A Reflective Framework for Fine-Grained Adaptation of Aspect-Oriented Compositions

Paul Grace*, Bert Lagaisse, Eddy Truyen, and Wouter Joosen

Department of Computer Science, K. U. Leuven, Leuven, Belgium
p.grace@lancaster.ac.uk,
{Bert.Lagaisse, Eddy.Truyen, Wouter.Joosen}@cs.kuleuven.be

Abstract. Dynamic Aspect Oriented Programming (AOP) technologies typically provide coarse-grained mechanisms for adapting aspects that cross-cut a system deployment; i.e. whole aspect modules can be added and removed at runtime. However, in this paper we demonstrate that adaptation of the finer-grained elements of individual aspect modules is required in highly dynamic systems and applications. We present AspectOpenCOM, a principled, reflection-based component framework that provides a meta object protocol capable of fine-grained adaptation of deployed aspects. We then evaluate this solution by eliciting a set of requirements for dynamic fine-grained adaptation from a series of case studies, and illustrate how the framework successfully meets these criteria. We also investigate the performance gains of fine-grained adaptation versus a coarse-grained approach.

1 Introduction

Component frameworks must now support complex compositions of application components, including a broad range of services that deal with non-functional concerns. Aspect-component frameworks e.g. JAC [14], JBoss AOP [7], Spring [16], and Prose [15] have contributed to improving the modularization of such complex applications, by supporting an aspect-component model that offers aspect-oriented composition (AO composition) alongside traditional composition of provided and required interfaces. The core concept in AO composition is the aspect [5]: a first-class citizen that encapsulates one specific (often crosscutting) concern in a separate software module. An aspect defines *behaviour* and *composition logic* describing where and when this behaviour is executed. Aspect-component frameworks often separate aspect behaviour and composition logic, for the purpose of reusing aspect-behaviour across applications. Composition logic is specified declaratively, e.g. in the form of: *whenever event X in the application occurs, execute method behaviour Y of component Z*. For example, whenever a component operation is executed, execute the enforcement method of the authorization component. The search for expressive composition mechanisms is an ongoing track in the research community, yet the composition logic of a real world application

* while on leave from Lancaster University, UK

remains complex and is subject to runtime changes. Therefore, there are increasing requirements to adapt the composition logic at runtime, as exemplified by autonomic, self-repairing and self-optimising systems [8].

A recent advance in aspect-component frameworks is the ability to express dynamic AO compositions that weave and unweave aspects at runtime. For the purpose of this paper, we identify two styles of adaptation of dynamic AO compositions:

- *Coarse-grained adaptation.* An entire aspect (i.e. behaviour and composition logic) is added to or removed from the application.
- *Fine-grained adaptation.* The fine-grained elements that compose an aspect are reconfigured. For example, the composition logic is altered to change where the behaviour is executed. Similarly the aspect behaviour can be adapted.

However, state of the art dynamic AOP frameworks [14, 16, 15] typically only provide coarse-grained adaptation. In this paper, we present AspectOpenCOM[1] a component framework for fine-grained, runtime adaptation of aspects. AspectOpenCOM is an extension of the OpenCOM reflective component model [2]. A reflective system maintains a representation of itself that is causally connected to the underlying system, and provides a *meta object protocol (MOP)* with a set of methods to introspect and adapt this meta-representation [12]. AspectOpenCOM adds two features to OpenCOM: i) AO compositions, and ii) the *aspect MOP* which provides two key functions to the developer: *aspect introspection*, where the deployed aspects can be enumerated and inspected e.g. in terms of composition logic and behaviour descriptions; and *aspect adaptation*, whereby deployed aspects are adapted in-situ.

We evaluate AspectOpenCOM using a set of requirements elicited from a series of case studies and illustrate how the framework is used to meet these requirements compared against coarse-grained adaptations.We demonstrate that fine-grained adaptation has the following benefits over a coarse-grained approach:

- *Flexibility and robustness of adaptation.* The wider range of introspection and adaptation operations available in the fine-grained method provides more support for informing and performing complex adaptations.
- *Conflict resolution.* A well established problem of dynamic AO compositions is that the behaviour from separate aspects can conflict once deployed [5]. A fine-grained approach can resolve this by adapting behaviour at a location without recomposing multiple system-wide aspects.
- *Performance improvement.* An AO composition can cross-cut a large number of system modules; hence, coarse-grained adaptation can disrupt portions of the system unnecessarily and degrade the adaptation time, whereas targeted fine-grained adaptation only adapts the required elements improving performance.

[1] Available for download from `https://sourceforge.net/projects/gridkit/`

The remainder of the paper is structured as follows. Section 2 provides a brief background on reflection and dynamic AOP technologies. Section 3 presents the series of case studies that require fine-grained aspect adaptation. Section 4 presents the concepts and implementation of the AspectOpenCOM component framework and the Aspect MOP, which is then evaluated in section 5. Section 6 provides analysis against related work, and finally section 7 draws concluding remarks and identifies areas of future research.

2 Background on AOP and Reflection

A key element in the specification of the aspect composition logic is the concept of a *pointcut* which is a description of a set of join points where aspects execute. *Join points* represent i) dynamic, runtime conditions that arise during program execution or ii) locations in the structure of the program code. The occurrence of such a condition or location can trigger the execution of aspect behaviour. *Advice* specifies what aspect behaviour should be executed and when the aspect behaviour should be executed (typically before, after or around the event) [9].

Pointcuts select join points by declaratively specifying the kind and context of join points. The *kind* of a join point refers to the type of instruction being executed. For example, two different kinds of join points are method call and field access. The *context* of joint points refers to additional information that can be made available to constrain the pointcut such as the method signature, the interface of the component and the component. Which kind of context and which available context that are supported by an aspect component framework, are defined by the framework's *join point model* [4].

Reflection is the capability of a system to reason about itself and act upon this information. For this purpose, a reflective system maintains a representation of itself that is causally connected to the underlying system that it describes [12]. Operations to introspect and make changes to the meta-representation are commonly referred to as the Meta Object Protocol (MOP). In component-based frameworks, two styles of reflection have emerged. Structural reflection is concerned with the underlying structure of objects or components i.e. it is possible to inspect interface information, and adapt software architecture topology. Behavioural reflection is concerned with activity in the underlying system, e.g. in terms of the sending and dispatching of invocations.

3 The Case for Fine-Grained Aspect Adaptation

To motivate fine-grained adaptation of aspects we consider the following reconfiguration rich use case scenario. A banking application has a façade-based architecture as seen in Figure 1; the façade *bankingservice* is a component which is accessed by remote clients; in this layered architecture the façade then interacts with entity components (e.g. *Account* and *BasicBanking*) at greater depths. We now illustrate scenarios involving tracing, security and caching aspects and elicit a set of requirements for fine-grained adaptation.

Fig. 1. Join point set adaptation

3.1 Join Point Set Adaptation

This use case illustrates the need for adaptation of the join point set of individual aspects. We show this in two cases: i) enforcing stricter security policy in the banking application, and 2) optimising resource usage while tracing the system

Security policy enforcement. An authorization aspect is used in the banking application to check the current user's rights; this typically occurs (and must occur) at the façade component: *bankingservice*. However, a more severe enforcement of the authorization policy can apply in the case of suspicious behaviour (e.g. detected and triggered by an intrusion detection system). This dynamic detection of suspicious behaviour can be based on i) malformed incoming messages on the network, ii) a suspicious number of authentication attempts by the user (e.g. 3+), or iii) a suspicious (accumulative) amount in the financial transaction executed by the authenticated user. The more severe enforcement includes executing the authorization advice deeper in the control flow of the incoming messages: e.g. also at the Account and BasicBanking components (by **reconfiguring the join point set**).

Tracing. In the layered set of components as depicted in Figure 1 the application requires that a monitoring aspect applies a set of advices to create a trace of the call flow. Depending upon the load of the system, the call flow depth of the trace is dynamically adapted at runtime: at a high load detailed tracing can impact the system throughput, hence only the called operations on the façade (bankingservice) are traced (using pointcut P3). When the load is lower for the system, a deeper trace is created (e.g. pointcut P2 selects all join points up to a 2-layer depth, pointcut P1 selects all join points up to a depth of 3 layers. Hence, the scenario requires switching at runtime between these 3 pointcuts while maintaining the state of the aspect.

3.2 Aspect Behaviour Adaptation

Here we examine separate use cases that illustrate the need to adapt the advice deployments that form the behaviour of aspects. In the banking application, when the intrusion detection system is triggered an additional audit advice must be applied to all those operations in the control flow of the user's session to construct an irrefutable behavioural audit trail of the session. This trail is typically used as a security measure to detect fraud or tampering a posteriori. This is carried out **by adding a new audit advice to the existing security aspect**.

Alternative to the banking application, consider a traditional client-server system with no initial aspects woven into the system. However, when the mean execution time of client requests deteriorates beyond some predetermined threshold due to network latency, a cache aspect is woven into the system. This aspect intercepts client requests and checks a local cache to see whether the same request has already been issued. Later, when the system must operate in a secure mode, an authentication aspect is dynamically woven into the system; this consists of an advice that denies the client access to the server until they provide correct identification credentials. These two aspects execute at the same join point, and their order is critical for the correct operation of the system. If the cache advice is executed before the authentication advice, clients are able to get access to resources without authenticating themselves first. Hence, in order to determine if there is a conflict it must be possible for a third-party program to **inspect the current state of aspect compositions**; if there is a conflict the same program must **re-order the advices**.

3.3 Requirements

From the bold text in the previous scenarios we elicit four requirements; these must be met to fully support fine-grained adaptation of aspect compositions. We believe that these cover a broad range of requirements across scenarios, although this is not exhaustive and other requirements may emerge from alternative scenarios.

1. Dynamically adapt the pointcut expression to change the join points where an aspect is deployed.
2. Add advices to an already defined set of join points in an existing aspect. Also replace and remove advices.
3. Inspect information about current aspect deployments to provide information to third-party reconfiguration programs.
4. Dynamically reconfigure the order of advices at a join point.

4 The AspectOpenCOM Framework

In this section we discuss the AspectOpenCOM framework for supporting fine-grained aspect adaptation. We first introduce the underlying component and aspect composition mechanisms. Subsequently, we discuss the reflective meta-object protocol that allows third parties to adapt system aspects.

4.1 A Reflective Component Model

An overview of the AspectOpenCOM architecture for composing components and aspects is presented in Figure 2. Notably, reflection is at the core of this architecture, as we believe this technique is the most principled approach to dynamic adaptation; information is available about the current system state to inform both adaptation decisions and the verification of changes. We essentially extend the well-established OpenCOM component kernel [2] to include AO compositions. The key features of the original component composition model are as follows. *Components* are encapsulated units of functionality and deployment that interact with other components exclusively through interfaces and receptacles. *Interfaces* are expressed in terms of sets of operation signatures and associated datatypes. *Receptacles* are "required" interfaces that are used to make explicit the dependencies of a component on other components. *Bindings* are associations between a single interface and a single receptacle. The kernel provides an API to create new components and aspect components and the bindings between them. In Figure 2 there are two components bound by a receptacle and interface, and an aspect component bound to component's interface.

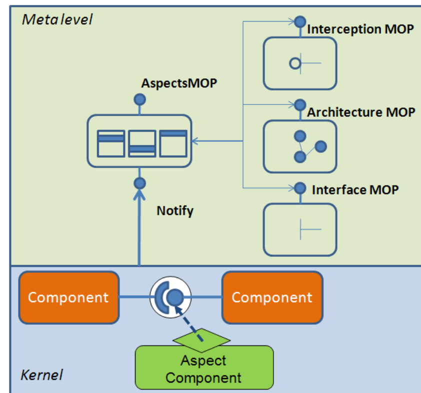

Fig. 2. The AspectOpenCOM Component Kernel Architecture

The kernel also maintains a set of four meta-object protocols each supporting the inspection and adaptation of a distinct system view. Changes made via these MOPs are reflected in the underlying compositions using a causal connection. We will show later how these MOPs can be combined to implement fine-grained aspect adaptation (i.e. we can re-use common reflection behaviour). The MOPs are briefly described as follows; we will examine the aspect MOP solely in this paper; for more information about the other three see [1].

- The *interface MOP* supports inspection of a component's provided and required interfaces. Typically, you can examine the operations available on these interfaces, and or dynamically invoke one of the operations.
- The *architecture MOP* accesses the software architecture of a component represented by a component graph; which is a meta-data description of com-

ponents and bindings, where a binding maps between a required and provided interface in the same address space.
- The *interception MOP* enables the dynamic insertion of interceptors, which support the insertion of pre-, around and post- behaviour on to interfaces. The interceptors are executed before and after each operation invocation. Hence, the interception MOP provides similar behaviour to traditional aspect compositions, however interceptors are interface specific and do not support the deployment of system wide cross-cutting concerns; hence, it isn't suitable as a full aspect MOP, rather here it forms the underlying weaving mechanism (discussed later).
- The *aspect MOP* supports fine-grained introspection and adaptation of aspect compositions.

4.2 AO Compositions

In AspectOpenCOM, the kind of a join point is a method call on a component interface, either at the caller (on the receptacle) or the called side (on the interface). Table 1 describes the key elements of a pointcut expression that locates the join points; i.e. what kind, and a set of regular expressions to match to elements in the component graph. Further, advice behaviour is encapsulated in *aspect components*. Table 2 describes the elements of a created advice i.e. how it is executed (either before, after or around the original method call), and where that operation is hosted on an aspect component.

Table 1. Elements of an AspectOpenCOM pointcut expression

Field	Description
PointcutType	Where applied: Call (receptacle) or Execution (Interface)
ComponentExpression	Regular expression for matching against component types
InterfaceExpression	Regular expression for matching against interface types
MethodExpression	Regular expression for matching against method signatures

Table 2. Elements of an AspectOpenCOM advice

Field	Description
AdviceRole	Before, After or Around advice
InstantiationScope	Singleton component per address space, per aspect, etc.
ComponentType	Type of advice component
Interface	Interface declaring advice operations
Method	Advice operation name

The runtime composition of aspects differs from a standard component to component binding (which is a direct reference from a receptacle to an interface). This binding infrastructure is illustrated in Figure 3. Each interface (execution join point) and receptacle (call join point) has a proxy that redirects the

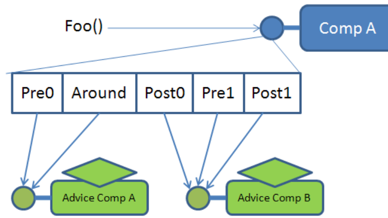

Fig. 3. Proxy-based advice execution chain

original call through a chain of advices. The call towards the original operation invokes pre and around advices in the order encountered (by redirecting to the corresponding advice component), ignoring post operations. After the call, the post and around advices are executed in the order encountered. Hence, in Figure 3 the order is: Pre0, Around (part before proceed), Pre1, Foo, Post1, Post0, Around (part after proceed).

4.3 Aspect Meta-Object Protocol

There are three parts to the aspect meta object protocol of AspectOpenCOM: i) the meta-representation of the deployed aspects, ii) the set of operations that act upon the meta-representation, and iii) the causal connection that ensures consistency between the base and meta-level.

The meta representation is illustrated in Figure 4; essentially the aspect MOP maintains two related data sets: one containing Aspect descriptions and one containing join point descriptions (i.e. locations in the component graph); these are related where an aspect is deployed at 0 or more join points, and a join point can have 0 or more aspects. The Aspect type is the description of the pointcut (described in table 1) related to a set of advices (whose attributes e.g. AdviceRole are discussed in table 2).

The **reflection operations** are also documented in Figure 4. The introspection operations are as follows. The *getAspectInfo* method returns the meta-data about a current aspect deployment; *enumerateAspects* lists all currently deployed aspects; *enumAdvices* lists the advices at either an individual join point or for a deployed aspect; *enumPoints* describes all the join points where an aspect is currently active at; finally, *AspectIntersect* takes two aspect deployments and calculates the set of join points where they are both active.

The dynamic adaptation operations are as follows. The *replacePointcut* method allows the developer to pass a new pointcut expression and the existing aspect behaviour will be moved from the prior join point set to the new join point set. Further, *addAdvice* adds new advice code to either an aspect composition or to an individual join point (*removeAdvice* the opposite); finally, *reorderAdvices* takes the new advice order for a join point location and adapts the behaviour accordingly. We also include coarse-grained adaptation with the *addAspect* operation; this composes a new aspect into the running system and *removeAspect* removes it.

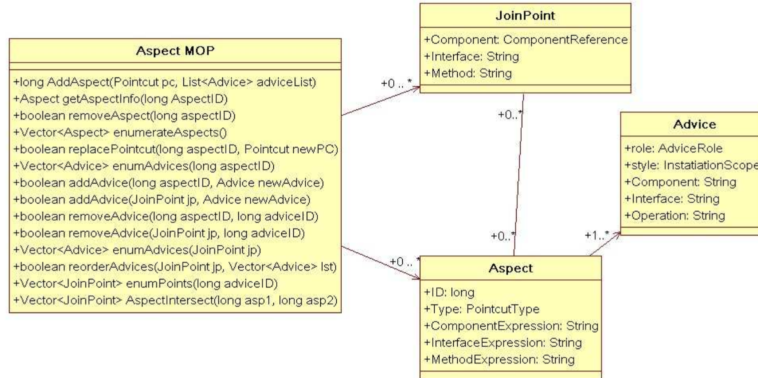

Fig. 4. The Aspect MOP

The **causal connection** is the key element of the aspect MOP implementation. Figure 2 illustrates this architecture. The Notify interface relationship between the kernel and the component implementing the MOP is a one-way notification mechanism that informs the meta-level of all base-level operations i.e. new component creations (including advice components), new binding compositions, component removals, etc. The MOP component then updates its meta-data to reflect the base level changes. It also ensures that Aspects are correctly applied, for example when new components are introduced whose join points match the composition logic of already deployed aspects then the meta-level automatically deploys these aspects to the newly introduced join points.

To perform the previously described introspection and adaptation operations, the aspect MOP interacts with the three additional reflective MOPs described earlier. For example, advice components can be created using the architectural MOP, composition of advices into the join point proxy chain is performed using the interception MOP, and join points can be discovered by introspecting the architecture, and interface MOPs. For brevity, we do not provide a complete description of how all fine-grained operations are performed; instead we examine the *replacePointcut* operation.

The pseudocode in Figure 5 illustrates the implementation of a reconfiguration of an aspect's pointcut description at runtime. The deployed aspect and new pointcut logic are passed as parameters to the *replacePointcut* meta operation. First the intersection of the two pointcuts is calculated to discover which join points will remain the same. Second, the difference between the original aspect's join point set and the intersection is calculated, and all of the aspect's advices are removed from these join points (n.b. the interception MOP is utilised for this behaviour). Finally, the intersection and new pointcut's join points is evaluated and all the aspect's advices are composed to this set.

```
1  ReplacePointcut(Aspect A, Pointcut B)
2    List<Joinpoint> isect = AspectIntersect(A, new Aspect(B, null));
3    List<Joinpoint> rDiff = JPSetDifference(enumPoints(A.pcut), isect);
4      Foreach Joinpoint jp in rDiff
5        Foreach Advice adv in A.adviceList
6          Delegator del = InterceptionMOP.getDelegator(jp);
7          del.removeOperation(adv.operation);
8      List<Joinpoint> aDiff = JPSetDifference (enumPoints(B), intersect);
9      Foreach Joinpoint jp in aDiff
10       Foreach Advice adv in A.adviceList
11         Delegator del = InterceptionMOP.getDelegator(jp);
12         del.addOperation(adv.operation);
```

Fig. 5. Pseudocode describing the implementation of replacePointcut

5 The Benefits of Fine-Grained Adaptation

5.1 Meeting the Requirements of Fine-Grained Adaptation

We examine how AspectOpenCOM meets the requirements described in section 3 and show that the framework provides a robust, flexible method to better support the developer perform complex adaptations; we also compare this against implementations of the same adaptation behaviour using traditional coarse-grained weaving and unweaving of complete aspect modules. Further, we show how problems such as conflicting aspect deployments can be overcome in a fine-grained manner.

Requirement 1: Dynamically adapt the pointcut expression to change join points where an aspect is deployed. Figure 6 illustrates the simple 2 lines of code for performing the reconfiguration from pointcut P1 to P2 as illustrated in Figure 1; first the new pointcut description is defined (this is an execution type applied to interface join points, the second parameter is the component expression i.e. all *BasicBanking* and *BankingService* components, the final two parameters indicate any interface and method on these components. Subsequently, the *replacePointcut* operation is called. This is comparable to a coarse-grained approach where the entire aspect must first be removed, a new aspect module must be created, then this must be dynamically woven as the replacement. This performs unnecessary adaptation at join points whose behaviour does not change, and is more complex to manage due to multiple instances of the same aspect.

```
1    Pointcut expr = new Pointcut(PointcutType.EXECUTION,
2        "BasicBanking*||BankingService*", "*", "*");
3    aspectMOP.replacePointcut(traceAspect, expr);
```

Fig. 6. Java code for performing join point set adaptation

Requirement 2: Add advices to an already defined set of join points in an existing aspect. Figure 7 shows the code to create a new meta data description of the audit advice (described in the use case scenario) and then add this to the security aspect using the addAdvice MOP operation. Hence, the *auditOperation*

hosted on the *IAudit* interface of the singleton *AuditComponen* is applied before execution of the operation. Note, if the advice component (*AuditComponent*) already exists it will be re-used, otherwise it will be created during the meta operation. As with the previous requirement, coarse-grained adaptation can mimic this behaviour by removing the complete existing aspect and replacing it with a new aspect that contains the additional advice. However, if multiple advices are deployed per aspect these will be unnecessarily adapted.

```
1 Advice audit = new Advice(AdviceRole.BEFORE, InstantiationScope.SINGLE,
2     "AuditComponent", "IAudit","auditOperation");
3 aspectMOP.addAdvice(securityAspect, audit);
```

Fig. 7. Java code for adding an advice dynamically

Requirement 3: Inspect information about current aspect deployments to inform decisions. There are 5 introspection operations that allow the developer to fully discover the state of aspect configurations. Figure 8 shows how the *aspectIntersect* operation is used to find where two conflicting aspects are deployed (in this case two aspects, one role-based access control the other credential based) and then deploy an advice at each resulting join point location using *addAdvice* to resolve this conflict (the resolve advice ensures authentic users are not denied by the conflicting access control method). Coarse-grained technologies typically provide only limited introspection capabilities that reflect the coarse-grained modules deployed, and as such do not support this behaviour.

```
1 Advice resolve = new Advice(AdviceRole.BEFORE, AdviceStyle.SINGLETON,
2     "Resolver", "IResolve", "resolve");
3 Vector<JoinPoint> lstJP = aspectMOP.aspectIntersect(Role, Credential);
4 for(int i=0; i<lstJP.size();i++){
5   aspectMOP.addAdvice(lstJP.get(i), resolve);
6 }
```

Fig. 8. Java code introspecting aspect behaviour

Requirement 4: Dynamically reconfigure the order of advices at a join point. Figure 9 illustrates how the security and cache conflict in section 3.2 can be resolved (note we assume only 2 aspects are deployed in this code). First we find the join point set of the security aspect, and calculate if the advices deployed here are in conflict; if so they are re-ordered. Coarse-grained adaptations do not have the fine-grained knowledge to re-order advices, one alternative is to remove all aspects and then add them in the correct order; again, this will perform unnecessary addition and removals of system elements.

```
Vector<JoinPoint> lstJP = aspectMOP.enumPoints(securityAspect);
Vector<Advice> correctOrder = new Vector<Advice>({security, cache});
for(int i=0; i<lstJP.size();i++){
  Vector<Advice> adList = aspectMOP.enumAdvices(lstJP.get(i));
  If (after(security, cache, adList))
    aspectMOP.reorderAdvices(lstJP.get(i), correctOrder);
}
```

Fig. 9. Java code for reordering advices at a join point

5.2 Analysis of fine-grained versus coarse-grained adaptation

This demonstrates the flexibility of the AspectOpenCOM framework to robustly support a wide range of complex fine-grained adaptation requirements. We summarise in table 3 the differences in how the four requirements can be met using both fine and coarse-grained approaches. It can be seen that fine-grained fully meets all the requirements, and where both offer support fine-grained does so without performing unnecessary adaptations. Coarse-grained approaches partially meet the first two requirements, in that complete module adaptation mimics the behaviour, however, this performs unnecessary adaptation, has the potential to lose state (when advices are removed) and increases the time taken to perform reconfiguration. We illustrate these issues further in the next section.

Table 3. Meeting the adaptation requirements

Requirement	fine-grained	coarse-grained
1 join point set adaptation	YES	PARTIAL
2 aspect behaviour adaptation	YES	PARTIAL
3 deployment introspection	YES	NO
4 advice re-ordering	YES	NO

5.3 Investigating Performance Gains

We now investigate the performance benefits of fine-grained adaptation versus coarse-grained adaptation in terms of changing the join point set. For this we created an experimental application with a set of components and pointcuts as illustrated in Figure 10(a). Here there are two component types (A and B) with a set of 10 join points on each. P1 is the initial pointcut that activates all join points on component A (illustrated by the filled circles); we then reconfigure to poincut P2, which matches all methods starting with "b" on both components. Hence, we have a join point set size of 10 in both cases, where 5 join points remain the same and 5 new ones are activated (i.e. 50 % remains unchanged). In subsequent experiments, we change the percentage of join points that stay the same by increasing the "b" join points on component A and reducing them on B (90% = 9 on A and 1 on B, 20% = 2 on A and 8 on B etc.,). Finally, to test scalability we increase the join point set size from 10 upwards by deploying more instances of components A and B (e.g. 10 of A and 10 of B equals a join point set size of 100). All experiments were executed in a single instance of a Java

1.5.0.10 virtual machine on a laptop with a 1.7 GHz Pentium 4 processor, 512 Mbyte of RAM and running Windows XP. For fine-grained adaptations (FG) we utilise the *replacePoincut* operation, for the coarse-grained approach (CG) we use the *removeAspect* followed by *addAspect* operations.

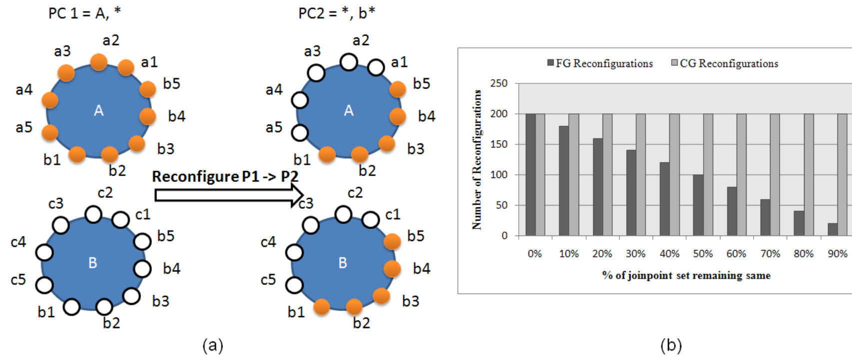

Fig. 10. (a) The application adaptation, (b) Unnecessary reconfigurations in coarse-grained adaptations

The first experiment illustrates the number of unnecessary reconfigurations undertaken by a coarse-grained approach. We created the application as described above with a join point set size of 100 and then traced the number of reconfigurations in terms of removing and adding advice behaviour. We did this for pointcut transformations between 0% of the set staying the same through to 90% of the set remaining unchanged. The results in Figure 10(b) indicate that in the worst case, where there are minimal changes to the join points, many unecessary adaptations are performed e.g. 180 at 90%.

Our second experiment illustrates the effect this has on the performance of the system in terms of the time taken to perform reconfigurations. For this, we timed the transformations from P1 to P2 for coarse-grained, and fine-grained with 0%, 30%, 60% and 90% of the join point remaining unchanged. Each of these five styles was measured with increasing join point set sizes from 10 up to 5000. Note, to discount anomolous results each measure was repeated 5 times with the median value being taken. Figure 11 illustrates the performance gains of the fine-grained approach. It can be first seen in Figure 11(a) that as the percentage of the join point set that stays unchanged increases then the adaptation takes considerably less time. Note, for 0% unchanged (which is equivalent to coarse-grained adaptation) the measurements are similar to the CG timings; this shows that the overhead of fine-grained reflection (e.g. calculating the changes) is not prohibitive even when all the join points must be changed. To demonstrate this further, Figure 11(b) shows the percentage performance increases of FG compared to the equivalent CG adaptation with much larger applications sizes; this illustrates the improvement is related to the change in the join point set (less

change implies bigger performance gains), and that this effect is maintained as the system scales.

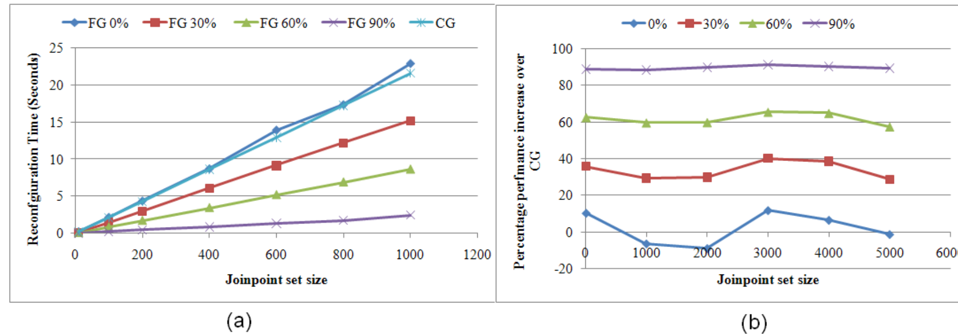

Fig. 11. Comparison of the performance of fine-grained and course-grained adaption (a) Time to Adapt, (b) % performance increase of FG over CG

To conclude, we can see that the performance of adaptation is related to the number of unnecessary adaptations performed in a coarse-grained approach. Where elements of a join point set remain unchanged significant performance benefits can be realised using fine-grained adaptation.

6 Related Work

A number of dynamic AOP technologies have been developed; these typically vary in how aspects are weaved (e.g. efficient bytecode rewriting, dynamic proxies, etc.), when aspects are weaved (i.e. load-time or run-time), and where aspects are weaved (i.e. internal or external to the component module). JBoss dynamic AOP [7] weaves interceptor-based advices (i.e it relies on Java dynamic proxies) to Java-based components within an application server; notably, the component join points must be pre-prepared at compile-time for aspects to be applied at run-time, hence it may not be able to facilitate all unanticipated adaptations. Prose [15] uses JIT compiling techniques to weave advice implementation dynamically at join points of plain Java objects. The JAC framework [14] composes aspect-components to core Java-based components at run-time using byte-code rewriting; the framework also provides a set of reusable aspects, namely persistence, caching. Each of these technologies typically focus on coarse-grained composition of aspects. ByteSurgeon [3] is a bytecode manipulation framework supporting fine-grained computational reflection at runtime (as opposed to load-time), and offers an alternative implementation approach for our Aspect MOP.

However, some dynamic AOP solutions provide features for finer-grained adaptation as discussed in this paper; for example, PROSE contains operations to inspect some elements of the aspect composition, and also adapt advice behaviour. Similarly, JAC and JBoss can manipulate the advices deployed at join

points. However, these have been added to the technologies via ad-hoc extensions, no solution currently provides a principled adaptation interface (cf reflection) with a complete set of fine-grained operations as advocated in this paper. Furthermore, none of these technologies supports join point set adaptations. In this respect, morphing aspects [6] is the closest to this concept; the join point shadow is morphed to apply only where required (reducing unnecessary checks to see if the aspect applies at the join point at a certain point in time). Our approach differs in that the morphing aspects are created at design time, and hence are not subject to third-party adaptations as available in a reflective approach.

Finally, the highly related nature of reflection and aspects (they are both meta-level technologies) means they are commonly combined. The Reflex AOP kernel [17] is a notable example of this; a partial reflection approach is developed that allows the meta-level to be tailored to particular requirements, in turn reducing the complexity of meta programming, and reducing resource usage; the kernel then utilises partial reflection for versatility, for example underpinning aspects from multiple languages. Alternatively, Kojarski et al. [10] explore the two-way relationship between aspects and reflection; they argue that AOP is another computational reflection mechanism, where a join point model reflects the program's behaviour and the advice provides the intercession capability. Further, they identify that AOP can be implemented atop reflection; pointcut descriptions rely on introspection information from structural MOPs, and advices rely on behavioural MOPs. Notably, they also identify that reflection can be implemented atop aspects i.e. using aspects to generate data provided by Java reflection (e.g. field introspection). Our approach differs from these related frameworks, in that we introduce an Aspect MOP with a richer set of fine-grained operations to support principled third-party adaptation of aspect compositions at run-time.

7 Concluding Remarks and Future Work

In this paper we have illustrated the need for fine-grained adaptation of aspect compositions. We have demonstrated scenarios where fine-grained adaptations are needed, and shown that significant performance gains can be attained compared against a coarse-grained approach. The AspectOpenCOM MOP was presented as a solution to meet the requirements for fine-grained adaptation, providing key operations to inspect all elements of aspects and adapt them e.g. changing the join point set, re-ordering advices and others. Notably, we advocated reflection as a key technology in underpinning the adaptation of deployed aspects.

In future work we plan to investigate the wider application of this technology. Distributed aspects are more likely to be affected by coarse-grained adaptations; this is because reconfigurations times are longer due to safety, security and verification mechanisms typically involving the shutting down (placing in a safe state) of remote components. Hence, unnecessary adaptations must be avoided. For this purpose, we will examine how to introduce fine-grained adap-

tations into distributed aspect composition technologies such as DyMAC [11] and AWED [13].

References

1. G. Blair, G. Coulson, and et al. The design and implementation of Open ORB 2. *IEEE Distributed Systems Online*, 2(6), Sept. 2001.
2. G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. A component model for building systems software. In *IASTED Conference on Software Engineering and Applications (SEA'04)*, Cambridge, MA, USA, Nov. 2004.
3. M. Denker, S. Ducasse, and E. Tanter. Runtime Bytecode Transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures*, 32(2-3):125–139, 2006.
4. R. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison Wesley, 2004.
5. P. Greenwood and L. Blair. Policies for an AOP based auto-adaptive framework. In *Proceedings of the NetObjectDays Conference*, Erfurt, Germany, Sept. 2005.
6. S. Hanenberg, R. Hirschfeld, and R. Unland. Morphing aspects: incompletely woven aspects and continuous weaving. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 46–55, New York, NY, USA, 2004. ACM.
7. JBoss. Jboss AOP. http://labs.jboss.com/jbossaop, last checked Oct. 2007.
8. Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
9. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, Budapest, Hungary, June 2001. Springer.
10. S. Kojarski, K. Lieberherr, D. Lorenz, and R. Hirschfeld. Aspectual reflection. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, 2003.
11. B. Lagaisse and W. Joosen. True and transparent distributed composition of aspect-components. In *Middleware'06*, pages 42–61, November 2006.
12. P. Maes. Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155, Orlando, Florida, United States, 1987. ACM Press.
13. L. Benavides Navarro, M. Sudholt, W. Vanderperren, B. De Fraine, and Da. Suvée. Explicitly distributed AOP using AWED. In *AOSD '06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 51–62, New York, NY, USA, 2006. ACM Press.
14. R. Pawlak, L. Seinturier, L.Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. JAC: an aspect-based distributed dynamic framework. *Software Practice and Experience*, 34(12):1119–1148, 2004.
15. A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect oriented programming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 141–147, Enschede, The Netherlands, April 2002.
16. SpringSource. Spring framework. http://www.springframework.org, last checked Oct. 2007.
17. E. Tanter. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*. PhD thesis, University of Nantes and University of Chile, November 2004.