Augmenting Reflective Middleware with an Aspect Orientation Support Layer

Bholanathsingh Surajbali, Geoff Coulson, Phil Greenwood and Paul Grace
Computing Department,
Lancaster University
Lancaster, UK
{b.surajbali, geoff, greenwop, p.grace} @comp.lancs.ac.uk

ABSTRACT

Reflective middleware provides an effective way to support adaptation in distributed systems. However, as distributed systems become increasingly complex, certain drawbacks of the reflective middleware approach are becoming evident. In particular, reflective APIs are found to impose a steep learning curve, and to place too much expressive power in the hands of developers. Recently, researchers in the field of Aspect-Oriented Programming (AOP) have argued that 'dynamic aspects' show promise in alleviating these drawbacks. In this paper, we report on work that attempts to combine the reflective middleware and AOP approaches. We build an AOP support layer on top of an underlying reflective middleware substrate in such a way that it can be dynamically deployed/undeployed where and when required, and imposes no overhead when it is not used. Our AOP approach involves aspects that can be dynamically (un)weaved across a distributed system on the basis of pointcut expressions that are inherently distributed in nature, and it supports the composition of advice that is remote from the advised joinpoint. An overall goal of the work is to effectively combine reflective middleware and AOP in a way that maximises the benefits and minimises the drawbacks of each.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures – Patterns (Reflection).

General Terms

Design.

Keywords

Reflective Middleware, Dynamic Adaptation, Complexity, Components, Aspect-Oriented Programming.

1. INTRODUCTION

The environments in which distributed applications must operate are becoming increasingly heterogeneous and dynamic, requiring the supporting middleware to adapt its behaviour dynamically to maintain required levels of service to applications [12]. Significant research work has focused on how to make middleware more adaptive; and a commonly-adopted approach to this is the *reflective middleware* approach [18, 27].

Reflective middleware solutions follow a now well-established pattern that typically combines components [31] and reflection [28]. A component-based approach offers modularity, late composition, and good reusability. Reflection then provides the capability for middleware to reason about itself and to act on this reasoning in service of adaptation [23, 28]. To do so, reflective middleware maintains a representation of itself through meta-objects (meta-level) that are causally connected to the underlying system that they describe (base level). Changes made at the meta-level are reflected in the underlying base-level, and vice versa.

However, the reflective middleware approach has two main drawbacks [30]. First, reflective APIs are typically complex and hence impose a steep learning curve on developers [4, 6]. Second, the introduction of unconstrained openness places too much expressive power in the hands of developers. To address these drawbacks, the application of Aspect-Oriented Programming (AOP) techniques to middleware has been proposed [20]. The AOP[1] approach [8, 17] differs from the reflection approach in managing complexity in that it separates cross-cutting concerns (such as distribution, security, persistence etc.), and inherently constrains expressive power by adopting a declarative (as opposed to procedural) emphasis. However, AOP middleware also has its drawbacks. In particular, its declarative approach can restrict the scope of adaptation to relative coarse-grained dimensions, thus losing expressive power. Furthermore, few if any current platforms support the coordinated runtime composition of aspects in a distributed system in a distribution transparent manner.

In this paper, we report on work that combines the reflective and AOP approaches to adaptive middleware in a way that attempts to maximise the benefits and minimise the drawbacks of each. More specifically, our approach is to build an *AOP support layer* on top of an underlying component-based reflective middleware substrate. This enables us to hide the complexity of the underlying reflection layer from programmers. Our AOP support layer can be dynamically deployed when required, and undeployed when further adaptation is not required in the foreseeable future (thus avoiding any overhead when not in use). Another key benefit of our approach accrues from the uniform use of components throughout the architecture (the AOP support layer is constructed from components like the rest of the system). This

[1] In this paper we assume that the reader is already familiar with basic AOP concepts such as aspects, advice, weaving, joinpoints and pointcuts. Please refer to the literature for more information (e.g. [8, 16]).

means that we can advise not only distributed applications, but also the underlying middleware services and the AOP support layer itself, which enables considerable flexibility. A final key contribution of our work is the provision of support for *distributed* dynamic aspects. This means that aspects can be straightforwardly dynamically deployed across a distributed system on the basis of pointcut expressions that are inherently distributed in nature. In addition, we support the composition of advice that is remote from the advised joinpoint.

The rest of this paper is organised as follows. Section 2 provides necessary background on the OpenCOM and GridKit technologies that form the basis of our underlying reflective middleware layer; and section 3 introduces our AOP support layer. Then, section 4 provides an evaluation of our approach, and section 5 discusses open issues and related work. Finally, we offer our conclusions in section 6.

2. BACKGROUND

In this section we provide necessary background on the OpenCOM component model which forms the basis of our work, and on the GridKit reflective middleware platform which is built using OpenCOM. More detail is available in the literature [7, 11].

2.1 OpenCOM

OpenCOM is a lightweight component-based programming technology that is programming-language independent, intended to support the development of low-level systems software (e.g., middleware) as well as applications, and to support runtime configuration and reconfiguration. It employs a minimal runtime kernel that supports the loading and unloading of software components. The elements of the OpenCOM programming model are as follows:

- ***Capsules*** are non-distributed containing entities for *components* that provide a runtime API for component life-cycle management. In particular, the API allows components to be deployed, instantiated, and connected together.
- ***Components*** are language-independent encapsulated units of functionality that are deployed into capsules during run-time. They interact with each other through *interfaces* and *receptacles*. Components may support multiple interfaces and receptacles. We generally use the term 'component' to denote an *instance* of a component as opposed to a type.
- ***Interfaces*** are units of service provision offered by components. To enable programming language independence, they are defined using OMG IDL [16].
- ***Receptacles*** are 'required interfaces' that make explicit the dependencies of a component on other components. Receptacles enable a third-party mode of component composition.
- ***Connectors*** are components that reify the connection of a receptacle and a (type compatible) interface. A receptacle can be connected to only one interface; but an interface may be connected to multiple receptacles. As they are components, connectors may themselves support receptacles and interfaces, thus enabling reflective control over component compositions.

OpenCOM also employs an extensible set of so-called *reflective meta-models* that enables inspection, adaptation and extension of the component composition within a capsule (e.g., of a component composition that represents a middleware platform instance). These can be optionally (and dynamically) deployed whenever required, and undeployed when no longer required. We rely mainly on the following three meta-models:

- The *interface meta-model* supports dynamic discovery of the interfaces supported by a component, and also supports dynamic invocation of operations defined on these interfaces without the necessity for a connector to be in place.
- The *architecture meta-model* represents the current component-compositional topology within a capsule as a 'component graph' that supports operations to insert, update and remove meta-data within the graph, and to alter the composition by manipulating the graph topology.
- The *interception meta-model* supports behavioural reflection [3] by allowing the dynamic insertion of new behaviour (i.e., 'interceptors') at a connector. This relies on a special type of connector called an *interceptor-connector* that supports an interface with operations to manage pluggable interceptors (which are themselves implemented as components).

Figure 1 illustrates the main OpenCOM programming model concepts and the reflective meta-models.

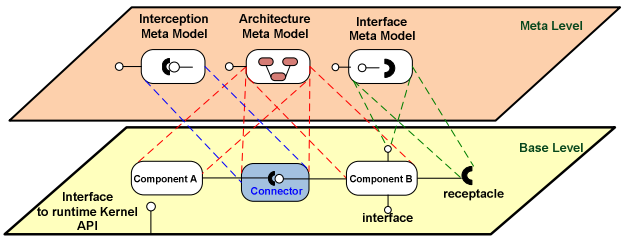

Figure 1: OpenCOM and its reflective meta-models

2.2 The GridKit Middleware Platform

GridKit is a highly configurable reflective middleware platform that is built as a composed set of reconfigurable component frameworks. These are used to extend the basic OpenCOM programming model to support distribution. The distribution machinery is itself built as a component framework, with the underlying networking layer being encapsulated within one or more components. The GridKit approach thus separates the roles of component composition and distribution, OpenCOM providing the former in an exclusively intra-capsule context, while GridKit's frameworks provide inter-capsule communication.

GridKit also provides an additional reflective meta-model to the three fundamental ones described above. This is a *distributed* architecture meta-model [11] that extends the same basic services offered by the per-capsule architecture meta-model to the multi-capsule, distributed, case. Each per-capsule instance of the distributed architecture meta-model maintains a meta-data view of the rest of the system. Consistency between the different instances is maintained using a range of pluggable protocols [9]. More detail on GridKit is available in the literature [13].

3. The AO Support Layer

3.1 Approach and Design Principles

Our basic approach is to structure the AO support layer as a set of components and to layer these over the underlying reflective layer provided by OpenCOM. And where distribution is involved, the AO services are layered over the GridKit frameworks. Before describing the set of components that comprise our AO support layer, we first outline the principles we have applied in our design, and discuss how these map to underlying OpenCOM concepts:

Aspects. To foster conceptual minimality, we realise aspects as regular OpenCOM components: components and aspects are distinguished only by the role they play, not by any fundamental distinction. However, for clarity, we often refer to a component playing an 'aspect' role as an *aspect-component*. It is possible to employ 'legacy' components as aspect-components in cases where the 'non-generic' advice convention is used (see below).

Aspect composition. Aspects are composed with the underlying component composition using the OpenCOM connector concept described above (see section 2.1). In more detail, we use the interceptor-connector employed by the interception meta-model as the basis of aspect composition. A key implication of this is that aspect composition is *non-invasive* – i.e., the encapsulation of an advised component is maintained: it is only possible to advise components at the level of their externally-facing connections.

Advice. As aspects are simply components, advice is realised as operations supported by those components. By using different interceptor-connector variants we are able to support two styles of advice. In the first, 'non-generic', style the advice signature is identical to that of the receptacle and interface being connected by the interceptor-connector. In the second style, a standard *Invoke()* signature is used and arguments are packaged in a generic structure. The first style has less performance overhead, but is less generic than the second style.

Joinpoint model. Because of the simple non-invasive nature of aspect composition, the joinpoint model is correspondingly straightforward. Firstly, we support either *call* or *execution* joinpoints; this distinction is meaningful where multiple receptacles are connected to a single interface – in such a case a call joinpoint picks out the individual receptacles, whereas an execution joinpoint picks out the single interface (in actuality, advices is to be attached to connectors, not receptacles or interfaces; therefore, advising an execution joinpoint where there are multiple connectors attached to an interface means that the advice is attached to *all* of the connectors concerned). Secondly, three types of *advice execution* are supported: 'before', 'after' and 'around' the call or execution of a joinpoint. Thirdly, when an aspect is to be composed with an underlying component composition, the following types of *advice scope* are supported:

- Per-instance – this associates a single aspect-component with *each* instance of the receptacle or interface specified in a pointcut (this is the default advice scope).
- Per-type – this associates a single aspect-component with *all* instances of the specified receptacle or interface type.
- Per-capsule – this associates a single aspect-component with all specified joinpoints within a given capsule (akin to a singleton class per capsule).

Pointcuts. We employ a proprietary programming-language independent pointcut language[2] which is based on work carried out in a collaborative research project [21]. The language is XML based and supports quantification over operation signatures, interface and receptacle signatures, component types and instances, and capsules. In addition, it can quantify over dynamic context properties associated with any of the above. The inclusion of capsules in the list of quantifiable entities means that distribution is inherently supported in a network-independent manner (this is sometimes referred to as 'remote pointcuts' [22]). Pointcuts are used only to identify a set of target joinpoints; to aid reusability, the aspects/advice to be associated with these joinpoints is specified independently using a separate XML-based language.

Distributed AO and Coordination. With the aid of the inherently distribution-aware pointcut language discussed above, distribution is built directly on top of the GridKit distributed architecture meta-model discussed above. Therefore distribution is an integral part of our design. Moreover, a coordination service is used to provide distributed weaving and unweaving of aspects similar to the GridKit [11].

3.2 Architectural Elements

We now describe the architectural elements that comprise our AO support layer (these are also illustrated in Figure 2):

Aspect Manager. The Aspect Manager is the top-level architectural entity and is responsible for dynamically loading, instantiating and removing aspects. It accepts pointcut specifications and associated advice specifications, and interacts with the Pointcut Evaluator and Advice Handler to get the job done. It also caches joinpoint information it receives from the Pointcut Evaluator in case similar pointcuts are submitted in the future. To avoid inconsistency, the Aspect Manager listens to changes in the distributed topology that are notified by the distributed architecture meta-model.

Pointcut Evaluator. The Pointcut Evaluator analyses pointcut specifications forwarded to it by the Aspect Manager, and to looks for matching joinpoint(s) in the specified capsules. To do so, it employs the distributed architecture and the interface meta-models to locate the appropriate joinpoints. Once the list of joinpoints is found it returns these to the Aspect Manager.

Advice Handler. There is an instance of the Advice Handler in each capsule in the distributed system. Its role is to act on instructions from the Aspect Manager to weave advice (using interceptor-connectors) at joinpoints in its capsule. The Advice Handler may additionally need to obtain the specified aspect-components from the Advice Repository (see below).

Aspect Repository. The Aspect Repository (or repositories; the functionality may be distributed) holds a set of instantiable aspect-components.

Interceptor-connector. The basic OpenCOM interceptor-connector mentioned in section 2.1 has been extended so that it can use GridKit endpoints as well as local receptacle and interface pointers. This enables the interceptor-connector to support the invocation of aspect-component instances that are resident in remote capsules.

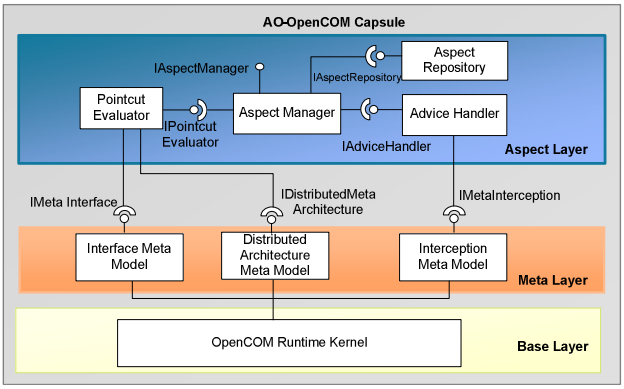

Figure 2: The AO support layer

[2] http://www.comp.lancs.ac.uk/~surajbal/pointcut.html

3.3 Detailed Operation

We now illustrate the operation of the AO support layer. To do so we first introduce a simple 'group messaging' application scenario in which each of the involved nodes (capsules) has Messaging Service and Communication Service components as illustrated in Figure 3.

Figure 3: Group messaging scenario

If a message sender needs to encrypt messages before sending them to other nodes, then an *encryption* aspect is placed between the Messaging and Communication services. To specify such an adaptation, the developer passes the following pointcut and advice specifications to the Advice Manager (figures 4a and 4b). The pointcut specification in Figure 4 specifies two capsules on which the encryption aspect will be applied.

```
<AOOpenCOM-Poincut>
        <Name>Encryption </Name>
        <PointcutKind>Call</PointcutKind>
        <Capsule>
                <CapsuleName>CapsuleHost1 && CapsuleHost2 </ CapsuleName>
        </Capsule>
        <Component>
                <ComponentName>MessagingService* </ComponentName>
        </Component>
        <Interface>
                <InterfaceName>ICommunicationService </ InterfaceName>
        </Interface>
        <Operations>
                <OperationType>* </ OperationType>
        </ Operations>
</AOOpenCOM-Poincut>
```

Figure 4a: Encryption pointcut specification

```
<AOOpenCOM-Composition>
        <Advice>
                <AdviceName>Encryption</ AdviceName>
                < AdviceScope>Per-instance</ AdviceScope>
                <AdviceType>Before</AdviceType>
        </Advice>
</AOOpenCOM-Composition>
```

Figure 4b: Encryption advice specification

The Aspect Manager sends the pointcut specification to the Pointcut Evaluator so that the latter can look for the set of connectors that correspond to the joinpoints specified by the pointcut. This is achieved using the distributed architecture meta-model. The Pointcut Evaluator also queries the interface meta-model to obtain the signatures of the joinpoints. Next, the Aspect Manager checks the availability of the specified set of aspect-components in the Aspect Repository and finally passes all the relevant information (including connector IDs, signatures and aspect-component type IDs) to each concerned Advice Handler.

Next, the Advice Handler obtains the relevant aspect-components from a suitable Aspect Repository, instantiates these, and uses interceptor-connectors to insert them at the identified joinpoints. In case a default connector is currently in use at the joinpoint, the Advice Handler replaces this with a new interceptor-connector[3].

Subsequently, each time a joinpoint condition occurs, the corresponding advice is invoked by the interceptor-connector. Figure 5 illustrates a per-capsule instance of the group messaging example after the encryption aspect has been woven.

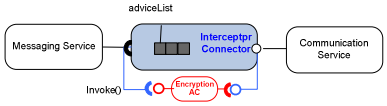

Figure 5: Inserted encryption aspect

[3] To avoid inconsistencies, atomic replacement of the default connector to interceptor is performed.

The whole process is summarised in Figure 6. The figure assumes that advice is placed in the same capsule as the associated joinpoint—i.e., it does not exploit the extended interceptor-connector's ability to invoke remote advice.

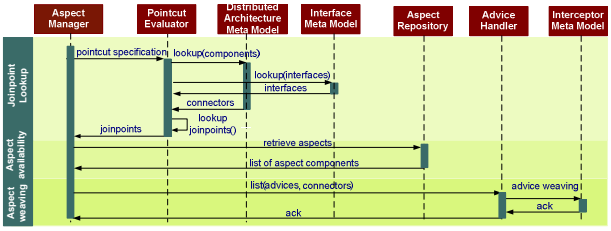

Figure 6: Steps involved in applying new advice

4. EVALUATION

To evaluate our proposal we first show how AO reduces the programmatic complexity seen by the developer. We then evaluate the overheads of the proposal. These are examined in terms of both set-up overhead (i.e., the time taken to dynamically weave a distributed advices) and execution overhead (i.e., the overhead in dispatching aspects when they are encountered at a joinpoint). For set-up overhead we examine the messaging overhead incurred in weaving the encryption aspect in the simple application scenario outlined above. Our benchmark tests are performed on a Pentium IV 3.4 GHZ PC with 1GB memory and running Windows and our AO support layer is implemented using a Java-based version of OpenCOM.

4.1 Programmatic Complexity

To evaluate the programmatic complexity of our AOP approach with respect to the complexity of the underlying reflective layer, we compare the number of lines of developer code required to specify the composition of encryption functionality in the application scenario of Section 3.3. We assume a configuration involving 2 capsules and in which the aspect weaving request is initiated from one of these (the one in which the Aspect Manager is also situated). In such a case, as illustrated in Figure 7, the application developer needs to use only *1* line of code.

```
((IAspectManager) AspectManager).executeAspect("pointcutencryption.xml", "adviceencryption.xml");
```

Figure 7: Deploying distributed advice

Compared to reflective middleware call per capsule, the number of lines of code involved to perform such a reconfiguration is *14*. The number of lines of code increases further if the default connector needs to be replaced by an interceptor-connector.

4.2 Set-up Overhead

In the following, we assume that all of the architectural entities at the top of Figure 6 are in separate capsules (with the exception of the distributed architecture and interface meta-models, and interceptor-connectors, which are present in all the capsules involved). We measure the number of inter-capsule messages involved in the same application set-up as that employed in Section 4.1. Given this set-up, *24* inter-capsule messages are involved. This is a significant number, but, crucially, the number increases only linearly for each additional application capsule involved (4 for each additional capsule). Therefore the design

shows good scaling properties even with a maximally-distributed deployment of the AO support layer[4].

4.3 Execution Overhead

We now analyse the execution overhead of our design in terms of the overheads attributable to invoking advice via our interceptor-connectors. As a baseline, we measure the time taken to invoke an operation in a component C_1 from another component C_2 across a receptacle-interface connection using OpenCOM's minimal 'default' connector. The target operation body is empty and the operation signature takes no parameters. This baseline was measured as 797x10E-9 seconds per call.

Next we measure a $C_1 \rightarrow C_2$ call across an interceptor-connector. The overhead is first measured with no advice installed; that is, the execution path is simply: $C_1 \rightarrow$ interceptor-connector $\rightarrow C_2 \rightarrow$ interceptor-connector $\rightarrow C_1$. The per-call overhead here is 817x10E-9 seconds. Thus the overhead incurred by using a minimally-configured interceptor-connector is a factor of 2.5%.

Next we add a single 'before' advice to the above so that the execution path becomes $C_1 \rightarrow$ interceptor-connector $\rightarrow$ advice body $\rightarrow$ interceptor-connector $\rightarrow C_2 \rightarrow$ interceptor-connector $\rightarrow C_1$. The advice has an empty body and takes and returns no parameters. The per-call overhead here is 834xE10-9 seconds. Thus the overhead incurred over the baseline is a still-low factor of 4.64%.

Finally, we measure again the previous configuration but with varying numbers of advices installed and varying numbers of (int) parameters in the respective advice signatures. The results, shown in Figure 9, demonstrate that the system scales linearly with the numbers of advices and the number of arguments.

Advices	Signatures (No of Parameters) [Time(microseconds)]		
	0	10	100
0	0.817	0.975	1.203
10	4.968	6.31	8.213
100	50.84	63.087	82.084
1000	508.433	631.083	821.992

Figure 9: Performance results with advice

A final point to note is that under normal circumstances the time needed to execute the actual code inside the body of an advice would dominate the overall computational effort, thereby rendering the overhead due to the interceptor-connector basically negligible.

5. DISCUSSION AND RELATED WORK

The essence of our approach is that it adds comprehensive AOP support as an independently-deployable service. This means that the AOP layer imposes no overhead when it not used, and can be dynamically deployed/undeployed where and when required.

In addition, our AO support layer is built using the same programming language independent component-based principles as the underlying reflective middleware layer, and the overlying application. Key implications of this are that the AO support layer:

- is naturally programming language independent (this is in conjunction with the XML-based pointcut language);
- automatically benefits from generic services that come as part of the GridKit middleware – e.g., failure handling, coordination, remote deployment, etc. [11];
- itself can be advised in same way as the rest of the system.

As illustrations of the third point, it is straightforward to apply aspect-based logging to our AO support layer, or to add transactional behaviour to serialise multiple weaving requests that impact common capsules. As another example, rather than relying on topology-change events from the distributed architecture meta-model (see Section 3.2) we could weave a transaction aspect to avoid race conditions such as attempting to adapt a configuration that is already in flux. Depending on factors such as the scale and dynamicity of the environment, a range of alternative concurrency control mechanisms could be envisaged.

Our other main contribution is the development of a fully distributed realisation of dynamic aspects. This is achieved by layering our AO provision on top of GridKit's distributed architecture meta-model, and by providing a pointcut language that is inherently distributed in nature (i.e., it supports quantification over capsules). In addition, we support in a natural way the composition of advice that is remote from the advised joinpoint.

We have also illustrated how our approach tends to maximise the respective benefits of reflective middleware and AO while minimising their drawbacks. In particular, we have demonstrated in Section 4.1 and 4.2 the significantly decreased complexity of deploying new functionality in a distributed environment as compared to the reflective middleware approach. Nevertheless, the lower-level reflective APIs are still available to the developer should they be required. Furthermore, the fact that both layers employ the same reflective meta-models means that conflicts between the two layers can be minimised.

Turning now to related work, [10] contains general arguments for the benefits of combining reflection and AOP. With respect to this, our work can be seen as a case study or exemplar or evaluation of the more abstract perspective given in that paper.

There are a number of systems that take a language-specific (usually Java-based), non-component-oriented, approach to AO-based middleware. For example, *AWED* [2] is an aspect-oriented programming language that features explicit distribution. It is built on top of *JAsCo* [33], a system that provides dynamic aspects for Java. Aspects are represented as plain Java objects that can be dynamically deployed. Unlike our approach, AWED does not treat distribution with full orthogonality; distributed joinpoints are specified differently from intra-node joinpoints. *ReflexD* [32] also employs a Java-based approach, as does *Modelware* [34]. *JBOSS AOP* [5] and Spring AOP [15] are two commercial AOP platforms that have integrated the AOP approach to middleware. However, neither of the approaches provides dynamic distributed aspect composition.

A number of other systems, like ours, take a component-based approach to AO-based middleware. *Fractal Aspect Components* (FAC) [25], is an aspect-oriented extension of the *Fractal Component Model* [29]. However, FAC currently supports only local node reconfiguration as it is not layered over a distributed infrastructure. In addition, because FAC is based on Java and uses *AOP-Alliance* [1] interceptors, it employs an invasive style of aspect weaving, unlike our non-invasive

[4] Note, however, that further, albeit linearly-scalable, messaging overhead is incurred by the underlying distributed architecture meta-model. This is difficult to quantify as it may involve different communication mechanisms depending on how it is configured. Note also that the distributed architecture meta-model is a generic service and may be used by a number of other sub-systems apart from the AO support layer.

approach. This adds greater expressive power but at the cost of programming language independence (and, arguably, excessive expressiveness). *JAC* [24], *CAM/DAOP* [26] and *DyMac* [19] are other component-based approaches that take a more principled approach to distribution. However, none of these systems straightforwardly supports distribution-transparent dynamic distributed (un)weaving. Moreover, *JAC* does not offer remote advices, instead common advices are replicated across each host.

One area in which a number of systems focus is distributed consistency, and ensuring consistency during distributed reconfiguration. For example, *AWED* provides a consistency protocol to ensure that whenever aspects are deployed on a host, the same aspects are also applied at the other involved hosts to preserve system-wide consistency. *ReflexD* is similar.

CAM/DAOP also supports coordination. Although, this is currently fixed as a core part of the middleware, investigation on how to separate out the coordination aspect is underway. Our work is less developed in the coordination area, currently relying on the GridKit reconfiguration principle [11] but shows considerable potential because of the above-discussed fact that the AO support layer itself can be straightforwardly advised—in this case by a putative coordination aspect. Thus in our approach, coordination is potentially a configurable matter.

6. CONCLUSION AND FUTURE WORK

We have summarised the benefits of our approach in Section 5. Our AO support layer is implemented and we are currently testing it with a range of application scenarios.

There are two main future research directions that we would like to pursue. First, as discussed above, we would like to experiment with different coordination styles for the AO support layer. A related issue is that of aspect conflicts, such that newly-introduced aspect behaviour may conflict with other existing behaviour in the system and bring the system to an inconsistent state. A similar issue arises with respect to dependencies: a given aspect may require the presence of another aspect to be able to properly fulfill its role. Note, however, that this issue is relatively independent of the particularities of our design, and could benefit from approaches under investigation in other projects (e.g., see [14]). Second, we plan to use our AO reflective middleware to investigate the use of distributed AO-based adaptation in resource-scarce environments such as mobile ad-hoc networks and sensor networks. This will demand a notion of context-aware adaptation of aspects, such that aspects are applied according to the resources found in the distributed topology in which they are deployed.

7. REFERENCES

[1] AOP-Alliance, http://sourceforge.net/projects/aopalliance,07.

[2] Benavides, L., Sudholt, M., et al., "Explicitly distributed AOP using AWED", *5thInt. ACM Conf. AOSD*, March 2006.

[3] Blair, G., et al., "The Design and Implementation of OpenORB v2", *IEEE DS Online Reflective Middleware,* '01.

[4] Bouraqadi, N. and Ledoux T., Aspect-Oriented Software Development, *Chapter 12 -Supporting AOP using Reflection,* pages 261-282. Addison-Wesley, 2005.

[5] Burke, B., "JBoss AOP Tutorial", *3rd International Conference on AOSD,* Lancaster UK, 2004.

[6] Colyer, A., et al., "Managing Complexity in Middleware", *Patterns for Infrastructure Software,* AOSD, 2004.

[7] Coulson, G., Blair, G, Grace et al., "A Component Model for Building Systems Software", *Proc. IASTED SEA, USA,* '04.

[8] Filman, R., Elrad, T., Clarke, S., and Aksit, M., *Aspect Oriented Software Development*, Addison Wesley, 2004.

[9] Ganesh et al., "SCAMP: Peer-to-peer lightweight membership service for large-scale group communication". *Workshop on Networked Group Communication*, UK, 2001.

[10] Grace, P., et al., "The Case for Aspect-Oriented Reflective Middleware*", In Proc. of the 6th International Workshop on Adaptive and Reflective Middleware*, Nov 2007.

[11] Grace, P, Coulson, et al., "A Distributed Architecture Meta Model for Self-Managed Middleware", *ARM* 2006.

[12] Grace, P., Blair, G., "Reflective Middleware, In Handbook of Mobile Middleware", CRC Press, 2006.

[13] Grace, P., et al, "GridKit: Pluggable Overlay Networks for Grid Computing",*In Proceedings of DOA*, October 2004.

[14] Greenwood, P., et al., "Interactions in AO Middleware", *Proc. Workshop on ADI,* ECOOP 2007.

[15] Harrop, P., Colyer, A., "AOP in Spring", *AOSD 2005.*

[16] "IDL to Java Language Mapping Specification," *The Object Management Group*, www.omg.org, 2007.

[17] Kiczales, G., Lamping, J., et al., "Aspect Oriented Programming", *Proceedings of ECOOP,* 1997.

[18] Kon, F., Costa, et al, "The Case for Reflective Middleware: Building middleware that is flexible, reconfigurable, and yet simple to use", *CACM Vol 45, No 6*, 2002.

[19] Lagaisse, B. and Joosen, W., "True and Transparent Distributed Composition of Aspect-Component", *Middleware Conference, 2006.*

[20] Loughran, L., et al, "*Survey of Aspect-Oriented Middleware*", AOSD-Europe Deliverable D8, June 2005.

[21] Loughran, N., et al "Requirements and Definition of AO Middleware",*AOSD-Europe, Project Deliverable*, Aug 2005.

[22] Nishizawa, M., et al., "Remote Pointcut A Language Construct for Distributed AOP", *AOSD Conference,* 2004.

[23] Maes P., "Concepts and Experiments in Computational Reflection", *Proceedings of OOPSLA* ACM Press, 1987.

[24] Pawlak, R., Seinturier, L., "JAC: A Flexible Solution for AOP in Java". *In Proc. Reflection* 2001.

[25] Pessemier, N., Seinturier, et al., "Component-based and Aspect-oriented Systems", *Conf. Software Composition*, '06.

[26] Pinto, M., et al., "A Component And Aspect based Dynamic Platform". *The Computer Journal, 2005*.

[27] Sadjadi, M. and McKinley, P.K, "*A survey of adaptive middleware*". Technical Report MSU-CSE-03-35, 2003.

[28] Smith, B., "*Reflection and Semantics in a Procedural Programming Language*", PhD thesis, MIT, January 1982.

[29] Stefani, J.,et al., "Fractal Component Tutorial" *ECOOP,* '06.

[30] Sullivan, G., "Aspect-oriented programming using reflection and meta-object protocols". *Comm. ACM Vol. 44,* 2001.

[31] Szyperski, C., "*Component Software: Beyond Object-Oriented Programming*". Addison-Wesley, 1999.

[32] Tanter, E., "*From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*", PhD Thesis - University of Nantes, November 2004.

[33] Vanderperren et al., "A visual component composition environment with advanced aspect separation features", *Conference on FASE Poland*, 2003.

[34] Zhang, C., et al., "Generic Middleware Substrate through Modelware". *6th Middleware Conference,* France 2005.