The Case for Aspect-Oriented Reflective Middleware

Paul Grace*, Eddy Truyen, Bert Lagaisse, Wouter Joosen
Department of Computer Science, K. U. Leuven, Belgium
{Paul.Grace, Eddy.Truyen, Bert.Lagaisse, Wouter.Joosen}@cs.kuleuven.be
* On Leave from Lancaster University

ABSTRACT
The emergence of applications domains such as pervasive and autonomic computing has increased the need for customisation and dynamic adaptation of both distributed systems, and the underlying middleware platforms. Two highly complementary technologies have been advocated to meet these challenges, namely: aspect oriented programming (AOP) and reflective middleware. However, these have so far been considered in isolation, or typically target a particular middleware challenge e.g. using aspects to customise a middleware implementation; or using reflection (or dynamic AOP) to alter runtime behaviour. We believe that in combination these technologies better support the engineering of dynamic distributed systems. In this paper, we explore how aspects and reflection have been utilised in both the programming language and middleware communities; building upon this work, we identify four core relationships that form the basis of our model for aspect-oriented reflective middleware. We then explore the potential of this model to i) increase support for the engineering of dynamic reconfigurations, and ii) improve the performance of adaptive systems.

Categories and Subject Descriptors
C.3.11 [**Software Architectures**]: Software Architectures - Patterns (Reflection)

General Terms
Design

Keywords
Aspect oriented programming, reflection, middleware

1. INTRODUCTION
Engineering distributed systems is becoming increasingly complex in domains where diversity and dynamic adaptation are central elements. There is a need to customise middleware, and middleware services to individual deployment domains e.g. in real-time and pervasive settings. Furthermore, it is clear in modern distributed systems that the operational environment, application requirements or general context may alter over time e.g. in autonomic or mobile applications; hence, enhanced support for adaptation is a central requirement.

ARM'07, November 26, 2007, Newport Beach, CA

Reflection and aspect-oriented programming (AOP) are two approaches that have been utilised to support developers in overcoming these challenges. Reflection provides introspection and adaptation of a wide range of system concerns (e.g. the component architecture, or resource usage); whereas AOP supports the composition and adaptation of cross-cutting system behaviour (e.g. security or persistence). Although the two are highly complimentary [1], they have typically been utilised in isolation for adaptive systems, or combined at different stages of the development lifecycle e.g. using aspects to customise reflective middleware at design time [2], using aspect weaving at compile time to make systems adapt-ready [3][4], or layering dynamic aspects atop reflection meta-object protocols (MOP) [5][6]. This demonstrates that the two approaches benefit one another; we wish to explore these benefits further by combining reflection and aspects at runtime. For this, we extend the multi-model approach (to meta-space) of reflective middleware [7] to include an *aspect MOP*; we term middleware built upon this model *aspect-oriented reflective middleware* (AORM).

We demonstrate in this paper, that aspect-oriented reflective middleware offers the following important benefits:

- The ability to perform *fine-grained* introspection and dynamic adaptation of aspects (using the aspect MOP), not supported in state of the art dynamic AOP systems [8]. This includes the ability to adapt or re-order advice behaviour, and importantly reconfigure the joinpoint set (i.e. where the aspect is deployed). Hence, self-adaptation and system wide validation of crosscutting concerns is supported.
- The provision of multiple system viewpoints to better support complex adaptations; each MOP manages adaptation of a system concern e.g. the architecture MOP manages component adaptation; the aspect MOP manages cross-cutting module adaptation; and the resource MOP manages resource usage adaptation.
- Increased system performance by reducing the overheads incurred by reflection. Aspects are used to deploy reflection only where required (c.f. partial reflection [9]).

We acknowledge that existing aspect and reflection solutions support some of these features; but we believe the combination of reflection and aspects offer a more complete, principled solution.

The paper is organized as follows. Section 2 analyses the existing research in the area of aspects and reflection; it explores work from both the programming language and middleware community, and identifies the core relationships between aspects and reflection. Section 3 then presents the key contribution of this paper, which is the aspect-oriented meta-space for dynamic middleware platforms; this follows closely from the principles

identified in section 2. Finally, section 4 draws conclusions and identifies a roadmap for future research.

2. AOP AND REFLECTION

2.1 Background

AOP [10] is a software engineering approach designed to tackle the problems of tangled code i.e. the basic functional implementation of your component becomes tangled with additional code for features such as security, persistence, logging, and monitoring. Developers often implement these features in an ad-hoc manner across the system, which leads to increased system development, debugging, and evaluation time because of the increased system complexity. Therefore, AOP supports the concept of separation of concerns to counter this problem; i.e. individual concerns such as security and monitoring code are not implemented within the base code, rather these are each implemented as an individual *advice*, which is a piece of code that can then be woven into the base code at compile time. Developers express *pointcuts*, which identify positions in the code (*joinpoints*) where these advices should be attached. Dynamic AOP promotes the same benefits as AOP, but the aspects are woven at run-time rather than compile time. In this paper, we focus of pointcut based AOP; other forms include composition filters and hyperslice approaches.

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 [11]. In middleware platforms, 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 arrival and dispatching of invocations.

Table 1. Comparison of AOP and reflection

Technology	Cross cutting	Adaptation	Self-Adaptation
Reflection	Poor	General	Yes
AOP	Strong	Aspects only	No

We now compare the two approaches (illustrated in table 1). Both support separation of concerns, where an aspect or MOP implements each concern. However, concerns that crosscut the base level are more easily applied using aspects (reflection can be used to manage crosscutting behaviour, but this becomes increasingly complex in large-scale systems). Similarly, both approaches support adaptation of system behaviour; however, in dynamic AOP only aspects can be added and removed, whereas reflection supports a wider range of adaptation types (e.g. component or resource adaptation). Furthermore, reflection supports self-awareness, so a system can base its adaptation decision on its current status; a capability not available in AOP. From table 1 it is clear that both technologies can benefit from the other; e.g. improving the management of crosscutting behaviour in reflective systems, or building self-adaptive aspect-based systems; in the following sections we examine the extent to which reflection and aspects have been successfully combined, both in programming languages and middleware solutions.

2.2 Uniting Aspects and Reflection in Programming Languages

Research from the programming language community has investigated the relationship between aspects and reflection; indeed the original work into AOP was inspired from MOPs [10], with AOP being seen as a principled subset of reflective programming. Sullivan [1] first identifies the complex nature of programming reflective systems (*"too much rope"* for the developer), and secondly states that reflection consumes too much overhead to be a worthwhile technology. He then promotes AOP languages as a means to tame the complexity and reduce overhead; the key contribution is the use of aspects as an interface to the functionality of MOPs.

Tanter [9] similarly advocates the use of cross-cutting techniques to reduce the expense of reflection. That is, he identifies that MOPs typically reify every object in the system, however the majority of these meta-object are rarely used; this increases memory costs, and adds unnecessary levels of indirection (as invocations pass through the meta-level). Hence, only locations that need to be reflected on are reified; this is known as *partial reflection*. An aspect-oriented approach is used to define where the MOP is added.

Alternatively, Kojarski et al., [12] explore the two-way relationship between aspects and reflection; they argue that AOP is another computational reflection mechanism, where a joinpoint 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).

2.3 Aspects and Middleware

2.3.1 Customising Middleware

Middleware technologies are typically deployed in multiple environments, where a one-size fits all approach results in unnecessary implementation. Hence, aspects have been used to modularize crosscutting middleware functionality, so that evolution and customisation of the middleware is straightforward. The following technologies apply aspects at compile time. [13] modularises the crosscutting concerns of a CORBA ORB. Similarly, [2] identifies that in reflective middleware, reflection crosscuts core middleware functionality; hence, aspects are used to customise the reflective MOPs. Finally, Demir et al. [14] present an aspect-oriented IDL that allow developers to insert application-level behaviour lower in the middleware stack, bypassing unnecessary layer processing, e.g. performing security checks on an object method invocation at the socket layer.

Aspects are also used to make systems adapt-ready; this is similar to partial reflection [9]. System locations that need to be adapted at a later stage have reflective MOPs added to them. Trap/J [3] uses aspects at compile time, to create meta-sockets that have a behavioural MOP for dynamic insertion of interceptors. Similarly, [4] advocates a two-stage process for developing

adaptive systems, namely using aspects at compile time to weave run-time reflective mechanisms into the system.

2.3.2 Dynamic Aspects

Dynamic AOP supports runtime weaving of aspects; hence, crosscutting modules can be reconfigured at runtime. A number of dynamic AOP tools have been developed; these typically vary in how aspects are weaved (e.g. efficient bytecode rewriting, dynamic proxies, etc.), when aspects are weaved (load-time e.g. AspectWerkz [15] or run-time e.g. JBoss [16]), and where aspects are weaved. [17] describes three styles of location weaving: i) total hook weaving (where a hook is a location where an advice is woven) augments every location in the code with a hook, ii) actual hook weaving weaves hooks only to locations of interest, and iii) collected weaving where actual advice code is placed instead of hooks (reducing indirection). Total hook weaving is the most flexible, and most expensive; while collected weaving is least flexible, but has the best performance.

Further examples of Dynamic AOP middleware are Dymac [18], MIDAS [19] and JAC [20]. Dymac provides a remote pointcut approach for deploying application specific aspect behaviour (e.g. application logging, or authentication) across remote hosts at run-time. MIDAS is a middleware layer underpinned by a dynamic AOP system (Prose [8]). MIDAS adds functional extensions to the developer's basic code implementation at run-time. When required, the extension is downloaded to the MIDAS middleware, which then dynamically weaves the code into the base application at run-time.

From our initial analysis of dynamic AOP systems, the majority provide *coarse grained adaptation* of aspects, namely the whole aspect (made up of pointcut description and advice implementation) can be added and removed. However, there is increasing need to make informed decisions about deployed aspects e.g. discovering the conflict issues between advice ordering and adapting accordingly [21]. Hence, we believe that introspection and fine-grained adaptation of aspects is a fundamental requirement. In current systems, Prose, JAC and JBoss provide fine-grained adaptation of the chain of advices that execute at joinpoints. JAC provides policies to resolve advice conflicts at runtime. Similarly, Prose provides an API to discover information such as the list of all system joinpoints, or the list of joinpoints related to a pointcut. However, these are ad-hoc approaches to fine-grained adaptation that can be improved through the use of a principled aspect MOP with richer facilities.

2.4 Reflective Middleware

Example middleware technologies that leverage reflection are: the work at Lancaster University [7], DynamicTAO [22], and Arctic Beans [23]. These systems use reflection to principally configure and reconfigure the behaviour of the middleware. For example, platforms can be tailored to support domain specific applications in heterogeneous environments; or the middleware can adapt its behaviour based upon changing context e.g. adapting a streaming binding in fluctuating QoS conditions. An important feature of reflective middleware is the separation of concerns provided by multiple meta-object protocols. For example, a system can be separated into its component architecture, and resource use; this allows adaptations to be made from either viewpoint. This contrasts with dynamic AOP where only aspects can be adapted.

There are examples were reflective middleware has utilised the potential of aspects. We have already discussed how aspects can deploy partial reflection [3]. Further, Arctic Beans has investigated the role of aspects to deploy security and transaction behaviour in the middleware. Alternatively, Rasche [24] defines a reconfiguration aspect that manages adaptation and hides the complexity of reflection from the developer. Finally, [6] describes how aspects can be layered on top of traditional meta-object protocols to constrain the complexity of reflective programming by supporting aspect composition, and hence presents a concrete example of part of the abstract model presented in this paper. However, no system treats aspects as a modular concern, which can be adapted in a similar manner to dynamic AOP.

2.5 Analysis

It is clear from this considerable body of work that together aspects and reflection have an important role to play in modern middleware. From this research we have identified four important relationships between aspects and reflection that can be leveraged in middleware to improve support for the engineering of dynamic distributed systems.

1. *Reflection can be applied to fully support self-aware adaptation of aspects, and the fine-grained adaptation of cross-cutting behaviour*. No reflective middleware or dynamic AOP system provides this capability.
2. *Dynamic aspects can be added to systems by leveraging the facilities provided by existing MOPs.* This is fairly common in programming languages; however, only [5][6] has investigated this in component systems, and at run-time.
3. *The overheads of reflection can be reduced by using aspect approaches to deploy MOPs where required*.
4. *Reflection is complex; aspects provide a principled subset of reflective programming to tame this complexity*.

Although initial work in this area has shown promising results; the solutions are either localised to individual relationships or focus on aspects or reflection in isolation. Similarly, many of the approaches apply aspects and reflection at different stages of the development lifecycle. Hence, we believe that further research is required to investigate how to apply them together at run-time in order to meet the requirements of highly-adaptive and autonomic systems.

3. AO REFLECTIVE MIDDLEWARE

3.1 The Core Model

The *Aspects and Reflection Meta Model* (figure 1) is an extension of the Lancaster multi-model approach [7]; every application level component offers a meta-space consisting of a set of distinct meta-models. Our extension combines (at the meta-level) the traditional reflective MOPs (i.e., architecture, interface, resource and interception), with a novel aspect-oriented MOP. A key benefit of this model over prior reflective and aspect systems is that it provides multiple viewpoints for adaptation of components, resources, interceptors, *and crosscutting concerns* (aspects). This model also supports at runtime the relationships identified in section 2.5 (the first three of the following are explored further in subsequent sections).

- The Aspects MOP supports fine-grained introspection and adaptation of cross-cutting behaviour.

- Aspects can be added to a system at run-time using the Aspect MOP as the implementation of this MOP is underpinned by existing reflective MOPs.
- The interception MOP can be deployed dynamically at specified locations using an aspect whose pointcut reacts to structural reflection events (e.g. component creation).
- Reconfiguration aspects can be deployed using the Aspect MOP that abstracts over reflective MOPs.

So far we have implemented this model within the OpenCOM platform (The implementation is named AOpenCOMJ and is available at http://gridkit.sourceforge.net); we plan to also implement the model in a dynamic aspect-oriented middleware to demonstrate wider applicability.

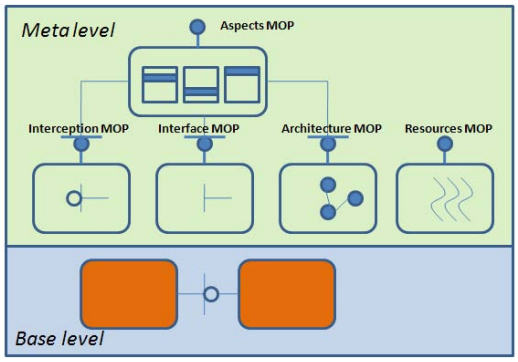

Fig 1. The Aspects and Reflection Meta Model

3.2 Fine-grained Aspect Adaptation

3.2.1 The Aspect Meta-Object Protocol

The aspect MOP introspects and adapts cross-cutting behaviour of the base component system. Hence, aspects are modules that apply across functional components; they have two core elements:

- the *joinpoint set* (where the aspect is applied in the system) described by a pointcut expression. Joinpoints can describe: traditional binding execution points e.g. receptacle call and interface call; component creation/delete/connect events; interface creation; resource change etc. That is, events from both the base and meta-level. Hence, our aspect model is extensible to new system behaviour.
- The *advice set*; i.e. the aspect module's behaviour implementation. We use generic advices (individual operations similar to traditional interceptors) that can be either pre, post or around behaviour. These are deployed in an ordered execution chain at each joinpoint.

```
...
List<AspectMeta> enumAspects()
List<Advice> enumAdvices(Joinpoint jp)
Boolean replacePointcut(AspectID a, Pointcut p)
Boolean addAdvice(Pointcut p, Advice av)
Boolean reorderAdvices(Joinpoint jp, List<Advice>)
...
```

Fig 2. Sample operations in the aspect MOP

The meta-level represents and adapts these base-level elements; this allows such behaviour as listing all aspects in operation (and more specifically the pointcuts and advices that compose them) which is important for: informing future adaptation decisions, verification and tracing of system behaviour against requirements, and identification of and resolution of interactions between deployed aspects [25].

Figure 2 describes a sample set of operations available from the MOP. Example introspection operations are *enumAspects* and *enumAdvices*; these return metadata describing information about the aspects currently deployed. The first describes the full information about the aspect's pointcut and its advice list, the second lists all behaviours (potentially from multiple aspects) at an individual joinpoint in the base. The MOP also includes operations for fine-grained dynamic adaptation. The *replacePointcut* operation allows the developer to pass a new pointcut expression and the existing aspect behaviour will be moved from the prior joinpoint set to the new joinpoint set. *addAdvice* adds new advice code to locations identified by the pointcut description; finally, *reorderAdvices* takes the new advice order for a joinpoint and adapts the behaviour accordingly.

3.2.2 Use Cases

To motivate the requirement for fine-grained adaptation of aspects using the aspect MOP we present following use cases.

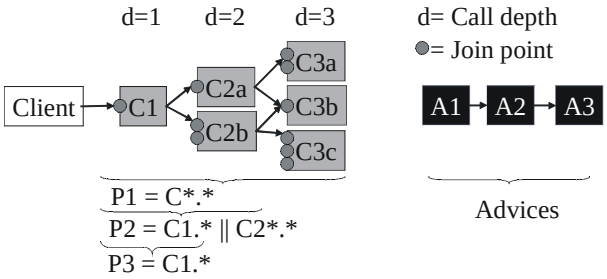

Fig 3. Joinpoint set adaptation

Consider a layered set of components as depicted in figure 3. A monitoring aspect applies a set of advices to create a trace of the call flow (for development purposes). Depending upon the load of the system (which can be discovered using the resource MOP), the call flow depth is determined: at high load, only the called operations on the façade (C1) are traced (using pointcut P3). When the load is lower for the system, a deeper trace is created (e.g. pointcut P2 selects all joinpoints up to a 2-layer depth, pointcut P1 selects all joinpoints up to a depth of 3 layers. Our aspect MOP then allows us to switch at runtime between those 3 pointcuts (e.g. using the replacePointcut operation), while keeping the runtime state of the aspect. This results in a performance gain by avoiding unnecessary interception and joinpoint reification. Without such an operation, the advice would be applied to all joinpoints; this advice would need to keep track of the depth, and decide whether to create the trace for that depth. When the time to intercept a joinpoint, reify it and activate the advice is of the same order of magnitude (or greater) than the execution time of the tracing advice, the performance gain becomes significant. In general, for advising a fluctuating set of joinpoints, the aspect MOP will offer performance gains as well as ease of composition compared to the state-of-the-art in dynamic AO and pure reflective middleware.

In the second use-case, when a new aspect must we woven into an existing aspect composition at runtime, a reordering of

advices at a shared join point may occur. Consider a client-server system with authentication, caching, logging, and encryption aspects [21]. Initially no aspects are 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 if 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. When aspects execute at the same join point, the order in which their respective advices are executed may be 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. As such, the only correct way is that the authentication advice executes before the cache advice. In AO frameworks, such as JBoss AOP, the order of advice execution is determined by the order in which aspects are added to the system. As a result, weaving the authentication aspect after the cache aspect has been woven yields the wrong execution order. The aspect MOP allows us to inspect the state for verification, and dynamically insert advices into particular positions of an existing advice chain to resolve such issues (using the reorderAdvices operation).

3.3 Dynamic AOP atop Reflective MOPs

As advocated in prior research [1], we leverage traditional reflection MOPs to add dynamic aspects within the model. Figure 1 shows that the aspect MOP can depend on the following three meta-object protocols.

- The interface meta-model supports inspection of a component's interfaces. Typically, you can examine the operations available on these interfaces, and or dynamically invoke one of the operations. The aspect MOP uses these operations to discover interfaces (and/or methods) that match a pointcut expression to form a given joinpoint.
- The architecture meta-model accesses the software architecture of a component represented by a component graph. Hence, the architecture meta-model can be used to both discover and change this structure at run-time. The aspect MOP uses the introspection operations of this MOP to discover components who match a joinpoint expression.
- The interception meta-model enables the dynamic insertion of interceptors. The aspect MOP fully utilises introspection and adaptation operations to apply and re-order advices using behavioural interceptors.

Note, this is one configuration of the meta-level; however, as the meta-level is itself implemented as components we can produce more flexible dependencies between MOPs; this is demonstrated in the following section to produce partial reflection behaviour at runtime.

3.4 Exploring Partial Reflection at Runtime

Tanter [9] applies partial reflection at compile time. Here, we present initial experiments into how dynamic aspects can be used to apply partial reflection to a running system. The traditional meta-space applies per composite component. Hence, you can compose the entire system as a single composite with a corresponding meta-space. However, this will reify the entire system with every MOP incurring expensive overhead. Alternatively, you can compose a system of multiple composites, each with distinct meta-space configurations. This is illustrated in figure 4; an initial composite of composites A and B has a full meta-space, whereas A has no meta-space, and B an interception MOP. Akin to prior research we use aspects at run-time to create such partial reflection systems; here aspects deployed in the base composite build the MOPs in the sub-composite.

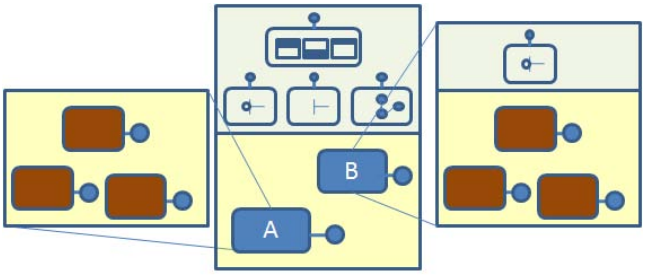

Fig 4. Using aspects for partial reflection

To illustrate this we describe how aspects are used to build the meta-space for composite B in figure 4. In this case, we wish to only apply an interception MOP, and furthermore we wish to tailor this further to ensure that delegators (advice proxies that code can be dynamically added around) are only attached to interfaces of a specific type; this reduces the indirection in the system (as invocations on components in B won't go through a delegator). For this, we define an aspect in the base composite whose pointcut locates the component (with a particular interface) creation event in composite B; on this "joinpoint match" an advice executes code to build (or add to) the MOP in B's meta-space. This is one example, but similar strategies can be employed to tailor MOPs (and indeed Aspect MOPs) to ensure optimization.

Table 2. The benefits of removing unnecessary indirection

Gridkit Configuration	Throughput - Original MOP (Msg/sec)	Throughput -Aspect MOP (Msg/sec)
CORBA client	2352	2399
Group	3708	4585
Publisher	2623	2844

To illustrate the potential benefits of this approach; we took an existing middleware (Gridkit [26]), which applies a full set of MOPs (architecture, interface and interception) to every component in the middleware. The attachment of delegators to every interface is a non-optimised solution. Interceptors are only utilised in a small percentage of interfaces; hence, in the majority of situations an additional level of indirection is unnecessary. We used an aspect to apply the interception MOP only where required and then compared existing configuration behaviour using the original full MOP against our aspect approach. Table 2 shows the results; for the publisher and group configurations there is an approximate 8% and 24% increase in performance respectively. The CORBA implementation consists of a less complex component configuration (in terms of components and connectors) and there is less impact from indirection; hence, there is only a small improvement. The group configuration is the most complex. Hence, this shows that fine-grained compositions with

frequent calls between components are particularly suited to this engineering improvement,

4. CONCLUDING REMARKS

In this paper we have identified the potential of combining aspects and reflection in middleware systems to increase support for the development of dynamic distributed systems. Our two key contributions are: i) an aspect MOP that supports fine-grained inspection and adaptation of cross-cutting concerns, and ii) an extension to the multi-model meta-space that considers aspects as another adaptation concern. We have performed initial implementation and experimentation in the Lancaster family of middleware, and demonstrated that early results are promising.

However, this remains work in progress; and further applications and experimentation in real systems is required to fully illustrate the power of this approach. Hence, in our roadmap of future research, we plan to apply the model to the Dymac aspect middleware, and compare directly the development of real world dynamic systems with both traditional dynamic AOP and reflective approaches.

5. REFERENCES

[1] G. Sullivan. Aspect Oriented Programming Using Refelection and MetaObject Protocols. Communication of the ACM, 44(10):95-97. October 2001.

[2] N.Cacho, T.Batista. Using AOP to Customize a Reflective Middleware. Int'l Symposium on Distributed Objects and Applications, pp. 1133–1150, Agia Napa, Cyprus, Nov 2005.

[3] Z. Yang, B. Cheng, R. Stirewalt, J. Sowell, S. Sadjadi, P. McKinley. An Aspect-oriented Approach to Dynamic Adaptation. ACM SIGSOFT Workshop On Self-healing Software (WOSS'02), pp. 85-92, Charleston, SC, Nov 2002

[4] P. David, T. Ledoux, N. Bouraqadi-Saadani. Two-step Weaving with Reflection using AspectJ. OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems. Tampa, USA, October 2001

[5] N. Bencomo, G. Blair, G. Coulson, P. Grace, A. Rashid. Reflection and Aspects Meet Again: Runtime Reflective Mechanisms for Dynamic Aspects. Middleware '05 Workshop on Aspect Oriented Middleware Development (AOMD 05). Grenoble, France, November 2005.

[6] B. Surajbali, G. Coulson, P. Greenwood, P.Grace. Augmenting Reflective Middleware with an Aspect Orientation Support Layer. 6th Workshop on Adaptive and Reflective Middleware, Orange County, CA, Nov 2007.

[7] G. Blair, G. Coulson, et al. The Design and Implementation of OpenORB v2. IEEE DS Online, Special Issue on Reflective Middleware, Vol. 2, No. 6, 2001.

[8] A .Popovici, T. Gross, G. Alonso. Dynamic Weaving for Aspect Oriented Programming. 1st International Conference on Aspect-Oriented Software Development (AOSD), pp. 141-147. Enschede, Netherlands, April 2002.

[9] E. Tanter. From Metaobject Protocols to Versatile Kernels for Aspect Oriented Programming. Ph.D. Thesis, University of Nantes, France, 2004.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Vieira Lopes, J. Loingtier, J. Irwin. Aspect Oriented Programming. ECOOP'97. pp. 220-242. Jyväskylä, Finland, June 1997.

[11] P. Maes. Concepts and Experiments in Computational Reflection. OOPSLA'87, Vol. 22 of ACM SIGPLAN Notices, pp. 147-155, ACM Press, 1987.

[12] S. Kojarski, K. Lieberherr, D. Lorenz and R. Hirschfeld. Aspectual Reflection. AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies. Boston, Massachusetts, March 2003.

[13] C. Zhang and H. Jacobsen. Refactoring Middleware with Aspects. IEEE Transactions on Parallel and Distributed Systems, 14(11):1058 - 1073, November 2003.

[14] O. Demir, P. Devanbu, E. Wohlstadter and S. Tai. An Aspect-oriented Approach to Bypassing Middleware Layers. Proceedings of Aspect-Oriented Software Development (AOSD), pp. 25-35. Vancouver, Canada, March 2007.

[15] J. Boner and A. Vasseur. AspectWerkz Web Site, http://aspectwerkz.codehaus.org, July 2007.

[16] JBoss AOP. http://labs.jboss.com/jbossaop/ July, 2007

[17] R. Chitchyan, I. Sommerville. Comparing Dynamic AO Systems. Proceedings of the AOSD'04 Dynamic Aspects Workshop, pp. 23-36, Lancaster UK, March 2004.

[18] B. Lagaisse and W. Joosen. True and Transparent Distributed Composition of Aspect-Components. Middleware'06, pp. 42-61. Melbourne, Australia, November 2006.

[19] A. Popovici, A. Frei, G. Alonso. A Proactive Middleware Platform for Mobile Computing. Middleware '03, pp. 455-473. Rio de Janeiro, Brazil, June 2003.

[20] R. Pawlak, L. Seinturier, L. Duchien & G. Florin. JAC: A Flexible Solution for Aspect-oriented Programming in Java. Proceedings of Reflection'01, pp. 1-24. Kyoto, Japan, 2001.

[21] P. Greenwood, L. Blair. Policies for an AOP Based Auto-Adaptive Framework. NetObjectDays Conference, Erfurt, Germany, September 2005.

[22] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. Magalhães, R. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. Middleware'2000, pp. 121-143, New York, April 2000.

[23] A. Andersen, G. Blair, V. Goebel, R. Karlsen, T. Stabell-Kulø, W. Yu. Arctic Beans: Configurable and Re-configurable Enterprise Component Architectures. IEEE Distributed Systems Online, Vol. 2, No. 7, 2001.

[24] A. Rasche, W. Schult, and A. Polze. Self-Adaptive Multithreaded Applications - A Case for Dynamic Aspect Weaving. 4th Workshop on Adaptive and Reflective Middleware (ARM 2005), Grenoble, France, Nov 2005

[25] F. Sanen, E. Truyen, W. Joosen. Managing Concern Interactions in Middleware. In Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems, pp.267-283. Cyprus, May 2007.

[26] P. Grace, G. Coulson, G. Blair, B. Porter: Deep Middleware for the Divergent Grid. IFIP/ACM Middleware 2005, pp. 334-353, Grenoble, France, Nov 2005.