Securing Aspect Composition

Andrew Camilleri
Computing Department
Lancaster University
LA1 4WA, UK
a.camilleri@lancaster.ac.uk

Lynne Blair
Computing Department
Lancaster University
LA1 4WA, UK
l.blair@lancaster.ac.uk

Geoffrey Coulson
Computing Department
Lancaster University
LA1 4WA, UK
g.coulson@lancaster.ac.uk

Categories and Subject Descriptors

D.3.0 [**Programming Languages**]: General; D.2.0 [**Software Engineering**]: General

Keywords

aspect oriented programming, security, role based access control models, software engineering

1. INTRODUCTION

Although research in AOP is increasing in maturity there remain many unresolved issues. While current AOP languages offer ever-increasing levels of flexibility, they still fail to offer a sufficient *discipline of application* to ensure that advanced AOP facilities are used safely and appropriately. Researchers have recognised the need to control aspect composition and have started to explore mechanisms to achieve this [2, 3, 4, 5]. In this paper we aim to provide a novel approach to control aspect composition (using AspectJ as reference) and we employ the concept of *roles* from Role Based Access Control Models [1] to characterise aspects in terms of both their *internal behaviour* and their *external composition*. Then, using *policies*, we express invariants and constraints on the associated advice and pointcuts.

2. PROPOSED APPROACH

2.1 Overview

The solution that is being presented in this paper has two underpinning concepts. The first one is the ability to associate an aspect with a unique role. If this could be done, aspects would cease to be anonymous and any aspect that cannot be mapped to a unique role could be safely rejected. The second one is that through a role we are able to enforce compositional constraints to control how aspects modify the target system. An aspect will only be allowed to modify certain parts of an application, in a certain manner.

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.
SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil
Copyright 2008 ACM 978-1-59593-753-7/08/0003 ...$5.00.

In our approach (represented diagrammatically in figure 1), aspects are associated with *roles*: each role can be associated with many aspects, but each aspect is associated with ('conforms to') only one role. A role serves to restrict the behaviour of its associated aspects. For example, a 'logging' role might restrict its conforming aspects to inspecting the arguments of operations and writing to files; but they may not, for example, alter arguments or open socket connections. To support the definition of roles we define an *ontology* that demarcates areas in which restrictions can be specified. One such area, for example, is concerned with restricting advice composition in terms of 'before', 'after' or 'around'. Restrictions are then specified using a *policy*, which is a predicate over the terms of the ontology. In our logging case, for example, a likely policy might specify that 'before' or 'after' advice can be used by conforming aspects, but not 'around' advice. Each role is additionally associated with one or more *clusters*. This is a subset of join points from the target system, and is defined by a predicate. Building on this, our hypothetical 'logging' role might use a cluster of classes that are useful for logging functionality (e.g. file manipulation classes), but would exclude inappropriate or unwanted functionality (e.g. socket libraries).

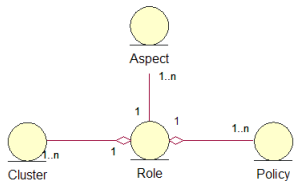

Figure 1: Entity Relationship

In the following subsections we expand on the notions of cluster, policy and role.

2.2 Clusters

An application is usually made up of a number of modules. The composition of these modules is achieved through the units of encapsulation provided by the underlying programming language. Now a cluster provides a means to *slice* the target system in some appropriate manner and this is achieved by means of a predicate which has the same addressing capabilities of a pointcut. In Java a cluster might take the form of a predicate over the package structure of the system, as shown in figure 2. In `logging` the '`/*`' matches all the classes beyond a prefix package, while in `manager` we select methods with specific attributes. The end result is

to provide a static join point grouping mechanism, with the objective to specify the kind of join points that an aspect is allowed to encapsulate, but also the kind of join points with which an aspect is allowed to be composed.

```
logging = {                          manager[Method] = {
        org.apache.log4j./*;                 public * com.bank.manager.*(java.lang.String);
        java.util.logging.*;         }
}
```

Figure 2: Clusters

2.3 Policy and Ontology

Policies allow us to express restrictions on the semantics implied by the different constituents of an aspect. But policies require an underlying ontology onto which restrictions can be applied. In our case the ontology is simply a set of *restriction domains* which focus on the individual pieces of advice, pointcuts, intertypes and on aspects. The policy language that we have designed reflects this ontology structure and is made up from a simple collection of name/value pair properties as shown in figure 3.

```
[pointcut]
pc-logging = {
        modifier = public, private;
        type = execution, withincode, set, get;
        ...
}
[aspect]
aspect-logging = {
        ...
        privileged = false;
        pointcut = pc-logging;
        advice = ad-logging;
}
```

(a) Aspect Policy

```
[advice]
ad-logging = {
        type = before, after;
        ...
}
```

```
[role]
LOGGERS = {
        qualifying-cluster = logging;
        weavable-clusters = manager, backend;
        weaving-clusters = runtime, utils;
        composition-time = compile, load;
        aspect = aspect-logging;
        sequence-num = 3;
}
```

(b) Role Policy

```
[weaving-role]
LOGMANAGER = {
        super = LOGGERS;
        weavable-clusters = manager;
        weaving-cluster = runtime;
}
```

Figure 3: Policies

The first part of the policy language therefore deals specifically with expressing restrictions over the ontology. Basically, this section of the policy language allows us to create templates of acceptable aspects. There are templates that focus on different consituents of an aspect and these are then composed to form a template of a complete aspect as shown in figure 3(a); the `aspect-logging` template imposes its own restrictions (an aspect is not allowed to be privileged) but it also has references to intertype, pointcut and advice templates. These templates allow us to express the different needs of cross-cutting concerns and effectively to define what is acceptable behaviour. An advice template might specify for example that it is possible to use a 'before' or 'after' advice, but not 'around' as shown in `ad-logging`. A pointcut template may restrict the use of specific type of primitive pointcuts as shown in `pc-logging`.

The second part deals specifically with roles and on the kind of restrictions that we want to impose through them. Roles achieve this by combining clusters and aspect templates as shown in figure 3(b). When combined together, they allow us to express a specific set of aspect instances that can be composed with the target system. The roles are made up of a two level hierarchy; the top most roles are specfied under `role` which define a complete *realm* and these are then restricted in `weaving-role`, the second level roles which are the only *active* roles. The join points that an aspect is allowed to use are specified in the `qualifying-cluster` and `weaving-clusters` properties. The semantics of the qualifying cluster will be explained in 2.4, but effectively these two cluster sets allow us to define precisely the join points that make up the body of an aspect. Thus for example, an aspect which performs socket communication will be rejected if join points related to socket communication are not listed in any of these clusters. Another constraint for any mapped aspect is that it needs to conform to the template specified in the `aspect` property. Finally the parts of the application that can be modified by an aspect are specified `weavable-clusters` property, while in case of a conflict we can specify the role weaving priority in the `sequence-num`.

2.4 Roles

Central to the concept of a role is the ability to identify an aspect within a role. This is achieved because a role is associated with at least one cluster, the qualifying cluster. If you recall from 2.2, a cluster is characterized from a predicate that defines a set of join points. Now an aspect is *qualified* to a specific role because it contains join points that are elements of the corresponding qualifying cluster. The qualifying cluster needs to provide *primary join points* i.e. it should encapsulate the join points that characterize the primary function of an aspect. An aspect that is addressing the logging concern would be primarily concerned in perform logging operations. So all the logging join points should be encapsulated in the qualifying cluster of the logging role. It is therefore central that all the join points that perform cross-cutting operations are restricted to the qualifying cluster of the corresponding role. This requirement is not unreasonable; intuitively if an aspect is trying to perform logging and security operations, it means that it is trying to address more than one cross-cutting concern and this goes against one of the underlying objectives of aspects and AOP.

3. CONCLUSION

It is obvious that research in AOP has not given enough attention to secure aspect composition. We provide a solution that is based around roles. After an aspect is qualified to a role (through clusters), we are able to impose restrictions through clusters and the ontology, using policies. The solution we provide is still a work in progress and we plan to evaluate it after implementation is completed.

4. REFERENCES

[1] Ravi S. Sandhu et al. Role-Based Access Control Models. In *Proc of IEEE Computer*, volume 29, pages 38-47. IEEE 1996.

[2] David Larochelle et al. Join Point Encapsulation. *In Proc SPLAT in conjunction with AOSD*, 2003.

[3] J. Aldrich. Open Modules: Modular Reasoning about Advice. *In Proc. of European Conference on Object-Oriented Programming*, volume 3586 of LNCS, pages 144-168. Springer 2005.

[4] S. Gudmundson and G. Kiczales. Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface. *In Proc. of ECOOP* 2001.

[5] Kevin Sullivan et al. Modular Software Design with Crosscutting Interfaces. In *Proc of IEEE Software*, volume 23, pages 51-60. IEEE 2006.