Reflection and Aspects meet again: Runtime Reflective Mechanisms for Dynamic Aspects

Bencomo N., Blair G., Coulson G., Grace P., and Rashid A.
Lancaster University
Comp. Dep., InfoLab21,
Lancaster, UK, LA1 4WA
{nelly, gordon, geoff, gracep, marash}@comp.lancs.ac.uk

ABSTRACT
Distributed applications and middleware systems typically involve language and system-wide heterogeneity e.g. different platforms (PC, PDA, embedded devices, etc.). Dynamic adaptation of distributed systems at run-time is a common approach to deal with the resultant environmental conditions. Dynamic aspects have been identified as a technique to address this problem. In such kind of applications, advices cannot be considered as a simple 'piece of code' as it might be in single-language AOP approaches; instead advices should be realised in different ways in distinct parts of the system depending upon the platform, language, and dynamics of program execution. This position paper discusses the use of a suit of orthogonal meta-level models as the basis to provide different reflective implementation mechanisms for supporting AOP approaches in a language and platform independent fashion.

Categories and Subject Descriptors
D.2.10 [**Software Engineering**]: Software Architecture; D.2.11 [**Software Engineering**]: Design; D.4.7 [**Operating Systems**]: Organization and Design – *Distributed Systems, Hierarchical design.*

General Terms: Design

Keywords: AOP, Reflection, Reflective Architectures, Middleware

1. INTRODUCTION
Dynamic adaptation of distributed systems at run-time is a common approach to deal with changing environmental conditions; for example as encountered by mobile applications as they move from one location to another. Dynamic aspects have been identified as one technique to address this problem, focusing in particular on promoting separations of concerns (SOC); where dynamic aspects involve plug and unplug of aspects without stopping, and restarting a running system. Currently, majority of implementations of Dynamic AOP are language dependent; for example, Lasagne/J [20] is implemented in Java, and PROSE [10] relies on manipulation of Java byte code. However, distributed applications and middleware systems typically involve system wide heterogeneity e.g. different platform (PC, PDA, embedded devices, etc.) and language heterogeneity. Therefore, it is often not feasible to realise an aspect as a 'simple piece' of code to be always inserted in the same fashion; rather an advice must be realised very differently in different parts of the system depending upon platform, language and dynamics of program execution.

In this paper, we present an approach for language and platform independent dynamic AOP based upon reflection. The approach is based on the OpenCOM programming model. *OpenCOM* [6] is a platform and language independent component model, developed at Lancaster, which offers three meta-level models: interface, interception, and architecture. We believe that these language independent meta-level models (meta-models for short) are naturally suited to the implementation of dynamic aspects. This position paper discusses the use of this suit of orthogonal meta-models as the basis for appropriate and different low level implementation reflective mechanisms for supporting AOP approaches in an open manner. In particular we focus on the use of interception and architecture meta-models.

The paper is organized as follows. Section 2 gives a short background about AOP, Reflection and the synergy between them. Section 3 briefly presents an overview of the reflective middleware at Lancaster. Section 4 discusses how the interception and architecture meta-models are used to underpin dynamic AOP capabilities. Some discussion about the challenges and issues is presented. Finally, section 5 gives some final remarks and outlines an agenda for future research.

2. SEPARATION OF CONCERNS WITH REFLECTION AND AOP
Separation of concerns (SOC) can be applied at different stages of the life cycle and different levels of abstraction. A number of approaches have been proposed to achieve separation of concerns [16]. Computational reflection [9][18] and aspect-oriented programming (AOP) [7] are examples of such approaches. The main purpose of reflection is to maintain an architectural separation of concerns between the base level and the meta-level. So called meta-object protocols (MOPs) define interfaces to this meta-level. Experiments with reflection had shown that structured organization of the meta-level brings benefits in terms of modularity and extensibility [2][8][17][24]. In the specific case of AOP, it is employed to separate any cross-cutting concerns. AOP introduces a unit of modularization -called aspect- that crosscuts other modules. The implementation of crosscutting

concerns is captured in aspects instead of mingling them with the rest of the modules. Figure 1 shows the dimensions of concerns separated by reflection and AOP. While reflection separates the base computation at the so-called base-level and meta-computations (computations about computation) at the meta-level, AOP separates any crosscutting concerns or aspects.

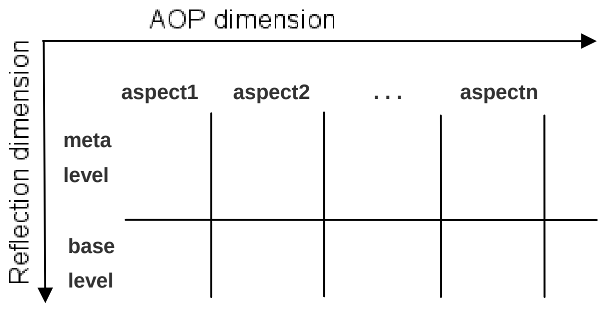


Figure 1. Dimensions of concerns separated by reflection and AOP:

In the specific case of middleware platforms, reflection has been used to provide a greater degree of configurability and dynamic adaptability at the middleware level. While the base-level consists of the implementation of the usual middleware services, the meta-level comprises reflective facilities to expose such implementation, enabling inspection and adaptation.

Middleware development includes a range of concerns that should be integrated, as transparently as possible. System wide concerns such as persistence, security, resource localization and synchronisation, are essentially of a crosscutting nature and, therefore, become entangled in the system, thus decreasing understandability and potential for reuse. All this, makes the development of middleware platforms even more complicated. Because of their capability to separate concerns, AOP has been accepted as a promising technique for middleware developments. AOP offers a useful abstraction principle to structure the base and meta-level of the middleware. This position paper is in the context of taking the advantages of both worlds, reflection and AOP, to separate concerns in the development of middleware platforms that can be adaptable and re-configurable at run-time.

3. REFLECTIVE MIDDLEWARE USING META-MODELS

The reflective middleware research [21] carried out at Lancaster University is based on three key concepts: components, components frameworks and reflection. At Lancaster both the middleware platform and the application are built from interconnected sets of components. The underlying component model is based on OpenCOM [4], a general-purpose and language-independent component-based systems building technology. Figure 2 shows the main concepts of the OpenCOM architecture: components, interfaces, receptacles, bindings and capsules (containers). OpenCOM supports the construction of dynamic systems that may require run-time reconfiguration. It is straightforwardly deployable in a wide range of deployment environments ranging from standard PCs, resource-poor PDAs, embedded systems with no OS support, and high speed network processors. Components are complemented by the coarser-grained notion of component frameworks (CFs) [19]. A CF is a set of components that cooperate to address a required functionality or structure (e.g. Discovery and Advertising, Security, Interactions, etc). CFs also accept additional 'plug-in' components that change and extend behaviour. Many interpretations of the CF notion foresee only design-time or build-time plugability. In our interpretation run-time plugability is also included, and CFs actively police attempts to plug in new components according to well-defined, per-CF, policies and constraints.

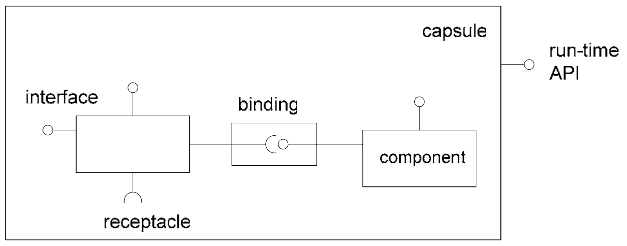
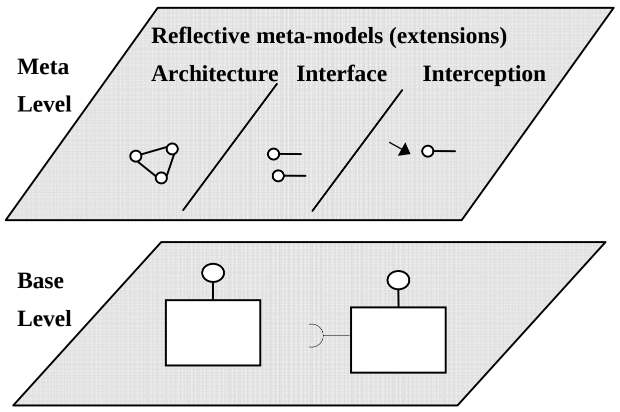


Figure 2. The OpenCOM main concepts

Reflection is used to support introspection and adaptation of the underlying component/ CF structures [2]. A pillar of our approach to reflection is to provide an extensible suite of orthogonal meta-models each of which is optional and can be dynamically loaded when required, and unloaded when no longer required. The fact that they are optional makes them to be called reflective extensions. The motivation of this approach is to provide a separation of concerns to reduce the complexity of the overall metalevel. This approach was first advocated by the designers of AL-1/D [11]. Three reflective meta-models are now supported (Architecture, Interface, and Interception); see Figure 3. These meta-models are explained below.



Architecture, Interface, and Interception meta-models provide a separation of concerns to reduce the complexity of the overall metalevel.

Figure 3. The meta-space structure:

The architecture meta-model represents the current topology of a composition of components within a capsule; it is used to inspect (discover), adapt and extend a set of components. For example, we might want to change or insert a compression component to operate efficiently over a wireless link. This meta-model provides access to the implementation of the meta-component that has a component graph where components are nodes and bindings are arcs. Inspection is achieved by traversing the graph, and adaptation/extension is realized by inserting or removing nodes or arcs.

The interface meta-model supports the dynamic discovery of the set of interfaces defined on a component; support is also provided for the dynamic invocation of methods defined on these interfaces [2]. Both capabilities enable the invocation of inter-faces whose types were unknown at design time.

The Interception meta-model supports the dynamic interception of incoming method calls on interfaces and also the association of pre- and post-method-call code [2]. The code elements that are interposed are called interceptors. For example, in the above wireless link scenario we might want to use an interceptor to monitor the conditions under which the compressor should be switched.

4. DIFFERENT RUN-TIME REFLECTIVE MECHANISMS FOR DYNAMIC AOP

Dynamic Aspect-Oriented Programming promotes the same benefits as AOP, but the aspects are woven at run-time rather than compile time [5]. This technique, although not labelled as reflective, complements reflection in nature. AOP provides a series of techniques to enable the programmer to reason at a higher level about issues that cross-cut the structure of a system, allowing such concerns to be adapted to suit the current context.

A number of dynamic AOP techniques, e.g., [12][13][14][15][21][22] have employed reflection as a basis for supporting dynamic aspects via interception or byte code rewriting. The goal in these cases, however, has been to use reflection and existing meta-object protocols as a tool to support dynamic aspects. We advocate for the use of the meta-level partitioned into several orthogonal meta-level models (or meta-models) to make it possible to realize aspects using different techniques or approaches in different modules of the system. This section discusses about how the interception and architecture meta-models are used in this context.

4.1 INTERCEPTION META-MODEL AS A REFLECTIVE MECHANISM FOR AOP

The interception meta-model described above provides a meta-model of the process of invoking an arbitrary operation in a component's interface. This is done by providing the ability to interpose interceptors in bindings. A meta-interface is provided that allows the meta-programmer to interpose arbitrary code elements called interceptors at interfaces, such that an interceptor is executed whenever one of the interface's operations is invoked [6]. The interceptor may be executed either before, after, or both before and after, the operation invocation.

Our realization of the meta-model provides both interception-capable and non-interception-capable bindings according the requirements – i.e. we choose an interception-capable binding if we are likely to need interception capabilities. Moreover, bindings can be broken and rebound in case we have made a wrong choice. Alternative implementations of interception are offered presenting different trade-offs.

Figure 4 shows an example of a binding component between the two components *C1* and *C2*. Component *C2* offers its services to *C1*. The binding offers an interface to allow dynamic introspection and to plug interceptors.

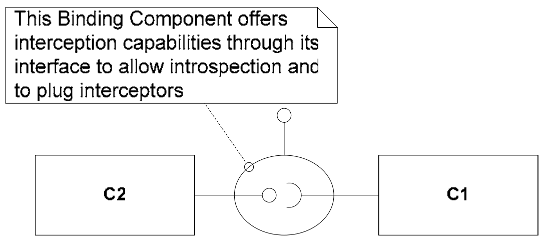

This Binding Component offers interception capabilities through its interface to allow introspection and to plug interceptors
C2
C1

Figure 4. Binding Component between components C1 and C2 offering interception capabilities through its interface

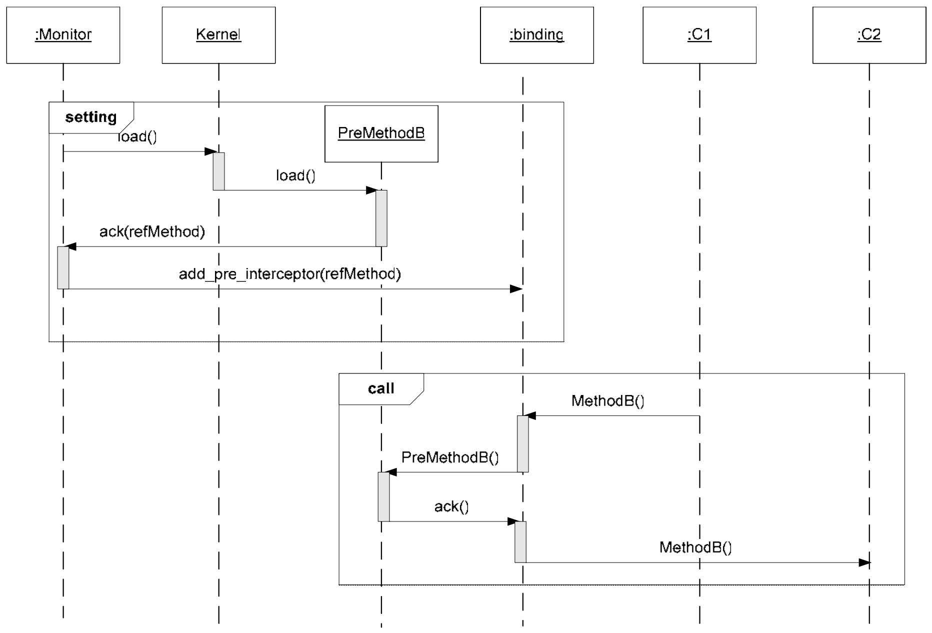
Interceptors as Advices: The (in) famous tracing aspect again

The *Tracing* aspect is a passive aspect that monitors calls to, and returns from, methods being traced within an application and displays this information in some way. This aspect is passive in that its occurrence within an application does not affect the rest of the application's behaviour; the tracing aspect uses the before and after forms of advice.

In the context of the interception meta-model, interceptors can be seen as advices. Let's study the following case: suppose component *C2* of Figure 4 has an interface with a method called *MethodB* and an interceptor is going to be (pre) added using the interception capabilities offered by the binding. The interceptor will be used as a tracing aspect that will log the entry of the *MethodB*. During the setting phase, the monitor (meta-program) requests the loading of the interceptor. The Kernel loads the interceptor *PreMethodB* and the interceptor is added to the binding. During the call phase, when *C1* calls *methodB* in C2 the interceptor is executed before the operation invocation.

Figure 5 shows the sequence diagram associated with the interception process. This process has two phases, the phase setting where the *PreMethodB* is going to be created (if it does not exist yet) and loaded into the capsule (container). The phase call consists of the call and therefore interception of the method to be traced, in this case the *methodB* in the component *C2*. A similar process can be applied in the case of a post interceptor (when the method tracing logs the exit of the method), or several interceptors are inserted.

While tracing is passive, the around form of advice is more intrusive. In the example of the tracing aspect above, the interceptor *PreMethodB* was simply executed as additional code before the advised operation. No changes were done when calling *methodB*. Given the fact that the interception meta-model provides capabilities to manipulate the parameters of the calls, additional execution before and/or after the join point can be done when modifying the arguments of the call. In this case we would be using the around style of advice. The sequence diagram for this case is very similar to the one shown in Figure 5.



During the *setting* phase, the monitor (meta-program) requests the loading of the interceptor. The Kernel loads the interceptor *PreMethodB* and the interceptor is added to the binding. During the *call* phase, when C1 calls methodB in C2 the interceptor is executed before the operation invocation.

Figure 5. Sequence diagram of the (pre) interception of *MethodB* of component C1

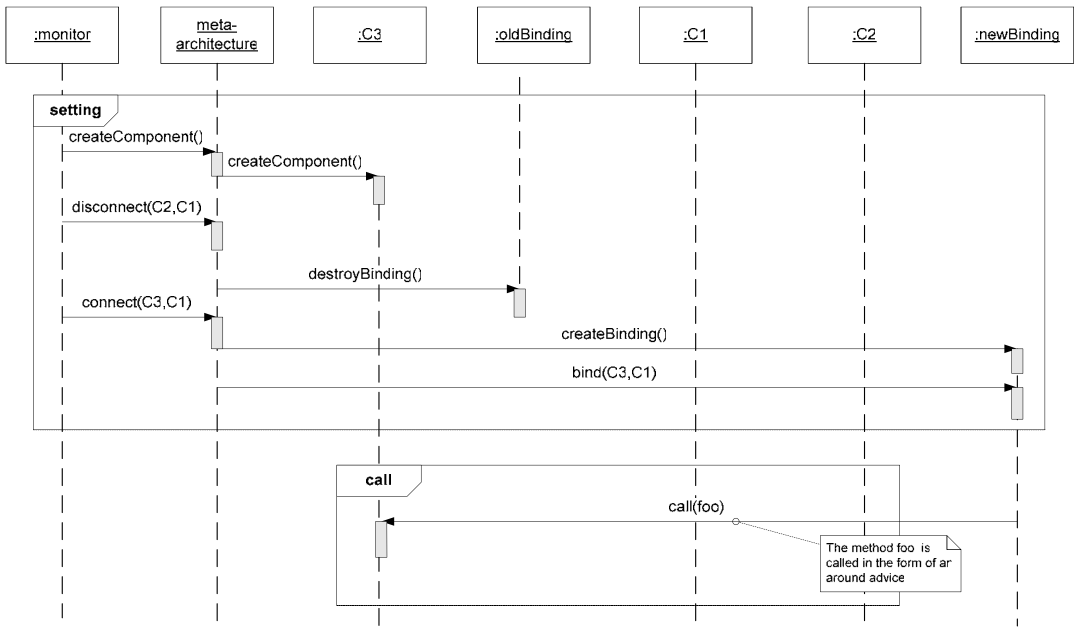
Matching the AOP and interception meta-model vocabularies we observe that the term *advice* coincides with the term *interceptor* and an *interception point* in the binding component matches a *joint point*. The interceptor may be executed either, before, or after, or both before and after, the operation invocation. The case of around advice is considered under the context of the architecture meta-model and is explained in the next section. An aspect in this context can be defined as a collection of interceptors and the mechanism to specify the join points (interception points) where the advices (interceptors) operate.

4.2 ARCHITECTURE META-MODEL AS A REFLECTIVE MECHANISM FOR AOP

As we stated before, the case of around advice is more intrusive than the cases of before/after advice. Some typical uses of this advice include when you might want to bypass the execution of the captured join point, execute it with different argument, execute it several times, and/or perform additional execution before and after the join point. It might end changing the structure of the application by the introduction of new components. In the next section we investigate the case where the original execution is bypassed and some other logic is performed in its place. The basic idea is to replace at run-time the component that offers the implementation of the behaviour to be bypassed with a new component that will offer the new required behaviour.

The around advice: an example using the architecture meta-model

The architecture meta-model provides adaptation capabilities allowing insertion (and removal) of components, i.e. insertion/removal of nodes or arcs in the graph of the meta-model. In this context, advices can be weaved by introducing new components into the graph offering a form of an around advice. Let’s study the following case. Suppose component *C2* has an interface with a method called *foo*, *C1* uses this interface. It means that a binding exists between *C1* and *C2*. A new component *C3* offering another implementation of the operation *foo* will be inserted. *C1* will be connected to *C3* instead of *C2* using a new binding. The operation *foo* in *C3* will be the new advice introduced instead of the *foo* implementation offered by *C2*. Figure 6 shows the sequence diagram associated with the insertion of the advice.



During the *setting* phase, the monitor (meta-program) requests the creation of the component *C3*, the disconnection of *C2* and *C1*, and the connection of *C3* and *C1*. The meta-architecture component will destroy the old binding between *C2* and *C1* and will create a new binding between *C3* and *C1*. When required during *call* phase, the new binding will call the operation *foo* on *C3*.

Figure 6. Sequence diagram of the insertion of an around advice using the Architecture meta-model

Matching the AOP and architecture meta-model vocabularies we find that *advices* coincide with a set of different operations in components, and a *joint point* matches the *interface binding* in the binding component *(*where operations in component are called). Aspects, in this meta-model are defined as a collection of components (with their implementations of operations) and the mechanism to specify the join points (inside the bindings) where the advices (in the form of operations) are triggered.

4.3 DISCUSSION

The examples presented in the previous sections have shown that the conjunction of reflection and dynamic AOP in our approach offer potential solutions for providing principled adaptation of middleware platforms in heterogeneous environments. We are now investigating how to combine the *interception* and *architecture* meta-model to implement other cases associated with the around advice.

The partial results show how different aspects can be treated in different ways using the meta-models. At runtime when components exchange messages, messages can be intercepted and additional processing can be realized before and after message delivery. As in the example of the tracing aspect given above, advices as interceptors can be used to monitor and check certain quality attributes at run-time.

Security policies and procedures at various points in the base level code can be modified at run-time to reflect the current operating environment, e.g. type of network [5]. Enforced dynamic security policies can be tailored using different implementations of operations of interfaces offered by distinct components. Using the architecture meta-model, advices can take the form of components and their operations (implementations).

5. SUMMARY AND FUTURE WORK

In this position paper we advocate for providing different reflective mechanisms (based on different orthogonal meta-models) as implementation of AOP. This gives us the ability to realize advice very differently in several parts of the system depending on the language, platform and dynamics of execution. This will result in a more flexible middleware platform that can be adapted to face the high levels of configurability and dynamic adaptability. So far, we have studied how the *architecture* and *interception* meta-models can be used to achieve our goal. More work has to be done in relation to the i*nterface* meta-model and how to use the meta-models together.

Reflection is often criticised of being an expensive solution, which incurs overhead in terms of system performance because of the storage and management of meta-information. Hence, we will thoroughly evaluate the efficiency of our implementation of dynamic aspects. This evaluation aims to demonstrate that the overhead of our reflective approach to dynamic aspects is minimised. For this purpose, we will compare the performance of our flexible approach to other dynamic aspects implementations for a set of specific aspects case studies. We foresee, based upon our previous experience of reflective solutions [5], that given the trade-off between flexibility and performance, it is possible to produce high performance, flexible systems.

Finally, we believe that an interesting and important area of future research is the ability to interpret pointcut expressions at run-time to dynamically change advice introductions. That is, we aim to develop a language-independent pointcut language to reason and make changes to aspects in systems in the same way we reason about components. This will allow us to specify pointcuts which pick the desired join points at runtime.

6. ACKNOWLEDGMENTS

This research is part-financed by the AOSD-Europe and RUNES projects. These projects are supported by research funding from the European Network of Excellence on Aspect-Oriented Software Development under contract number. T IST-2-004349 and the European Commission's 6th framework Programme under contract number IST-004536 respectively.

7. REFERENCES

[1] Blair G., Blair. L., Rashid A. Moreira A., Araújo J., Chitchyan R. *Aspect-Oriented Software Development, chapter 17 - Engineering Aspect-Oriented Systems*, pages 379-406. Addison-Wesley, 2005

[2] Blair, G., Coulson, G., Grace, P.: *Research Directions in Reflective Middleware: the Lancaster Experience*, Proc. 3rd Workshop on Reflective and Adaptive Middleware (RM2004), Canada, (2004), 262-267.

[3] Costa, F. *Combining meta-information management and refection in an architecture for configurable and reconfigurable middleware*. Ph.D. Dissertation, University of Lancaster, 2001

[4] Coulson, G., Blair, G.S., Grace, P., Joolia, A., Lee, K., Ueyama, J. *A Component Model for Building Systems Software*, Proceeding of IASTED Software Engineering and Applications (SEA'04), USA, 2004

[5] Coulson G., Blair G., and Grace P., *On the Performance of Reflective Systems Software*, In Proc. of International Workshop on Middleware Performance, USA, 2004

[6] Grace P., Blair G., *Reflective Middleware, Chapter in book:Mobile Middleware*, ed: P. Bellavista and A. Corradi, CRC Press (To be published)

[7] Kiczales G. Aspect *Oriented Programming*, ACM Computing Surveys (CSUR), vol. 28, no. 4, 1996

[8] Kon F., Costa F., Blair G., Campbell R. *The case for reflective middleware*. Commun. ACM 45(6): 33-38 (2002)

[9] Maes, P.: *Concepts and Experiments in Computational Reflection*, Proc. OOPSLA'87, Vol. 22 of ACM SIGPLAN Notices, pp147-155, ACM Press, 1987

[10] Nicoara A., Alonso G.: *Dynamic AOP with PROSE*, Proc. of International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA'05) in CAISE'05, Portugal, (2005)

[11] Okamura H., Ishikawa Y. and Takoro M., *AL-1/D: A Distributed Programming System with multi-Model Reflection Framework*, Proc. Int. Workshop on reflection and Meta-level Architecures, Japan, 1992, 36-47

[12] Pawlak R., Seinturier L., Duchien L., and Florin G., *JAC: A Flexible Solution for Aspect-Oriented Programming in Java*, 3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection), 2001, Springer-Verlag, Lecture Notes in Computer Science, 2192, pp. 1-25.

[13] Popovici A., Alonso G., and Gross T., *Just-In-Time Aspects: Efficient Dynamic Weaving for Java*, 2nd International Conference on Aspect-Oriented Software Development (AOSD), 2003, ACM, pp. 100-109.

[14] Popovici A., Frei A., and Alonso G., *A Proactive Middleware Platform for Mobile Computing*, ACM/IFIP/USENIX International Middleware Conference, 2003, Springer-Verlag, Lecture Notes in Computer Science, 2672, pp. 455-473.

[15] Popovici A., Gross T, and Alonso G, *Dynamic Weaving for Aspect-Oriented Programming*, 1st International Conference on Aspect-Oriented Software Development (AOSD-2002), 2002, ACM, pp. 141-147

[16] Rashid, A. *A Hybrid Approach to Separation of Concerns: The Story of SADES*. 3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection). Springer-Verlag Lecture Notes in Computer Science. Volume 2192, 2001, 231-249

[17] Rodriguez L., Tanter E., Noye J. *Supporting Dynamic Crosscutting with Partial Behavioral Reflection: A Case Study*, sccc,. 48-58, XXIV 2004, 48-58

[18] Smith B.: *Reflection and Semantics in a Procedural Language*, PhD thesis, MIT Laboratory of Computer Science, 1982

[19] Szyperski C.: *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, (2002)

[20] E. Truyen, *Dynamic and Context-Sensitive Composition in Distributed Systems*, PhD thesis, K.U.Leuven, Belgium, (2004)

[21] *AspectS Home Page*, http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/, 2005.

[22] *AspectWerkz Team*, "AspectWerkz Project", http://aspectwerkz.codehaus.org/, 2005

[23] *Middleware at Lancaster* http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/index.php

[24] *The JBoss Projec*t: http://www.jboss.org/index.html.