section, we present the prototype for an OSGi based component system supporting our ontology-based pointcut approach. The implementation is to be deployed on each participating host. A high-level architectural overview is shown in Fig. 3. Each of the depicted components is composed of one or more bundles that have to be deployed in an OSGi instance and provide their interfaces as services.

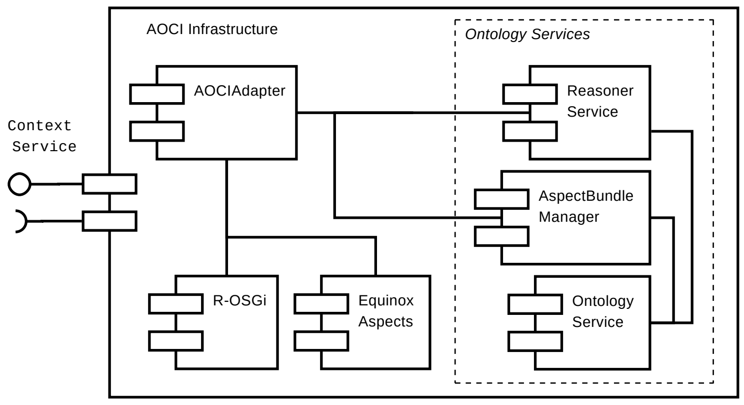

Fig. 3. Architecture of the AOCI infrastructure

Currently, our prototype offers two methods for remote interaction (see Fig. 4). The `evaluate` method is responsible for initiating the adaptation of bundles. This method receives a map of annotations (ontology-based pointcuts) and checks whether the host represented by the particular service instance has aspects-bundles matching these pointcuts. The details of this process are discussed in Section 4.4. Furthermore, the `getID`-method uniquely identifies an AOCI-host. In the following sections we will briefly discuss the subcomponents, which form the AOCI infrastructure followed by the description, how the adaptations of bundles take place using our ontology-based pointcut concept.

```
public interface ContextService {
  Map<AspectBundle, Set<String>>
    evaluate(Map<String, Set<Annotation>> bundleInformation);

  ContextID getID();
}
```

Fig. 4. Interface of the AOCI infrastructure component

4.1 AOCI Adapter

The AOCI adapter represents the part within the infrastructure that communicates with other hosts and manages the adaptation process. Therefore, it provides the support to discover other AOCI-aware peers and to exchange aspects between them. Depending on the scenario, the adapter either acts as a *context adapter* or a *server adapter*. Within a ubiquitous environment with many hosts, it is possible that a specific host offers some services to other hosts, which can be adapted based on the ontology-based pointcuts (server adapter), as well as it can provide aspects to other services and adapt them (context adapter).

Taking the role of a context adapter, an AOCI infrastructure provides the interface `ContextService`, thereby enabling a host to publish a set of aspect bundles, which can be applied to the remote host. To achieve this, the activator of this component registers an object implementing this interface with the local service registry. The service property `service.remote.registration` is set, causing R-OSGi to export this service. Thus, the context service is discoverable by other peers.

An AOCI infrastructure serving as a server adapter is responsible for weaving aspects received from other hosts. It registers a listener for peer context services within R-OSGi. On discovery a `FrameworkUpdater` thread is created, which initiates an update run, hence weaving the advice. This is explained in detail in Section 4.4.

4.2 Ontology Services

The ontology services, comprised of *Reasoner*, *Ontology Service* and the *Aspect Bundle Manager*, provide the ontology and its functionality for reasoning and maintaining it. The Ontology Service encapsulates the ontology model as a Jena OWL `OntModel` object. The ontology can be serialized as XML and can be updated at runtime. References to aspect bundles implementing concepts represented in the ontology model can be registered, deregistered and looked up using this service.

Based on this ontology, the Reasoner Service is responsible for mapping sets of annotations defined in the source code to sets of aspect bundle references. To achieve this, it queries the Ontology Service for each annotation if there is an aspect bundle implementing that concept, thus creating a mapping from annotations to aspect bundles. In our prototype, we use a basic matching algorithm. However, this can be enhanced to more complex algorithms that allow a more sophisticated reasoning (e.g., use of rules or logical operators).

The aspect bundle manager maintains the aspect bundles and acts like an aspect store. It saves references of the implementations within the ontology and offers functionality to register and deregister aspect bundles at runtime within AOCI.

4.3 R-OSGi Service

As previously noted, we use R-OSGi as part of our infrastructure to discover and use remote context services. The discovery of a remote AOCI context triggers the adaptation-process within our infrastructure. This can happen either manually by calling `RemoteOSGiService.connect(URI endpoint)`, or via a service discovery protocol like SLP. R-OSGi communicates across distributed services by generating and installing bundles containing dynamic proxies. As R-OSGi is not tightly integrated into the OSGi service registry on each host, AOCI uses a `GenericROSGiServiceTracker` to serve as a bridge between the two. For this purpose, it registers a `GenericDiscoveryListener<T>` with the registry. This listener implements the interface `ServiceDiscoveryListener`, which is called by R-OSGi when a new service has been discovered and made locally available. Our

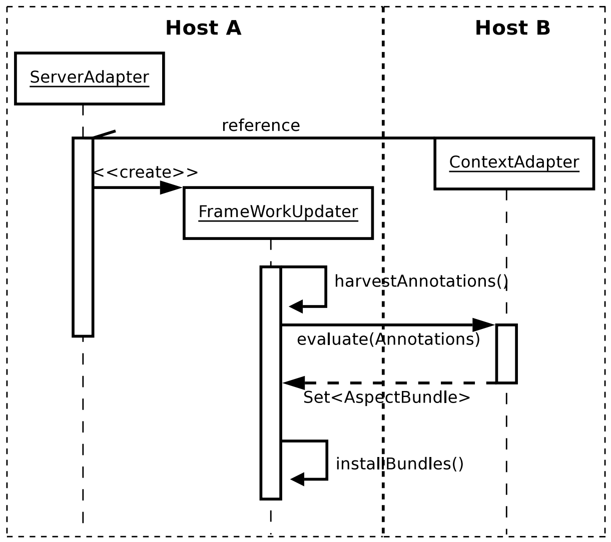

Fig. 5. Discovery, evaluation and adaption of aspects

generic listener can be specialized via the template parameter to look for any service interface. On discovery, it acquires the service and registers it with the local OSGi registry, thus making it transparently available to local bundles.

4.4 Adaptation Process

After presenting the structural overview of the AOCI infrastructure in the previous sections, we describe the dynamic interaction of the components using a sample adaptation process. During the interaction of two participating hosts, our infrastructure acts according to two distinct roles, as a *server adapter* (host A) and as a *context adapter* (host B). This is reflected in Fig. 5.

Once two nodes have discovered each other using the R-OSGi service, the server adapter starts a `FrameworkUpdater` thread. This thread is responsible for exchanging information about the hosts and their possible points for adaptation. Based on this information, the infrastructure decides which adaptions have to take place. Once running, it scans the currently installed application bundles of the local framework for AOCI annotations. This is done by the `AnnotationHarvester`, which loads each class from each bundle and uses reflection to find fitting code elements. This piece of information can be cached for further use (the current prototype does not provide caching support yet). In the next step, the resulting set of annotations is sent to the context service for evaluation. This process returns a suitable set of aspect bundles. These bundles are then installed in the local framework. The manifest headers trigger Equinox Aspects to adapt the `Require-Bundle` headers in the manifests of the application bundles by adding the symbolic names of the relevant aspect bundles. On each affected application bundle an *update*-operation is performed, which causes all dependencies to be re-evaluated. That way the contents of the aspect bundles become ”visible” to the application bundles class loaders. As Equinox Aspects is hooked into the OSGi frameworks class loading, each class is processed by a weaving service, which uses the AspectJ load time weaving facilities to weave the class if an `aop.xml` file has been found.

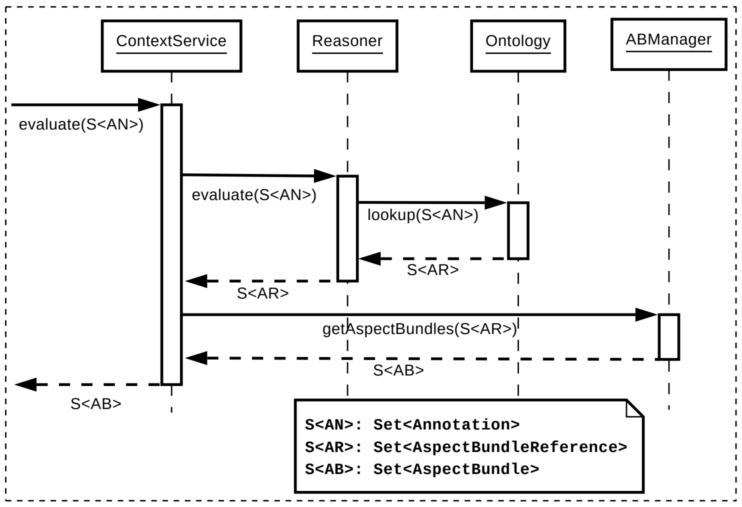

Fig. 6. Processing the context to select suitable aspects

The component interaction on the side of the context adapter is depicted in Fig. 6. The context adapter receives the set of annotations as an argument to its `evaluate`-method. The evaluation is then delegated to the Reasoner Service, where each annotation is looked up within the local ontology service returning a set of aspect bundle references. The reasoner then consults the aspect bundle manager to map the references to objects implementing `aoci.util.aspectbundle.api.AspectBundle`. These objects representing the aspect bundles selected by the reasoner are customized by the context service to reflect the specific situation in the receiver framework. This includes adapting their manifest headers so that the `Eclipse-SupplementBundle` entries contain the correct application bundles. The resulting set of aspect bundles is then returned to the caller of the `ContextService.evaluate` method, the receiving side.

4.5 Development Model

The use of ontology-based pointcuts and aspects within the AOCI infrastructure is fairly easy. The application developer simply annotates the code for use with AOCI by means of annotations. In our prototype, we have implemented the ontology-based pointcuts by means of Java annotations. In Fig. 7 the approach is outlined. The method `doPost` is annotated for the application of logging aspects. The mapping of the annotations to the ontology description is quite simple. While ontologies are represented as hierarchical trees, this structure is also included within the annotations, whereas the corresponding nodes in the tree are seperated with points. Additional properties for nodes can be set with key-value pairs within round brackets.

```
@AOCI.Logging
public void doPost(HttpServlet request, HttpServletResponse
    response) throws IOException, ServletException
{...}
```

Fig. 7. Code annotated for use with AOCI

The aspect developer has the full power of the AspectJ advice constructs. As a consequence, this permits him to access the language constructs (e.g., methods, fields, constructors) annotated in the source code. However, for some concepts within the ontology this is not enough information to implement meaningful aspects, especially if parts of the advice are functionally dependent on source code (e.g., when annotating code for encryption, the component developer has to publish information what can be encrypted). To solve this issue, the component developer can make use of OWL class properties to provide this additional information to the aspect developer. The information which properties are available for use is stored within the ontology. Furthermore, in AOCI all aspects are only applied within the context of the user, meaning that the weaving of code does not influence other services running in contexts also using this component.

4.6 Security Considerations

Dynamic loading of code is risky as the code could be malicious. However, due to the fact that OSGi is Java-based, it provides standard Java security mechanisms such as sandboxing [22]. In addition, OSGi offers support for digital code signatures [23] and a special optional *Permission Admin* service for a fine-grained security-related framework configuration since release 4 [9]. So far, off-the-shelf Java runtime environments provide no support for denial-of-service attacks where malicious code consumes as much resources as possible. However, this problem can be solved once the isolate concept [24] is available in a production JVM.

5 Evaluation

For the evaluation we conducted experiments with RUBiS (Rice University Bidding System) [25]. This web-based bidding system was built to benchmark common middleware infrastructures under realistic load situations. Thus it can be configured using different middleware stacks such as EJB, Java servlets, or a plain CGI variant. The information about items for sale, bids, and users is stored in a database. In the context of our work we selected the Java servlet variant, as it can easily be ported to OSGi, which already specifies an HTTP service that is provided by most of the OSGi platforms. Within the evaluation, we used a Jetty based (a java servlet container) service implementation and a MySQL database. The servlets representing the core of RUBiS had to be packaged as a bundle. This bundle covers about 5000 lines of code and has been extended and annotated by 17 possible points of extension. These annotations represent the opportunities to adapt the system to user and administrator demands, hence forming the ontology-based pointcuts. For the evaluation, we have annotated two servlets, the AboutMe and the SearchsItemsInCategory servlet. Within RUBiS, there are over 20 servlets.

In the context of our experiments, we envision the scenario that a certain user has a specific problem; for example, due to a configuration problem or unexpected inputs. In the common case, a user would get an error page indicating an unsuccessful request. This page might include some information, but in general, this is not enough to detect the cause of the bug. Thus, the user sends a bug report to the site administration. The site administrator has now the duty to search for the error based on the available information. Often it can be cumbersome and time consuming to detect the cause for the error if it is possible at all.

Using our infrastructure, this process can be eased. The user is confronted with the error and wants to know the reason to get it fixed in a timely fashion. She discovers that the RUBiS system supports AOCI and that it is possible to get logging information. Fig. 8 shows the relevant parts of the ontology that are used by our sample application. Note that parts of the ontology represent non-functional classes, while others show functional elements. As RUBiS implements a selling platform, there are already domain-specific elements, which describe the most important services offered by common selling platforms. Instances of classes can have relationships to other classes, which means in our case, that RUBiS supports logging. So the user sends over a logging aspect bundle that is woven at the server side. She re-issues the error-prone request and gets logging information about the cause of the bug. This information gives her a basic information about the reasons and she is able to provide a valuable error report that helps the administrator identify the cause of the bug more rapidly. Of course, the execution of the user-supplied aspect has to be restricted to the request executions of the user. This can be achieved by evaluating user context from the current request. If the current user issuing a request matches the supplier of an aspect, it is executed, otherwise the request is processed without further interception.

The user-initiated injection of the logging aspect results in a service downtime as the affected bundles have to be reloaded. The integration resulted in a mean downtime of the service of 2.21 seconds, contrasted to a downtime of 0.42 seconds on average for a simple update and restart. The increased service unavailability is a consequence of the bytecode rewriting to integrate the aspect during the class-loading process.

We have benchmarked the plain RUBiS system and compared it with AOCI deployed and the applied logging aspect for one user. The results are shown

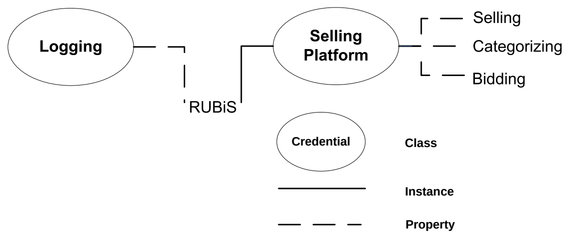

Fig. 8. RUBiS ontology

in Table 1. The table shows the access time for two servlets (`AboutMe` and `SearchItemsInCategory`) for an increasing number of clients. For a lightly loaded system the impact of AOCI is up to six percent whereas for a highly loaded system the impact is between 19-41 percent for the average case. Of course, the impact is related to the underlying AOP engine. Most of the additional spent time is used for weaving the aspect bundles. Within our measures, the time used for the matching algorithm and for collecting the annotations can be neglected.

Servlet	No. Clients	Without AOCI			With AOCI			Avg. Diff. in %
		Min.	Max.	Avg.	Min.	Max.	Avg.	
AboutMe	200	17	476	32	17	230	32	0
	300	16	726	41	16	2623	49	19
	400	16	1926	74	17	4060	105	41
SearchItemsInCategory	200	37	581	75	4	316	79	5
	300	5	1681	97	5	4576	99	2
	400	5	18974	218	21	16765	278	27

Table 1. Benchmark results: plain RUBiS vs. AOCI and RUBiS

In the future, we want to extend the ontology and scenarios. For example, a user could supply an aspect that generates information about her bids and offers.

6 Related Work

We identified three areas of related research that will be discussed in the following.

Extended Pointcut Concepts intend to leverage the actual matching of aspects and code to a higher level. Kellens et al. [26] state that AOP suffers from a *fragile pointcut problem*, which means that seemingly safe modifications of an aspect-oriented program can lead to unexpected impacts in the application. To overcome this issue, they introduce model-based pointcuts, which decouples the aspects from the base code. Hence, the fragile pointcut problem is transferred to a more conceptual level. As opposed to AOCI, they do not introduce ontologies to realize the matching between aspects and pointcuts, instead they extended the CARMA aspect language combined with the formalism of so called ”intensional views”. CARMA uses SOUL, a language akin to Prolog, to reason about the application of advice. The fragile pointcut problem is also addressed by Cyment et. al [27]. In their work, they have introduced an intermediate abstraction layer. On top of this they have implemented model-based pointcuts and a semantic pointcut framework. Compared to Kellens, they embody a higher level of abstraction. However, as opposed to AOCI, they do not use ontologies either.

Distributed AOP applies the ideas and concepts of AOP to distributed applications. In their work, Navarro et al. propose AWED [28], a new aspect language that addresses the distribution of aspects. AWED is realized as an extension of JAsCo, which provides dynamic aspects for Java. AWED offers several explicit distributed programming mechanism. Firstly, they introduce remote pointcuts, which are more general than those of related approaches; secondly, they support the execution of asynchronous and synchronous advice and lastly, they define a notion of distributed aspects including models for deployment, instantiation, and state sharing of aspects. The distribution of aspects is also addressed by Tanter et al. [29]. In their work, they introduce a flexible infrastructure that allows aspects to be used in a distributed manner. Based on their work, it is possible to define distributed aspect languages and frameworks. The distribution of aspects is also supported within AOCI. However, in our infrastructure, we did not extend the aspect language, but enhanced the infrastructure to offer this functionality.

Adaptive Component Systems support modularization of software through components and offer support for runtime adaptation. 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. [30] 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 applications using the support of an ontology or an alternative semantic model.

In their work, Rho et al. [31] 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 every single requirement. Therefore they introduced the concept of context-sensitive service aspects, which can be changed dynamically at runtime. Their framework is Ditrios, which is based on OSGi and supports a context-sensitive weaving. But as their system addresses adaptation on a SOA level, AOP techniques are restricted to calls on the service level.

Duclos et al. [32] 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 works at the 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. These aspects can be applied without code availability. However, the software allows the adaptation of components, but this has to be triggered manually, hence, it is not for use in a ubiquitious environment.

The Rainbow framework [33] provides a software architecture and infrastructure to support the self-adaption of software systems. To fulfill this, 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 and therefore misses concepts for dynamic injection of custom code.

Another approach to enhance component based software development is introduced with the JAsCo [34] language. It is an aspect-oriented programming language, which main contributions are its highly reusable aspects and its strong aspectual composition mechanism for combination of aspects. However, while the adaptations can be applied dynamically at runtime, there is no concept to automate this.

CAMidO [35] is an architecture for supporting development and execution of context-aware component applications. It provides an ontology metamodel, which facilitates the description of context information. 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 a semantic model to automate the adaptation.

Mügge et al. [36] 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. This metadata can be used to weave aspects from the aforementioned Ditrios middleware. However, they do not address context-aware applications. Although there is support for AOP, there is no concept like a greybox.

7 Conclusion and Future Work

We presented the concept of ontology-based pointcuts to enable the dynamic weaving of local and remote components without requiring their source code. Thus, our technique offers support for dynamic adaptation of services in distributed and ubiquitous environments going beyond the adaptation of their external interfaces. Ontology-based pointcuts are provided by introducing an additional abstraction layer mediating between aspects and components using an ontology. We have implemented this concept in the context of the AOCI infrastructure on top of the component framework OSGi. Bundles, which support components in the context of OSGi, provide possible points for extension using the ontology. Aspects on the other side do not need to match at the source level but on the ontology layer level. Thus, AOCI-aware components do not resemble black boxes, in fact they can be seen as greyboxes publishing additional metadata for later adaptation. In conjunction with our service discovery layer, which automatically interacts with other frameworks, our prototype enables adaption in distributed environments. The latter was demonstrated by adapting RUBiS, a web-based bidding system, to provide user induced logging support.

Albeit we have implemented a proof of concept for ontology-based pointcuts and shown that they enable adaptation at a higher level and perform well in the context of OSGi, we see several main fields for further investigation. Firstly, we want to address and evaluate scenarios within the ubiquitious computing domain, where already existing services and functionalities hosted on devices could be extended by other devices. As an example, a mobile device could request another device to apply an encryption aspect before beginning the communication. Secondly, the AOCI infrastructure has to be ported to smaller devices for attacking scenarios in ubiquitous environments. We are confident that this can be achieved by using the Concierge OSGi implementation [37], which was specially designed for mobile and embedded devices and by using the JamVM that we already used for research in the context of resource restricted systems [38]. Thirdly, we have plans to extend the expressiveness of our annotations to enable more complex adaptations. For example, we want support for matching of subclasses within an ontology as well as the use of logical operators within the matching algorithms. Furthermore, we want to evaluate the scaling of our approach in a bigger environment.

References

1. Weiser, M.: The Computer for the 21st Century. Scientific American **265**(3) (1991) 66–75
2. Ferscha, A.: Contextware: Bridging Physical and Virtual Worlds. In: Proc. of the 7th Ada-Europe Int. Conf. on Reliable Software Technologies. Volume 2361., Vienna, Austria, Springer LNCS (June 2002) 51–64
3. Ferscha, A., Hechinger, M., Mayrhofer, R., dos Santos Rocha, M., Franz, M., Oberhauser, R.: Digital Aura. In: Advances in Pervasive Computing. A Collection of Contributions Presented at the 2nd Int. Conf. on Pervasive Computing (Pervasive 2004). Volume 176., Vienna, Austria (April 2004) 405–410
4. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loinghier, J., Irwin, J.: Aspect-Oriented Programming. In: ECOOP. (1997) 220–242
5. Xu, J., Rajan, H., Sullivan, K.J.: Understanding Aspects via Implicit Invocation. In: Proc. of the 19th IEEE Int. Conf. on Automated Software Engineering. (2004) 332–335
6. Filman, R.E., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley Longman Publishing Co., Inc. (2004)
7. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc. (2002)
8. Söldner, G., Kapitza, R.: AOCI: An Aspect-Oriented Component Infrastructure. In Reussner, R., Szyperski, C., Weck, W., eds.: Proc. of the 12th Int. Workshop on Component Oriented Programming (WCOP 2007). (2007) 53–58
9. OSGi Alliance: OSGi Service Platform: Core Specification, Release 4, Version 4.1. Technical report (2007)
10. Klyne, G., Carroll, J.J.: Resource Description Framework (RDF): Concepts and abstract syntax. Technical report, W3C (2004)
11. Büchi, M., Weck, W.: A plea for Grey-Box components. Technical Report 122, Turku Centre for Computer Science (1997)
12. Webster, M.: AOSGi. http://www.eclipse.org/equinox/incubator/aspects/ (May 2008)
13. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: ECOOP '01: Proc. of the 15th European Conf. on Object-Oriented Programming. (2001) 327–353
14. OMG: CORBA Component Model Specification Version 4.0. Technical Report formal/06-04-01, Object Management Group (2006)

15. McGuinness, D.L., van Harmelen, F.: OWL Web Ontology Language. Technical report, World Wide Web Consortium (W3C) (2004)
16. Staab, S., Studer, R., eds.: Handbook on Ontologies. Int. Handbooks on Information Systems. Springer (2004)
17. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF (W3C recommendation). Technical report, W3C (2008)
18. Jena 2: A Semantic Web Framework. http://www.hpl.hp.com/semweb/jena2.htm (2006)
19. Foundation, E.: Equinox OSGi framework. http://www.eclipse.org/equinox (2008)
20. Rellermeyer, J.S., Alonso, G., Roscoe, T.: R-OSGi: Distributed applications through software modularization. In: Middleware. (2007) 1–20
21. E. Guttman, C. Perkins, J.V.: RFC 2608: Service Location Protocol v2. IETF, June 1999
22. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. SIGOPS Oper. Syst. Rev. **27**(5) (1993) 203–216
23. Necula, G.C.: Proof-Carrying Code. In: POPL'97, Paris (1997) 106–119
24. Czajkowski, G.: Application isolation in the Java Virtual Machine. In: Proc. of the 15th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Appl. (2000) 354–366
25. ObjectWeb: RUBiS: Rice University Bidding System. http://rubis.objectweb.org
26. Kellens, A., Mens, K., Brichau, J., Gybels, K.: Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts. In: ECOOP. (2006) 501–525
27. Cyment, A., Kicillof, N., Altman, R., Asteasuain, F.: Improving AOP Systems' Evolvability by Decoupling Advices from Base Code. In: RAM-SE. (2006) 9–21
28. Navarro, L.D.B., Südholt, M., Vanderperren, W., Fraine, B.D., Suvée, D.: Explicitly distributed AOP using AWED. In: AOSD. (2006) 51–62
29. Tanter, É., Toledo, R.: A Versatile Kernel for Distributed AOP. In: DAIS. (2006) 316–331
30. Pessemier, N., Seinturier, L., Coupaye, T., Duchien, L.: A Safe Aspect-Oriented Programming Support for Component-Oriented Programming. In: Proc. of the 11th Int. ECOOP Workshop on Component-Oriented Programming (WCOP'06). (2006)
31. Rho, T., Schmatz, M., Cremers, A.B.: Towards Context-Sensitive Service Aspects. In: Work. on Object Technology for Ambient Intelligence and Pervasive Computing. (2006)
32. Duclos, F., Estublier, J., Morat, P.: Describing and using non functional aspects in component based applications. In: AOSD '02: Proc. of the 1st Int. Conf. on Aspect-oriented Software Development, ACM Press (2002) 65–75
33. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer **37**(10) (2004) 46–54
34. Suvée, D., Vanderperren, W., Jonckers, V.: JAsCo: an aspect-oriented approach tailored for component based software development. In: AOSD. (2003) 21–29
35. Behlouli, N.B., Taconet, C., Bernard, G.: An architecture for supporting Development and Execution of Context-Aware Component applications. In: ACS/IEEE Int. Conf. on Pervasive Services. (2006) 57–66
36. Mügge, H., Rho, T., Cremers, A.: Integrating Aspect-orientation and structural annotations to support adaptive middleware. In: Proc. of the 1st workshop on Middleware-application interaction: in conjunction with Euro-Sys 2007, Lisbon, Portugal. (2007)
37. Rellermeyer, J.S., Alonso, G.: Concierge: a service platform for resource-constrained devices. SIGOPS Oper. Syst. Rev. **41**(3) (2007) 245–258
38. Nikolov, V., Kapitza, R.: Recoverable class loaders for a fast restart of Java applications. In: MOBILWARE '08: Proc. of the 1st int. conf. on MOBILe Wireless MiddleWARE, Operating Systems, and Appl. (2007) 1–8