AOCI: Ontology-based Pointcuts

Guido Söldner, Rüdiger Kapitza
Department of Computer Science 4
Friedrich-Alexander University
Erlangen-Nuremberg, Germany
{soeldner,rrkapitz}@cs.fau.de

Sven Schober
Institute of Distributed Systems
Ulm University, Germany
sven.schober@uni-ulm.de

ABSTRACT

In this paper, we propose *ontology-based pointcuts*, a novel mechanism based on the ideas of AOP, the separation of concerns and system modularization, to enhance components, thus making them more adaptable and evolvable. The core idea of ontology-based pointcuts is to specify pointcuts in terms of ontological concepts instead of patterns that are matched against source code. 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 are extended by exporting possible adaptation points, which are enriched by ontological information. This ontological metadata represents concepts from a domain model, defined by a domain expert. The ontology is implemented within an intermediate layer that decouples the matching pointcuts from the concrete source level within the components.

This enables the application of AOP techniques without detailed knowledge about the component's internals, enabling dynamic and distributed adaptation and reducing the *fragile pointcut problem* to the component scope. This paper presents the design and the expressiveness of the ontology-based pointcuts. We show several practical examples how to use joinpoint model and the pointcut specification followed by a short discussion.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Design

Keywords

OSGi, Dynamic Adaptation, Metadata-driven adaptation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACP4IS'09, March 2, 2009, Charlottesville, Virginia, USA.
Copyright 2009 ACM 978-1-60558-453-9/09/03 ...$5.00.

1. INTRODUCTION

Aspect-oriented programming (AOP) [1] is a technique that can enhance the modularization capabilities of existing programming paradigms like object orientation. 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 applied in the application.

However, the use of AOP in large-scaled software systems or components [2] can sometimes be problematic. As components are black boxes, they hide their implementation which consequently makes the use of AOP difficult. Additionally, some properties of AOP can be quite problematic in large software systems as well.

One of the most important principles of AOP is the *obliviousness* [3] property. Obliviousness states that the base code should not be aware of the appliance of aspects. This permits a greater separation of code as crosscutting concerns can be separated into aspects. Consequently, programmers of the base code don't have to expend any additional effort to make AOP mechanisms work. However, obliviousness is also criticized by some researchers [4], as they claim that finding the code that may be executed at any given point in the program is more difficult. In addition, a developer has to be aware of which advice code 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.

Furthermore, the techniques of aspect-oriented software development that help to improve the modularity of software can restrict the evolvability of that software. As aspects specify where they are invoked by means of a pointcut definition, the pointcut definitions typically rely on the structure of the base program. However, this tight coupling has consequences on the evolvability: whenever the base program evolves, it has to be checked if the pointcuts are still valid, as it can be that they do not capture any longer the intended points in the base program or accidently capture unwanted join points. This problem is named the *fragile pointcut problem* [5].

Based on our previous research work AOCI (Aspect-Oriented Component Infrastructure) [6], we address these problems by introducing *ontology-based pointcuts*. They basically act like standard pointcuts, however they do not rely on the structure of the base code, but instead on an additional abstraction layer which is based on ontologies. These ontologies represent a set of concepts within a domain and the relationships between

those concepts. To match the base code and the ontology-based pointcuts, the developer exports potential points for adaptations by attaching ontology-based meta information to the source code. Thus, information is provided what kind of adaptation is suitable at this points. Aspect-writers describe where advice code should be applied by means of the meta model. Thus, our system supports a greybox component [7] approach, while preserving encapsulation and the concepts of a black box to a certain degree.

Technically, our solution is based on the Equinox OSGi [8] platform and utilizes Equinox Aspects [9], which provides basic AOP support for OSGi. At its core, Equinox Aspects integrates the AspectJ [10] load time weaving features enabling the dedicated weaving of OSGi components.

Our current research work focuses on the dynamic weaving of components in a distributed environment. We have built a prototype for AOCI and demonstrated the feasibility of ontology-based pointcuts. However, so far, the joinpoint model and the pointcut specification lacked of expressiveness. In this work we will describe the enhancements and the feasibility within our ontology-based intermediate layer. This includes enhanced matching capabilities with generalization / specialization concepts, providing access to context, using expressions and context-sensitive weaving.

The next section explains our ontology-based pointcuts followed by an example. An overview of related work can be found in Section 3, followed up by a concluding section.

2. ONTOLOGY-BASED POINTCUTS

In this section, we define the concepts of our aforementioned ontology-based pointcuts. We then explain capabilities of the additional abstraction layer, followed by a discussion. Furthermore, we present a possible use case for our solution.

2.1 Concepts

In our solution, aspects are decoupled from the structure of the base program, consequently the coupling between the source code and the ontology-based pointcuts is lower than with standard AOP.

Fig. 1 illustrates how ontology-based meta information, annotated by the developer of the component, and aspects play together to realize the decoupling between aspects and source code. Within our solution, three different layers can be identified. The application components act as greyboxes and hence export possible points for adaptations. The adaptations are realized as aspects and are maintained by the upper aspect layer. The concrete mapping between the aspects and the greybox components is fulfilled by means of an ontology within the middle layer. The middle layer stores the aspects metadata within its ontology and provides querying capabilities to suggest an appropriate matching between aspects and greybox components.

Within the development model, three different roles can be identified. The application developers enriches the source code with additional metadata, which is expressed in terms of ontology concepts, thus creating a greybox component. Hence, it is his responsibility to decide where and what kind of adaptations are possible. Aspect developers, however, write their advices and reference ontology concepts when creating ontology-based pointcuts. These pointcuts are evaluated by the middle abstraction layer. Hence, the aspect developer can write adaptation code without in-depth knowledge of the application component code. The middle layer reasons about the pointcut expression by means of the ontology and provides a concrete mapping between the aspects and the source code. The ontology is maintained by a domain expert and is extensible.

The use of ontology-based pointcuts within the AOCI framework

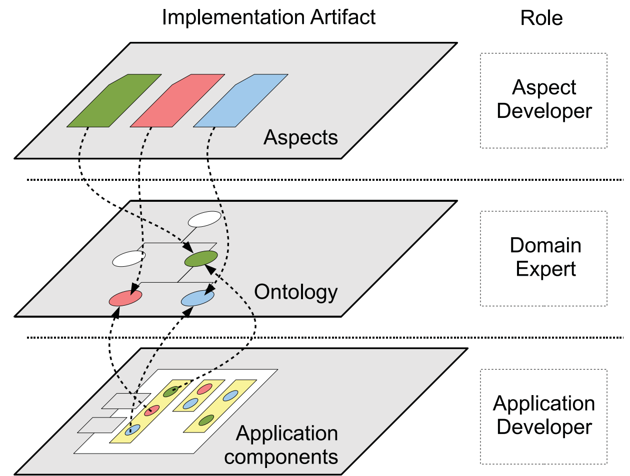

Figure 1: Conceptual structure of an AOCI-aware component

is fairly easy. The application developer enriches his source code with additional metadata. To fulfill this, he can explicitly annotate points for adaptations with attributes. However, this can be cumbersome if there is much source code. To ease this, it is also possible to use standard AOP pointcuts and attach them with ontology concepts to obtain a better quantification [3]. This can be fulfilled by the AspectJ `declare`-statement to define patterns. In Fig. 3 the Java AOCI-annotations, which are based on the annotation capabilities of AspectJ, are automatically converted into OWL (Web Ontology Language) [11] constructs. The method `listItems` is annotated for the application of logging aspects. The mapping of the annotations to the ontology description is quite simple. Ontologies are represented as hierarchical trees, their structure is also represented within the annotations, whereas the corresponding subnodes in the tree are separated with points in the annotation.

```
@AOCI.Logging
public boolean listItems()
{
        ...
        return true;
}
```

Figure 3: Code annotated for use with AOCI

2.2 Intermediate model

As stated, AOCI uses ontologies as the concept representation formalism for the intermediate layer. 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*, *Encryption*, *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. Our ontology is expressed in terms of the ontology modeling language OWL, which provides a set of necessary reasoning and querying operators.

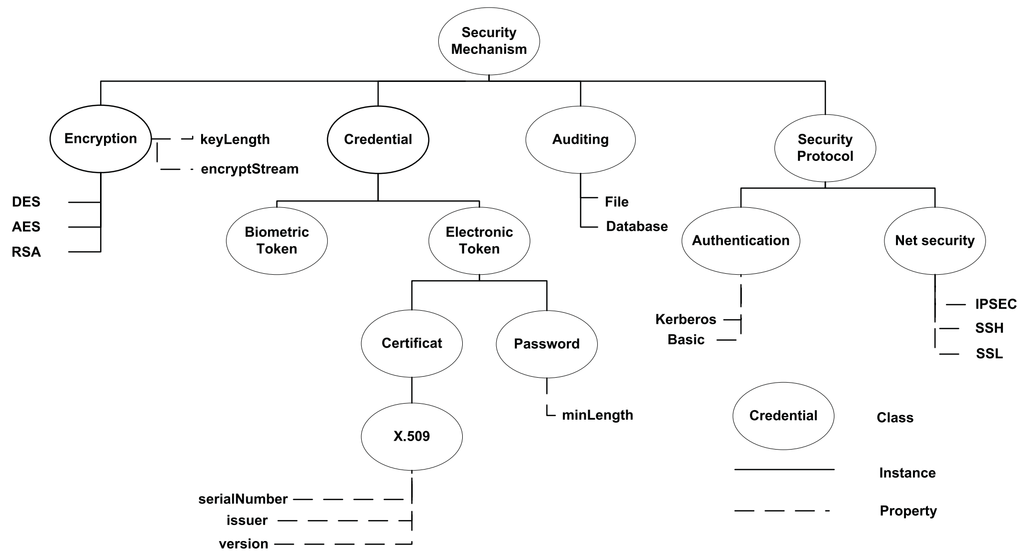

Figure 2: Security ontology

The intermediate layer encapsulates the ontology and does the matching between ontology-based pointcuts and the source code by means of this model. As previously outlined, our abstraction layer already provides the functionality to perform a basic matching, however we plan to extend it to address more complex scenarios. We have identified different enhancements within the middle layer that will be discussed in the following.

2.2.1 *Generalization / Specialization*

Within our current solution, to perform a matching between an application component and an aspect, it is important to specify the same concrete concept within the ontology. However, as there might be several different implementation and sub-concepts within ontologies, we provide support for specialized aspects. The matching between aspects and source code can be enhanced that aspect developers can also provide a specialized implementation within the ontology for the annotated point for adaption. For example, the application developer could state the possibility for authentication at a special method, but leave it to the aspect developer to decide how the authentication is realized (e.g. by a biometric or an electronic token). This scenario is illustrated in Fig. 4. The `subclass`-property states that a more specialized aspect may be woven.

```
@AOCI.Security.Credential(subclass=true)
public boolean listItems()
{
        ...
        return true;
}
```

Figure 4: Code annotation for specialized aspect

2.2.2 *Provide access to context*

So far, aspect developers can only access the execution context by means of reflection. For this purpose, AspectJ provides a special reference variable, `thisJoinPoint`, that contains reflective information about the current join point for the advice to use. Developers can use its rich reflective hierarchy of signatures to access both the static and dynamic information about the join points such as the arguments of the join point. However, as the required metaprogramming skills may become quite complex, a more feasible approach is to provide access by means of the intermediate layer. Therefore, we defined an API to provide access to the execution context and all the properties in the ontology classes. Based on the ontology-based meta information, the AOCI-framework provides getter- and setter-methods for the retrieval of the published metadata. This can be realized by means of code generation and byte code manipulation, which is based on the *ASM* [12] library. Considering the example in Fig. 5, the application developer enriches his source code with information about the stream which can be encrypted (line 2). The aspect developer, on the other side, gains access to the context by means of the `AOCIContext.getCurrentContext()`-method (line 11), which acts as entry point to the AOCI-context. Within the context, the ontology is accessible, subclasses are represented as classes in the class hierarchy and properties are accessible by means of `getter`-methods. Hence, in the example, a simple call against the API is sufficient to access the execution context and read the stream for encryption.

2.2.3 *Expressions*

In more advanced pointcut languages, it is possible to use complex expressions to diminish possible join point captures. In AOCI, we propose to enhance the ontology-based meta information by expressions. Hence, it will be possible to express more complex pointcuts and such achieve a more sophisticated reasoning.

To match the points for adaptations and the aspects, AOCI uses the Resource Description Framework (RDF) [13] to express the metadata and to enable easy processing of the ontology. The underlying structure of an RDF expression is a triplet 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. To implement the ontologies, we used OWL , which is a RDF-based ontology markup language that enables context sharing and reasoning. In AOCI, we use OWL Full [14], a sublanguage of

```
1  //Application component
2  @AOCI.Security.Encryption("encryptStream",
       "a")
3  public OutputStream sendMessage()
4  {
5          OutStream a =  ...
6          return a;
7  }
8
9  //Aspect Bundle
10 try{
11 OutStream a = (OutStream)AOCIContext.
       getCurrentContext().Security.Encryption
       .getEncryptStream("a");
12 }catch(AOCIException ae)
13 {
14 ...
15 }
```

Figure 5: Provide access to context

OWL, which contains the complete set of language constructs of OWL. 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 meta information and the ontology-based pointcut within an aspect. Therefore we use SPARQL [15], 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. Application developer can use SPARQL-queries to specify possible ontology-based meta information. The listing in Fig. 6 shows how to use a query that constrains the use of encryption aspects with less then 64 bit key length. Note that a discussion of SPARQL is out of scope of this paper.

```
@AOCI.Expression("PREFIX aoci <http://cs.fau.
    de/aoci> SELECT ?instance WHERE(?x
    instance ?instance ?me keylength > 64)")
public boolean listItems()
{
        ...

}
```

Figure 6: Annotation with SPARQL query

However, due to the complexity of SPARQL, its use might be cumbersome. Therefore, AOCI provides also a reduced instruction set by means of formulas, which offer the possibility to specify ontology-based meta information. These formulas may contain logical operators as well as comparison operators. The example in Fig. 7 shows how to specify a key algorithm with a key length between 64 and 128 bit.

```
@AOCI.Expression("Formula","(AOCI.Security.
    Encryption.keyLength>64)&&(AOCI.Security.
    Encryption.keyLength<128)")
public boolean listItems()
{
        ...

}
```

Figure 7: Annotation with formula

2.2.4 Context-sensitive behavior

The support for context-dependent behavior is an increasingly important feature for a wide range of application domains, from pervasive computing to common business applications. Therefore, we also plan to extend the AOCI framework by context-sensitive weaving, i.e. the aspect developer may decide the context in which his aspects should be applied. In our solution, we want to address client-specific, system-specific and temporal-specific constraints for context sensitive weaving, i.e. the system behaves differently due to different contexts. An important scenario, for example, might be that aspects should only be applied on a per-user level, for example a logging aspect should only trace the requests of a certain user. To control this behavior, we plan to implement a special file, which is maintained by the administrator of the software. Alternatively, it can be controlled by a set of policies.

To fulfill this feature, once again we plan to use the ASM library to do bytecode engineering and modify the corresponding advices after the weaving of the aspects has taken place. This allows to dynamically adapt the software due to changes in the configuration. Although AspectJ already provides an `if`-pointcut, this is not applicable as it does not work with load-time weaving.

3. RELATED WORK

This paper proposes a mechanism to decouple base programs and aspects using an ontology-based middle-layer. Based on this mechanism, we have implemented our framework AOCI, which enables the application of AOP techniques without detailed knowledge about the component's internals, enabling dynamic and distributed adaptation.

There is also some related work regarding the fragile pointcut problem and the modularity of aspect-oriented programming.

Like AOCI, Open Modules [16] also proposes to explicitly export join points. This prevents aspects from writing fragile pointcuts. As opposed to AOCI, the published join points do not have any semantics and there is no additional intermediate layer.

In [17] Sulivan et al. introduce crosscutting programming interfaces (XPIs). These interfaces are used to abstract crosscutting behavior, hence decoupling aspects and the base code. Pointcuts are written according to design-rules, and in this way pointcuts are made more rubust. Like Open Modules, there is no semantic in the pointcuts and hence no dynamic matching between aspects and base code.

A similar approach is proposed in [18], which introduces explicit join points (EJPs). Similar to AOCI, programmers of the base code are aware of crosscutting concerns. However, like the aforementioned solutions, it lacks of the flexibility of an additional intermediate layer.

Leveraging the actual matching of aspects and code to a higher level is a quite actual topic and several approaches are currently under investigations.

Kellens et al. [19] 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 [20]. 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. Furthermore, the synchronization between the middle layer and the component requires more efforts while the base code is evolving.

Furthermore, there are several works [21, 22, 23] which also address the fragile pointcut problem. To achieve this, they increase the expressiveness and the abstraction level of pointcuts by introducing semantics. As a consequence, they further improve the robustness and precision of the pointcut model. However, as these solutions do not have an intermediate layer, no automatic matching of aspects and base code by reasoning is provided.

Adaptations due to changes in the context are also subject to some research work. In their work [24], Constanza et al. propose a new programming technique, called Context-oriented Programming (COP) which also addresses this problem. Truyen et al. [25] also describe the need for client-specific customization. Therefore, they offer extensions that can be selectively integrated into the core system to achieve the adaptations. We will further investigate which concepts can also be used within AOCI.

4. CONCLUSION

This paper proposes ontology-based pointcuts, a mechanism to make components more evolvable and robust. Such pointcuts are specified by means of concepts within an ontology. Developers of components can attach ontology based meta information to enable dynamic weaving of aspects. Based on our framework AOCI, we propose enhancements to further facilitate and leverage the use of the ontology based meta information and pointcuts. The extensions of AOCI shown in this paper are currently under development. We further plan to implement sophisticated use cases to evaluate the scaling of our approach in a bigger environment. For example, we will focus on scenarios within the ubiquitious computing where dynamic adaptation of components plays an important role.

5. REFERENCES

[1] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loinghier, J., Irwin, J.: Aspect-Oriented Programming. In: ECOOP. (1997) 220–242

[2] Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc. (2002)

[3] Filman, R.E., Friedman, D.P.: Aspect-Oriented Programming is Quantification and Obliviousness. Technical Report 01.12 01.12 01.12, RIACS (2001)

[4] Clifton, C., Leavens, G.: Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy. Technical Report TR03-01a, Iowa State University. 2003

[5] Störzer, M., Koppen, C.: PCDiff: Attacking the Fragile Pointcut Problem. In: EIWAS. (2004)

[6] Söldner, G., Schober, S., Kapitza, R.: AOCI: Weaving Components in a Distributed Environment. In: DOA. (2008)

[7] Büchi, M., Weck, W.: A plea for Grey-Box components. Technical Report 122, Turku Centre for Computer Science (1997)

[8] Eclipse Foundation: Equinox OSGi framework. http://www.eclipse.org/equinox (2008)

[9] Webster, M.: AOSGi. http://www.eclipse.org/equinox/incubator/aspects/ (May 2008)

[10] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: ECOOP. (2001) 327–353

[11] McGuinness, D.L., van Harmelen, F.: OWL Web Ontology Language. Technical report, W3C (2004)

[12] Bruneton, E., Lenglet, R., Coupaye, T: ASM: A Code Manipulation Tool to Implement Adaptable Systems. France Telecom. Technical report (2002)

[13] Klyne, G., Carroll, J.J.: Resource Description Framework (RDF): Concepts and abstract syntax. Technical report, W3C (2004)

[14] Staab, S., Studer, R., eds.: Handbook on Ontologies. Int. Handbooks on Information Systems. Springer (2004)

[15] Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF (W3C recommendation). Technical report, W3C (2008)

[16] Aldrich, J.: Open Modules: Modular Reasoning about Advice. In: ECOOP. (2005) 144–168

[17] Sullivan, K.J., Griswold, W.G., Song, Y., Cai, Y., Shonle, M., Tewari, N., Rajan, H.: Information Hiding Interfaces for Aspect-Oriented Design. In: ESEC/FSE. (2005) 166–175

[18] Hoffman, K.J., Eugster, P.: Bridging Java and AspectJ through explicit join points. In: PPPJ. (2007) 63–72

[19] Kellens, A., Mens, K., Brichau, J., Gybels, K.: Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts. In: ECOOP. (2006) 501–525

[20] Cyment, A., Kicillof, N., Altman, R., Asteasuain, F.: Improving AOP Systems' Evolvability by Decoupling Advices from Base Code. In: RAM-SE'06-ECOOP'06 Workshop on Reflection, AOP, and Meta-Data for Software Evolution. (2006) 9–21

[21] Ostermann, K., Mezini, M., Bockisch, C.: Expressive Pointcuts for Increased Modularity. In: ECOOP. (2005) 214–240

[22] Masuhara, H., Kawauchi, K.: Dataflow Pointcut in Aspect-Oriented Programming. In: APLAS. (2003) 105–121

[23] Sakurai, K., Masuhara, H.: Test-Based Pointcuts for Robust and Fine-grained Join Point Specification. In: AOSD. (2008) 96–107

[24] Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-Oriented Programming. Journal of Object Technology **7**(3) (2008) 125–151

[25] Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P., Jørgensen, B.N.: Dynamic and Selective Combination of Extensions in Component-Based Applications. In: ICSE. (2001) 233–242