Aspects for Run-Time Component Integration

Eddy Truyen[†], Bo Nørregaard Jørgensen[♣], Wouter Joosen, Pierre Verbaeten

[†]Distrinet Labs, KULeuven
Dept. ComputerWetenschappen
Celestijnenlaan200A, 3001 Leuven
{eddy,wouter}@cs.kuleuven.ac.be

[♣]The Maersk Mc-Kinney Moller Institute for Production Technology,
University of Southern Denmark, Odense Campus,
DK-5230 Odense M, Denmark.
bnj@mip.sdu.dk

abstract. Component framework technology has become the cornerstone of building a family of systems and applications. A component framework defines a generic architecture into which specialized components can be plugged. As such, the component framework leverages the glue that connects the different inserted components together. We have examined a middle ground between aspect-oriented programming and computational reflection that improves the dynamics of this gluing process such that interaction between components can be refined at run-time. In this paper, we show how we have used this middle ground to dynamically integrate into the architecture of middleware systems new services that support non-functional aspects such as security, transactions, real-time.

1 Introduction

The success of distributed object technology, depends on the advent of Object Request Brokers (ORBs) and middleware architectures that are able to integrate flexible support for various non-functional aspects. Non-functional aspects pertain to requirements that are not directly included in the functionality of a distributed application (i.e. what the application does) but rather express additional properties according to which the application should behave. For instance, in industrial manufacturing settings such additional requirements include fault tolerant and real-time responsiveness with different application-specific fault tolerance policies and deadlines. The development of ORBs that support vertical integration of such non-functional aspects from the application level all the way down to the network layer is crucial for successful application of distributed objects.

To deal with the wide range of non-functional aspects, ORBs are required that can be customized to application-specific preferences. Application-specific customization of an ORB requires some level of flexibility and openness in the ORB implementation. For example in E-commerce, deployed middleware should provide client applications with additional API's for transaction support and an application-specific level of security. Furthermore client-side ORB components must be able to adapt to or integrate specific protocols (e.g. authentication protocol) expected by E-commerce servers.

These examples show that ORBs must be highly reconfigurable such that they are easy to extend with new functionality and allow upgrading the quality of their service.

1.1 Component Frameworks

Current ORB technologies such as Java RMI, DCOM and CORBA [1,2,3] are monolithic coded systems that are a priori difficult to reconfigure. Some ORB implementations offer customizability support that is however limited to customizing specific features.

[†] This research was supported by a grant from the Flemish Institute for the advancement of scientific-technological research in the industry (IWT) and …

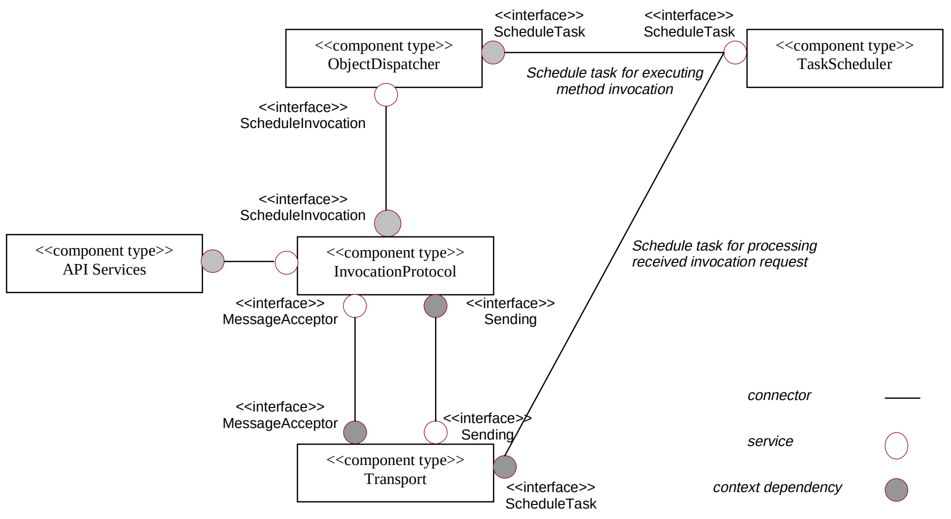
[♣] the A. P. Møller and Chastine Mc-Kinney Møller Foundation.

Generic platforms are needed in stead that support reconfiguration of an ORB system vertically all the way down from the application-level to the network layer. In recent research component framework technology [4] has more and more been put forward as the key to realize such platforms for protocol stacks, object request brokers as well as distributed applications [5,6,7,8,9,10,11,12].

The low coupling between the contractually specified interfaces of a component on the one hand and the implementation of the component on the other hand carries the promise of dynamic reconfiguration. The interfaces of a component make up the type of the component, or short *component type*. We distinguish between the interfaces on which a component depends (its *context dependencies*) and the interfaces that a component exports (its *services*).

An ORB implementation is built as a composition of components. A composition is defined as a set of connectors, that each connects a context dependency interface of one component with a type-equivalent service interface of another component.

By separating the composition between components on the one hand from the component implementations that make part of this composition, the composition is encapsulated in an ORB component framework as a reusable blueprint.



<<component type>> ObjectDispatcher
<<interface>> ScheduleTask
<<interface>> ScheduleTask
<<component type>> TaskScheduler
Schedule task for executing method invocation
<<interface>> ScheduleInvocation
<<interface>> ScheduleInvocation
<<component type>> API Services
<<component type>> InvocationProtocol
Schedule task for processing received invocation request
<<interface>> MessageAcceptor
<<interface>> Sending
<<interface>> MessageAcceptor
<<interface>> Sending
<<component type>> Transport
<<interface>> ScheduleTask
connector
service
context dependency

Figure 1 Example of an ORB component framework

Figure 1 illustrates a part of such an ORB component framework. For example, a component of type `Transport` is expected to offer an interface for sending invocation requests to remote endpoints. A `Transport` component is furthermore dependent on a `TaskScheduler` component that coordinates how the `Transport` component is processing incoming invocation requests. It is also dependent on an `InvocationProtocol` component for delivering incoming invocation request and replies.

On the boundaries of the component framework, component types define plug-ins for specific components. An ORB implementation is then constructed as a specialization of the component framework by selecting appropriate component implementations for each component type. There can be more than one possible component implementation per component type; the alternatives implement different protocols and algorithms with different quality of service (QoS) support. Upgrading an ORB implementation for a specific distributed application is then a matter of selecting those component implementations that perform better for that specific application.

1.2 Aspect-Oriented Programmin g

AOP [13] is a well-known open implementation technique, that strives to offer an easy programming model to application programmers that in general do not have the skills to comprehend the complexities of using a Meta-Object Protocol (MOP). AOP enables that various aspects related to distribution and synchronization can be integrated with third party applications at compile-time. This makes AOP a static approach. Based on earlier experiences [20] and the lessons learned from building dynamical re-configurable ORBs as described in section 1.1, we argue that an explicit representation of some aspects at run-time is required to capture the dynamic preferences of an application with regard to which component implementation is to be plugged into the framework.

Composition Filters [21] is an aspect-oriented technique that implements run-time weaving of aspects into applications. In this approach, the aspect-oriented program is however kept centralized within the system itself, leaving no opportunity for application-specific customization

1.3 Overview

In section 2 we present a case in middleware deployment that shows the need for integrating components at run-time. We further present the logical process of how to achieve run-time integration of type-incompatible components. This integration process is based on the use of wrappers. In section 3 we present a reflective architecture for building ORB component frameworks that support such an on-line integration process. In section 3.1 we examine how the use of aspects and a clear separation of the different dimensions that appear in the component integration process help controlling crosscuts between non-orthogonal non-functional policies.

2 Component Integration At Run-time

A running ORB system consists of multiple processes that are created in the deployment space of the ORB, which typically spans a topology of host connected in a network[9]. There is need for a reconfiguration tool for upgrading running ORB systems on-line, guaranteeing 7x24 availability of applications such as banking and telecommunications. This reconfiguration tool needs to be able to reflect on the existing functionality of a running ORB system and reconfigure its functionality at appropriate intercession points.

The set of possible reconfigurations may be not known at compile-time. This means that a running system may need to cope with unanticipated changes. For example, the question arises what to do when in the process of linking a running application with an external system, the underlying middleware is required to integrate new protocols that the design of the ORB component framework is not anticipated for. Typically this involves a new component with incompatible interfaces that must be linked into the deployment space of the running ORB system. To achieve this integration anyway, the interaction behavior of at least one of the existing components in the ORB system must be adapted or extended at run-time in order to enable cooperation with the newly introduced components.

Wrappers are known mechanisms for introducing new interaction behavior to existing components [14]. For example, suppose that an encryption/decryption component must be introduced into the running ORB system of Figure 1, with the intention that some client invocation requests must be sent over the network in a confidential way.

To achieve this reconfiguration, the system integrator may decide to extend the interaction behavior of the component implementations of type `Transport` running in the system. At the sending ORB-side, each remote call has to be encrypted, before being sent by a `Transport` component, and at the receiver ORB-side the remote call must be decrypted after its receipt by a peer `Transport` component. Figure 2 shows the basic scheme of wrapping when handling an outgoing invocation request that must be send to a remote peer confidentially.

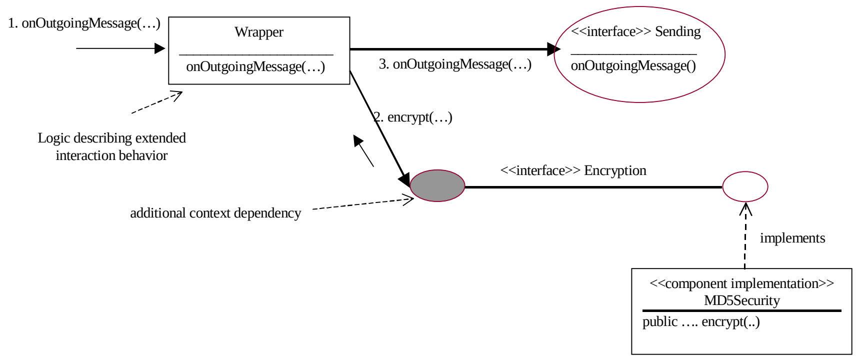


Figure 2 Wrappers

The wrapper codifies which additional context dependency interfaces (i.e. `Encryption`) the type of the `Transport` component type must be widened with. Furthermore the wrapper contains also codified logic that implements how the interaction behavior of the existing `Transport` component implementation(s) must be extended to incorporate the services of the newly introduced component.

The on-line reconfiguration tool enables then a system administrator to “inject” and connect this wrapper on-line into the running ORB system at the chosen intercession point.

3 Architectural Reflection as the Basis for Interaction Refinement

Run-time reconfiguration of unanticipated changes using wrappers raises two issues that must be dealt with. First, applying wrappers must be performed with caution, since type conflicts can arise, caused by semantic incompatibility between existing component types and the newly introduced component type. In [15], one formulated two programming constraints that lead to type safe unanticipated component adaptation. These criteria are based on the existence of a common parent type shared between the wrapped component and the wrapper.

More importantly, the second issue relates to the problem that current component architectures do not meet the requirements for unanticipated run-time reconfiguration. To achieve run-time reconfiguration using wrappers all the input connections of the component to be wrapped must be re-wired to the wrapper. Such a re-wiring is not supported by current component architectures like JavaBeans, COM, etc.[15].

We deal with the second issue by localizing structural information of a component (i.e. its **component type** and its **connectors** to other components), making this knowledge observable, controllable, and changeable. Systems that are able to observe and manipulate themselves are known as *reflective systems*. Since we reflect upon architectural issues, we refer to *architectural reflection*.

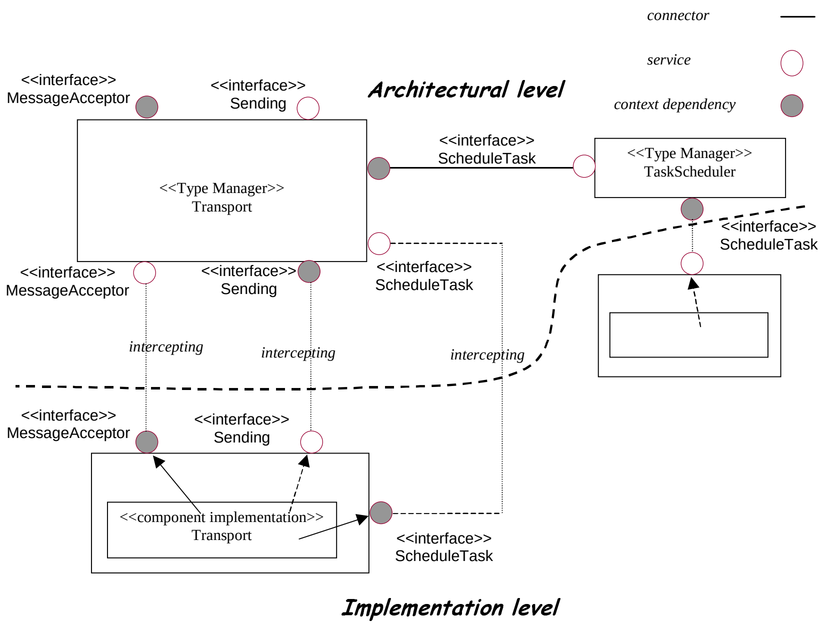

connector
service
context dependency
Architectural level
<<interface>> MessageAcceptor
<<interface>> Sending
<<Type Manager>> Transport
<<interface>> ScheduleTask
<<Type Manager>> TaskScheduler
<<interface>> ScheduleTask
<<interface>> MessageAcceptor
<<interface>> Sending
<<interface>> ScheduleTask
intercepting
intercepting
intercepting
<<interface>> MessageAcceptor
<<interface>> Sending
<<component implementation>> Transport
<<interface>> ScheduleTask
Implementation level

Figure 3 Architectural Reflection

Architectural reflection is implemented by splitting a component-based system into two separate levels. First, there is an *architectural level* that consists of *component type managers* – or short type managers – who manage the structural information of components that belongs to a specific component type. Second there is an *implementation level* that consists of the various *component implementations* that implement the functionality of the ORB system. Each type manager is code-generated in the same programming language that was used for the component implementation. In order to control the architectural knowledge of its component implementations, the type manager intercepts outgoing and incoming messages of its component implementation(s). Such an interception mechanism is commonly applied in building computational meta-object protocols (MOPs) [16]. A crucial difference however is that objects at the meta-level are required to define a general (and often fixed) interface. In this way, explicit type information of the component implementations is lost at the meta-level. The reason for this is that a computational MOP reifies messages exchanged between component implementations into first-class objects, making type information implicitly embedded in the inner parts of the reified objects. In our approach, we take a step back and limit ourselves to an interception mechanism, but we don’t perform reification of the messages exchanged between component implementations, keeping the component type information explicit at the architectural level, as shown in Figure 3. We have developed a prototype of this for Java Beans[17]. Each type manager is also implemented as a separate Java Bean class. We used a simple interception mechanism that consists of registering the type manager as an event listener with its component implementation(s).

A reusable ORB component framework, as described in section 1.1, is then built at the architectural level by connecting type managers, since they explicitly represent component types. Each type manager controls which component implementation(s) are plugged into the framework and is able to extend the interaction behavior of its component implementation(s) by for example redirecting intercepted messages to dynamically inserted wrappers.

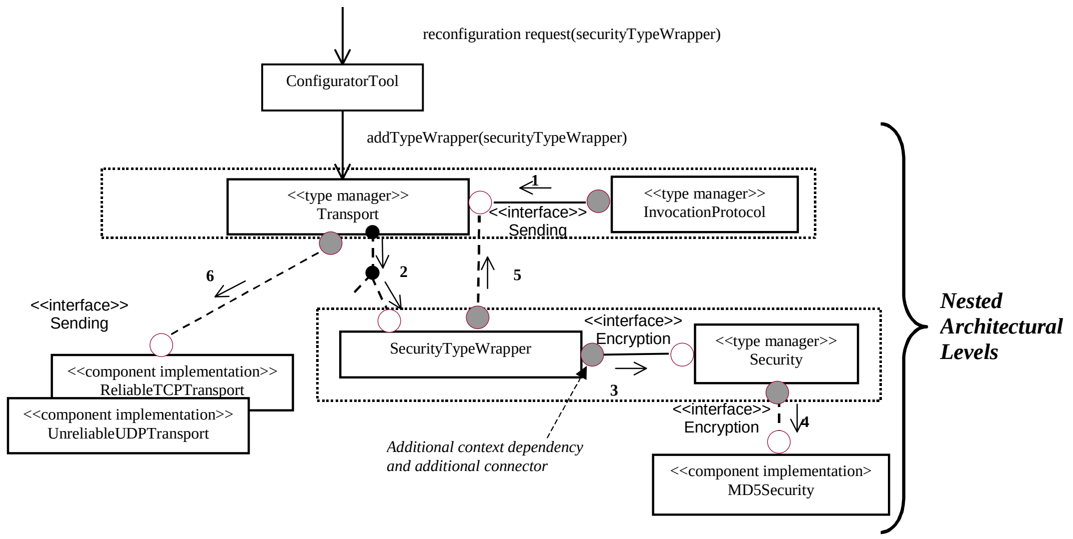
3.1 Injecting Wrappers at Run Time

In this section, we describe how the process of performing an unanticipated reconfiguration by injecting the wrapper for the example described in section 2 is to be performed by a reconfiguration manager (see Figure 4). We are working on an implementation of it in Java.

First, type information of the newly introduced component `MD5Security` is extracted by inspecting its service and context dependency interfaces. Based on this information, a new type manager `Security` for the newly introduced component type can be code-generated and is registered with the reconfiguration manager.

To exclude type conflicts after the wrapper would have been injected, the configuration manager first may check whether the two constraints formulated in [15] are not violated. If no type conflict is found, the injection of the wrapper into the running ORB system can effectively be performed. This is shown in Figure 4.

Each type manager has a method for accepting a new wrapper to be inserted into the ORB system The intercession point for the wrapper is the type manager of the component implementation(s) that are to be wrapped (i.e. type manager `Transport`). Besides passing the wrapper to this intercession point acting type manager, the reconfiguration manager is also responsible for connecting the additional context dependency interface `Encryption`, defined by the wrapper, to the newly created type manager `Security`. As such, the wrapper becomes a placeholder for a new architectural level that is nested within the existing architectural level. The wrappers itself is placed in a tree structure that is managed by the type manager. This allows that interactions can be recursively refined by injecting a second wrapper that adapts the interaction behavior of the first wrapper and is appended in the tree as a child node of the latter. As such *recursive nesting of architectural levels* as proposed by [18] is possible. This concept presents the architecture of a system as consisting of multiple strata. These strata are not layers in the normal sense (like for example in the ISO OSI model), but may actually contain the same object as another stratum but with a wider interface reflecting the effects of an interaction refinement.



reconfiguration request(securityTypeWrapper)
ConfiguratorTool
addTypeWrapper(securityTypeWrapper)
<<type manager>> Transport
1
<<interface>> Sending
<<type manager>> InvocationProtocol
6
2
5
<<interface>> Sending
SecurityTypeWrapper
<<interface>> Encryption
<<type manager>> Security
3
<<interface>> Encryption
4
<<component implementation>> ReliableTCPTransport
<<component implementation>> UnreliableUDPTransport
Additional context dependency and additional connector
<<component implementation> MD5Security
Nested Architectural Levels

Figure 4 Run-time wrapping

4 The Glue is the Clue: Global Composition Strategies

Type managers are responsible for governing how a wrapper is to be composed with the wrapped component. A type manager for example must know for which invocation requests the wrapper must be applied and also whether the wrapper must be applied before or after executing the behavior of the wrapped component implementations. Furthermore the type manager must apply wrappers in a certain order such that the required ORB semantics are not violated. This all is defined as the *composition strategy* of the type manager – "the glue is the clue".

Setting up a generic gluing mechanism that works for the general case is not practical. Instead, different gluing mechanisms that are optimized for specific purposes must co-exist. In this section we focus on gluing wrappers that are used for extending a running ORB system with new services that support *non-functional aspects* such as security, synchronisation, transactions and replication, etc. In the remainder of this section we refer to it as *aspect-wrapping*.

Run-time composition of non-functional aspects into ORB systems is a big challenge, since injection of an aspect-wrapper on behalf of a specific end-user application may semantically conflict with already performed reconfigurations or reconfiguration requests for non-functional aspects that are originating from remote sites in the deployment space of the ORB system. As such, it is very difficult to reliably control the resulting ORB semantics when multiple aspect-wrappers are independently added to a running ORB system.

The source of these difficulties is that non-functional aspects are often not orthogonal to each other, yielding to hidden dependencies. This problem has been referred to in general as the *Aspect-Composition Issue (ACI)* [19]. For example, if adding a synchronization wrapper (integrated by the `InvocationProtocol`[1] type manager) to an running ORB with an already integrated authentication wrapper (managed by the `APIServices` type manager) makes that all authentication request are to be synchronized, it can have tragic effects on the system performance.

In order to control the ACI, type managers are required to cooperate by mutually synchronizing their composition strategies in order to avoid semantic conflicts between two overlapping aspects. This means that adding a new aspect-wrapper to a specific type manager may require that one or more type managers within the entire deployment space of the ORB system must adapt the composition strategy for their aspect-wrappers in order to avoid possible conflicts with the newly introduced aspect-wrapper.

We try to tackle ACI by developing a domain-specific language that is used for expressing in a declarative manner a *global* composition strategy tailored for a specific distributed application. This global composition strategy defines a number of properties that must be enforced system-wide for the entire distributed application. Configuration management may even store various global composition strategies each tailored to a specific use-case of the distributed application. For each use case, the declarative descriptions are parsed into first-class objects, which are then attached at run-time to client invocation requests that logically makes part of this use case. As such, an application-specific global composition strategy travels together with the invocation request as part of the control flow within the ORB deployment space. This makes that type managers can dynamically adjust their local composition strategy to each other by conforming themselves to the global composition strategy.

A global composition strategy specifies which non-functional aspects must be applied for a specific invocation request and consists of a set of constraints that expose the hidden dependencies between non-orthogonal aspects. For instance in the previous example a constraint would have been formulated that states "if the invocation request is an authentication request, don't apply the synchronization aspect".

Currently, our Java Beans prototype implementation only supports declarative specification of which non-functional aspects must be supported. For each non-functional aspect a separate template is defined that offers the vocabulary for expressing quality of service (QoS) preferences concerning that non-functional aspect[8]. These templates are defined as an XML DTD.

5 Conclusion

The ultimate goal of this work is an integrated development and deployment environment for building component-based middleware based on the reflective architecture presented in section 3. This development environment consist of an extended Bean Box for composing component types into ORB architectures and selecting one or more component implementations for each component type. The deployment environment offers a reconfiguration tool for integrating new services and protocols on-line into a running ORB system, and XML

[1] See Figure 1

editing tool for defining application-specific global composition strategies that govern the gluing of aspect-wrappers in a semantically correct way.

6 References

1. Object Management Group, “The Common Object Request Broker: Architecture and Specification”, 2.2 ed., Feb. 1998.
2. A. Wollrath, R. Riggs, and J. Waldo, “A Distributed Object Model for the Java System”, USENIX Computing Systems, vol. 9, November/December 1996.
3. D. Box, “Essential COM”, Addison-Wesley, Reading, MA, 1997.
4. C. Szyperski, “Component Software: Beyond Object-Oriented Programming”, Addison-Wesley, ISBN 0-201-17888-5.
5. D. C. Schmidt and C. Cleeland, “Applying Patterns to Develop Extensible ORB Middleware”, Design Patterns in Communications, (Linda Rising, ed.), Cambridge University Press, 2000.
6. G.S. Blair, G. Coulson, P. Robin, M. Papathomas, "An Architecture for Next Generation Middleware", Proc. IFIP International Conference on Distributed Systems.
7. R. Hayton, A. Herbert, D. Donaldson, "FlexiNet Open ORB Framework", APM Technical Report 2047.01.00, APM Ltd., Poseidon House, Castle Park, Cambridge, UK, October 1997.
8. B. N. Jørgensen, E. Truyen, F. Matthijs, W. Joosen, “Customization of Object Request Brokers by Application Specific Policies”, To appear in Proceedings of the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing (Middleware2000).
9. F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, R. H. Campbell, “Monitoring, Security, and Dynamic Configuration with the *dynamicTAO* Reflective ORB”, To appear in Proceedings of the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing (Middleware2000).
10. F. Matthijs, “Component Framework Technology for Protocol Stacks”, Phd. Thesis, K.U.Leuven, ISBN 90-5682-224-1
11. Junichi Suzuki and Yoshikazu Yamamoto, "OpenWebServer: an Adaptive Web Server Using Software Patterns". In IEEE Communications, Vol. 37, No. 4, pages 46-52, April 1999.
12. Jakob Hummes and Bernard Merialdo, “Design of extensible component-based groupware”, in Computer Supported Cooperative Work - An International Journal, 1998.
13. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, J. Irwan, “Aspect-Oriented Programming”, Proceedings of ECOOP’97, Springer-Verlag LNCS 1241, June 1997.
14. J. Brant, B. Foote, R. e. Johnson, D. Roberts, “Wrappers to the Rescue”, Proceedings of ECOOP’98. Springer LNCS 1445.
15. G. Kniesel, “Type-Safe Delegation for Run-Time Component Adaptation”, In R. Guerraoui (Ed.): Proceedings of ECOOP99. Springer LNCS 1628.
16. P. Maes, “Concepts and Experiments in Computational Reflection, In Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 147-155, 1987.
17. Eddy Truyen, Bo N. Jørgensen, Wouter Joosen, "Customization of Object Request Brokers through Dynamic Reconfiguration", accepted for Tools Europe 2000.
18. C. Atkinson, T. Kühne, C. Bunse, “Dimensions of Component Based Development”, in Proceedings of the 4th International Worshop on Component-Oriented Programming.
19. R. Pawlak, L. Duchien, and G. Florin, “An automatic aspect weaver with a reflective programming language”, in Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection (Reflection’99), Springer-Verlag LNCS 1616, pp. 147-149.
20. P. Kenens, S. Michiels, F. Matthijs, B. Robben, E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, "An AOP Case with Static and Dynamic Aspects", In Proceedings of the Aspect-Oriented Programming Workshop, ECOOP'98.
21. M. Aksit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa, “Abstracting Object-Interactions Using Composition-Filters”, In object-based distributed processing, R. Guerraoui, O. Nierstrasz and M. Riveill (eds), LNCS, Springer-Verlag, pp. 152-184, 1993.