Customization of Component-based Object Request Brokers Through Dynamic Reconfiguration

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

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

[♠]*Computer Science Department, Katholieke Universiteit Leuven*
Celestijnenlaan 200A, B-3001 Leuven Belgium
{eddy,wouter}@cs.kuleuven.ac.be

Abstract.
The success of distributed object technology, depends on the advent of Object Request Broker (ORB) architectures that are able to integrate flexible support for various non-functional requirements such as security, real-time, transactions, etc. We promote component framework technology as the cornerstone for realizing such generic platforms. An ORB component framework leverages the "glue" that connects customized ORB components together. In this paper we present a reflective component architecture that improves the dynamics of this glue, such that system-wide integration of new non-functional requirements into a running ORB system becomes possible.

1. Introduction

The success of distributed object technology in critical application domains, such as robotics control systems and E-commerce, depends on the advent of Object Request Brokers (ORBs) that can adapt their implementations to application-specific preferences for various non-functional requirements. Non-functional requirements pertain to requirements that are not directly included in the functionality of the application (i.e. what the application does) but rather express additional characteristics that the application should have. Among others the list of non-functional requirements include real-time, reliability, availability and security.

Traditionally, object oriented analysis and design only focus on the entities within the problem domain, their relationships, and how they interact with external actors. This is all part of describing the functional requirements of a distributed application. However, non-functional requirements should also be dealt with during analysis and design and should not be postponed until the implementation phase. During use-case analysis considerations about non-functional requirements often come up. For instance, when describing a use-case for an E-commerce system the system developer may very well ask himself whether or not the transaction responsible for payment should be secure. By extending the use-case analysis phase to include

[†] 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.

the specification of non-functional requirements, the application developer can record non-functional requirements together with the functionality they apply to.

In robotics control applications real-time responsiveness and reliability are crucial for the correctness of the system, whereas availability and security are of more concern in E-commerce applications. Common to both application domains is the observation that use-cases within these applications have different expectations to the implementation of non-functional requirements that are realized within an ORB system. In E-commerce, use-cases involved in secure transactions require encryption of remote method invocations before they are sent over the network. However, encryption is not required for remote method invocations that are used to obtain non-confidential information such as public available product catalogues. In robotics control systems the situation is similar, here use-cases involved in on-line control of the robots are subject to timing constraints, hence remote method invocations related to the implementation of those use-cases must be executed in a timely correct fashion. However, the remote method invocations involved in setting up the system are not subject to such constraints.

This observation implies that next-generation ORBs must be able to change their behavior accordingly to application-specific preferences for non-functional requirements on a per remote method invocation basis. By now, it is well known that conventional ORB technologies like CORBA [1], Java RMI [2], and DCOM [3] do not cope well with supporting non-functional requirements. The reason is that the abstractions in their respective implementations are too limited to integrate proper support for various non-functional requirements. One of the most severe problems is the high coupling between the internal modules that make up the implementation of conventional ORBs. This means that the structural relationships between modules are encoded in the low-level implementation details of these modules, making ORBs monolithic pieces of system software, where dynamic reconfiguration is impossible.

Existing literature have proposed different approaches for introducing flexibility into ORB implementations [4][5][6]: design patterns [7], meta-level architectures [8], component-oriented programming [9]. However each approach has its drawbacks with respect to dynamic reconfiguration: design patterns lack the ability to cope with changes that must be applied at run-time, meta-level architectures make the reconfiguration process more complex than the average application developer can comprehend, and component-oriented programming often results in ad-hoc solutions.

This paper presents an approach that provides uniform support for non-functional requirements on a per invocation request basis through dynamic reconfiguration of the ORB implementation at run-time. Here, run-time reconfiguration refers to mechanisms that allow the implementation of the ORB to be dynamically adapted or extended during execution. The approach utilizes a simple reflective architecture that supports a dynamic and adaptable process of gluing components together.

The paper is organized as follows. In section 2, we motivate and show how component framework technology forms the cornerstone for building flexible middleware solutions. Section 3 gives a short overview of our approach that allows application-specific customization of Object Request Brokers. In section 4, we present a reflective component architecture that provides support for such an application-specific customization process. In section 5, we illustrate the power of this reflective architecture by showing how it is capable of dealing with unanticipated reconfiguration requests. Related work is discussed in section 6. Finally, section 7 gives concluding remarks.

2. Component frameworks

In conventional ORB designs, ORBs are viewed as black boxes. This information hiding principle helps during ORB development, but it locks in decisions that affect the ORB's ability

to meet and support non-functional requirements. To deal with this we build ORB implementations as a composition of components. Encapsulating all features of an ORB implementation within separate components allows us to reconfigure the ORB implementation accordingly to application-specific preferences for non-functional requirements. Generally identified ORB features include marshalling, threading, etc. A detailed list of ORB features can be found in [10].

By separating the architectural structure of the composition of components on the one hand from the component instances that make part of this composition, the composition process can be encapsulated in an *ORB component framework* [11][12] as a generic architecture for a family of ORB implementations. This separation is enabled by defining the basic architecture of an ORB in terms of architectural entities that abstract away from concrete implementation details. These architectural entities explicitly uncouple the type of an ORB component from its implementation. As such, we differentiate between the notions of *component type* and *component instances*. A component type defines the role of the ORB component and its relations to other ORB components. A component type is specified as a set of contractually specified interfaces. We distinguish between the interfaces on which a component depend (its *context dependencies*) and the interfaces that a component exports (its *services*).

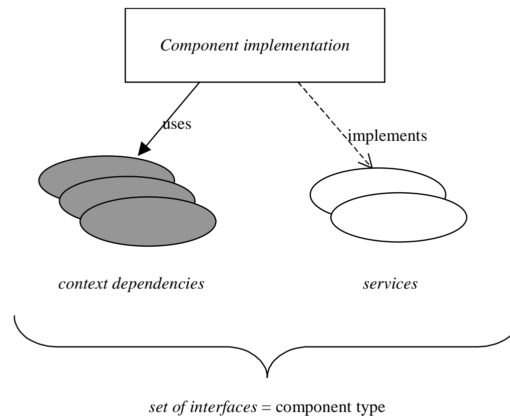

Figure 1 Component Type

A component instance realizes the implementation of an ORB component. There can be more component instances associated with each component type. Various component instances may vary in the non-functional requirements that they support and how this support is implemented. Each component instance has a *component descriptor* that describes the quality of service that its implementation provides for the non-functional requirements it supports.

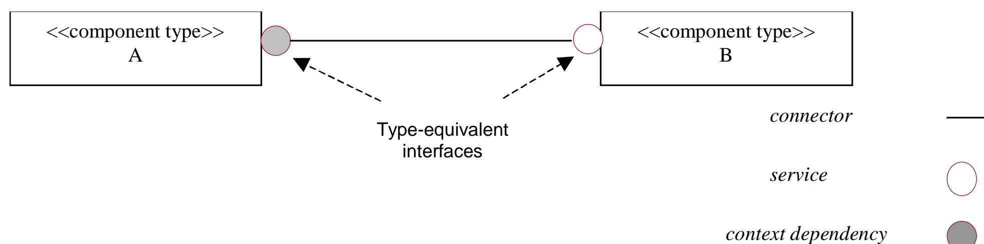

Figure 2 Connector connecting type-equivalent interfaces

Hence, a reusable architecture of an ORB is defined as a composition of component types. A composition is defined as a set of connectors, that each connects a context dependency interface of one component type with a type-equivalent service interface of another component, as shown

in Figure 2.

Figure 3 shows a component framework that consists of a JavaBeans-based ORB architecture developed for supporting distributed real-time (D-RT) applications [10]. Java Beans are normally connected to each other via events. A connector between two ORB components A and B is then setup by registering the service interface of B as an event listener with A. Java Beans can also cooperate by means of normal method invocations. A connector is then setup by setting a reference to the service interface of B as a type-compliant property of A.

The component framework also defines the responsibility of each component type:

- **Reference** implements a specific semantic for invocation requests: synchronous, asynchronous, synchronous-deferred, etc.
- **Marshaller** is responsible of marshalling outgoing invocation requests and replies, and unmarshalling incoming invocation requests and replies from Transport components.
- **Transport** transmits messages containing marshalled invocation requests and replies between address spaces.
- **InvocationScheduler** defines the concurrency model for servant objects. For each incoming invocation message, it creates a new task for performing the method call and sends the newly created task to the TaskScheduler for execution.
- **TaskScheduler** controls the execution of all tasks within the system. A task is a basic unit of computation. This includes computations related to the basic functionality of the ORB (e.g. listening for incoming messages on server sockets), as well as computations related to method execution on servant objects. The applied scheduling algorithm determines the CPU usage assigned to each task.

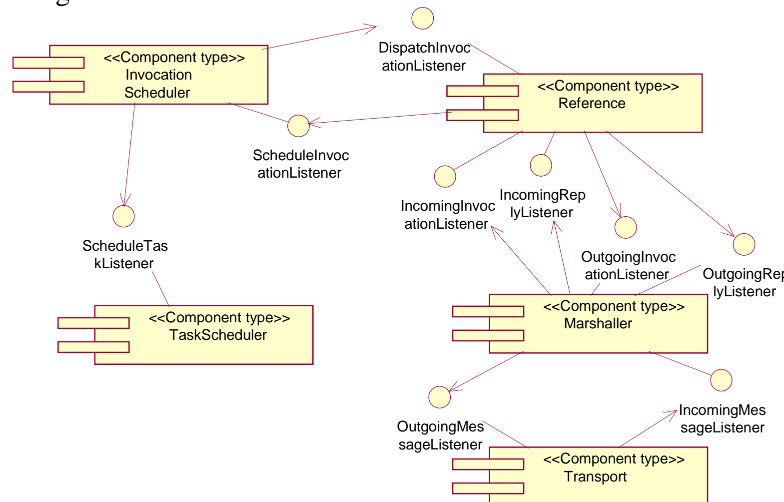

Figure 3 ORB Component Framework for D-RT applications

On the boundaries of the component framework, component types define plug-ins for specific component instances. An ORB implementation is then constructed as a specialization of the component framework by selecting an appropriate component instance for each component type. There can be more than one possible component instance per component type; the alternatives implement different protocols and algorithms with different quality of service (QoS) support. Figure 4 illustrates this construction process.

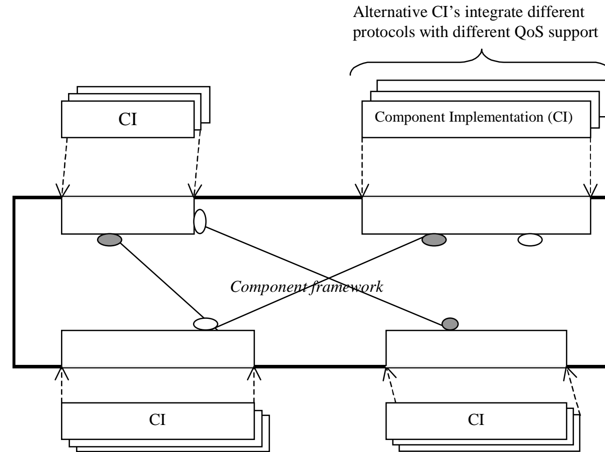

Figure 4 Conceptual structure of a Component framework

3. Application-specific customization

Building an ORB implementation that realizes support for the non-functional requirements identified as part of the extended use-case analysis mentioned in the introduction, is then a simple matter of populating the basic ORB architecture with those component instances that provide the expected behavior for each non-functional requirement.

Application-specific expectations for non-functional requirements are specified as a set of *policies*, one for each non-functional requirement. For each non-functional requirement a specific Aspect Oriented Programming (AOP[13]) language is defined, that is used for expressing policies as well as component descriptors. In our approach, AOP languages are defined as Extensible Markup Language (XML) compliant Document Type Definitions (DTD). Policies and component descriptors for a non-functional requirement are specified as XML-documents using the same DTD.

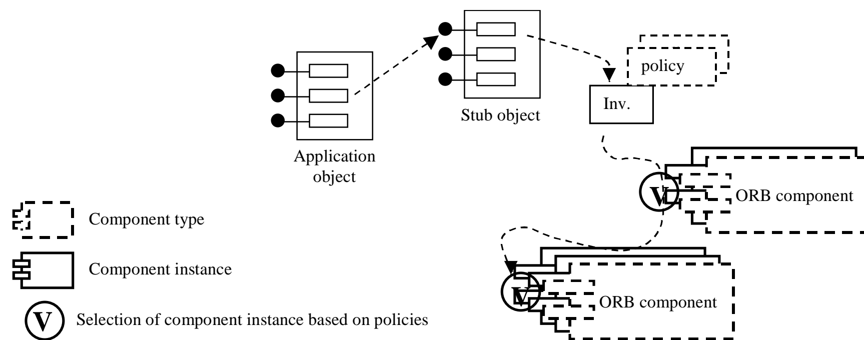

Figure 5 Conceptual view on the dynamic reconfiguration based on policies

Policies, as aspect-oriented programs, are not executed by compile-time weaving [13], but by a dynamic reconfiguration process that is based on *matching* between policies and component descriptors [10]. Figure 5 shows the conceptual view of this reconfiguration process. After a client calls a remote method, this invocation request is reified as a first-class object by a light-weight stub. The stub further attaches to the reified invocation request the application-specific policies that were specified for the remote method during use-case analysis. The invocation request then flows through the system as a sequence of events from one ORB component to the other. For one or more component types, multiple component instances however exist. In such a

case, the policies attached to the invocation request are inspected and matched with the component descriptors provided by the alternative component instances. The best matching component instance is then selected for handling the message. This scheme of selecting between alternative component instances to incorporate application-specific policies into the ORB implementation is based on the variation point concept, introduced in [14].

4. Run-time reconfiguration

When implementing the prototype of the real-time ORB shown in Figure 3 using JavaBeans, we ran into several difficulties that inhibited the realization of the dynamic reconfiguration process, illustrated in Figure 5. This is because the current component architecture of JavaBeans fails in delivering a *gluing* process that supports the dynamic rewiring we need.

In this section, we present a reflective architecture that provides ORB component frameworks with a dynamic gluing process such that interaction between component instances can be manipulated and refined at run-time.

4.1. Dynamic reconfiguration in JavaBeans

We first demonstrate the experienced shortcomings of the component architecture of Java Beans through an example. Consider an end-user application that has timeliness requirements for only a few of its use-cases. Further, suppose the ORB is initially configured with a FifoTaskSchedulerBean that implements a non-real-time scheduling algorithm. When the InvocationSchedulerBean receives an IncomingInvocationRequest event that has a deadline, the ORB must be reconfigured, such that the newly created task (for executing the method invocation), is sent to a RealtimeTaskSchedulerBean instead of the FifoTaskSchedulerBean. In architectural jargon, this re-wiring is described by changing the input connections of the "onScheduleTask" connector from the FifoTaskSchedulerBean to the RealtimeTaskScheduler-Bean.

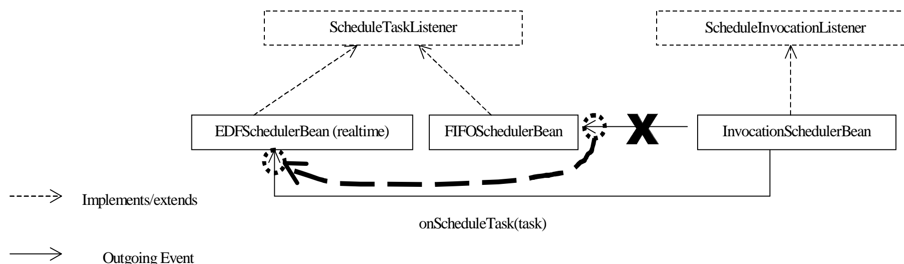

Figure 6 Re-wiring of input connections

However, when implementing this reconfiguration using JavaBeans we experienced several difficulties. In the JavaBeans model input connections correspond to the registration of a component as an event listener of other components. Hence re-wiring corresponds to unregistering the FIFOSchedulerBean and registering the EDFSchedulerBean with the InvocationSchedulerBean. However, a Java Bean is only required to maintain a list of registered listener objects but no lists of the source objects to which it listens self; thus it only knows about its outgoing connections, but not about incoming connections. So there is no way for unregistering listeners unless this information is stored elsewhere.

The second problem is that the FIFOSchedulerBean is a stateful component. There are probably still tasks running that were scheduled with the FIFOSchedulerBean. Replacement of stateful components is a non-trivial issue. First of all, the FIFOSchedulerBean might not be

prepared to hand its running tasks over to the EDFSchedulerBean, due to semantic incompatibility of the different kind of scheduling algorithms (how would you specify the deadline of a task that is scheduled by a FIFOSchedulerBean). Secondly, it is probably so that the application might require joint use of both schedulers. When the application later executes a use-case without any real-time requirements, the scheduling of tasks created on its behalf requires the FIFOSchedulerBean. So the question of when to preserve or replace a component instance must be carefully considered. Third, the re-wiring must happen as an atomic operation in order to guarantee that the system is not left in an inconsistent state. There is, however, no atomic operation for handling a bean's incoming connections over to another bean.

4.2. Architectural reflection

We deal with these issues by localizing the *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 the concept of *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 architectural knowledge of components that belongs to a specific component type in the component-based system. Second there is an *implementation level* that consists of *component instances* that implement the functionality of the system.

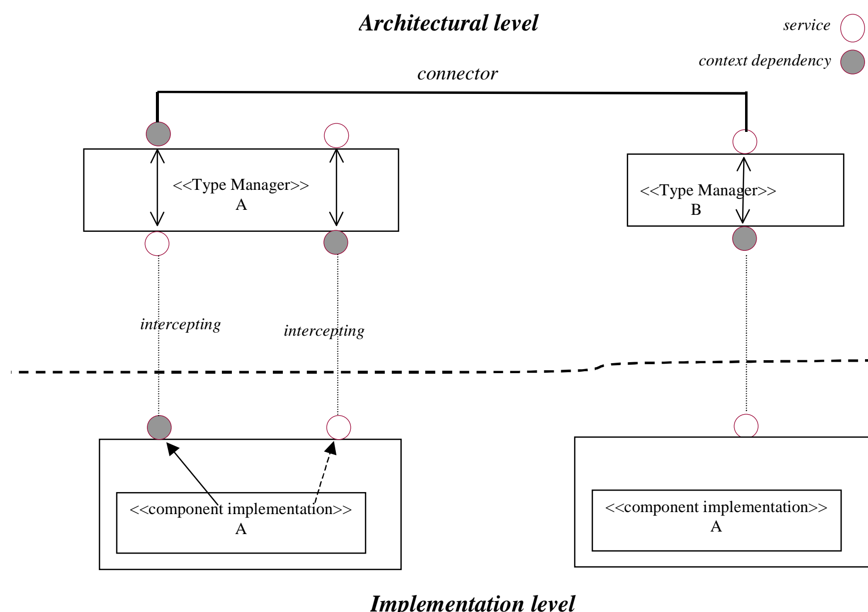

Figure 7 Architectural reflection

A type manager itself is also implemented as a Java Bean that is code-generated by a parser that inspects its component type's service and context dependency interfaces. In order to control the structural information of its component instances, the type manager intercepts outgoing and incoming messages of its component instance(s). Such an interception mechanism is commonly applied in building computational meta-object protocols (MOPs) [8]. A crucial difference however is that objects at the meta-level are required to implement a general (and often fixed) interface. In this way, explicit type information of the component instances is lost

at the meta-level. The reason for this is that a computational MOP reifies messages exchanged between component instances 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 instances, keeping the component type information explicit at the architectural level. We implemented a simple interception mechanism by registering the type manager as a listener with its component instances and by performing the setting up of event-connectors between component instances at the architectural level..

A reusable ORB component framework, as described in section 2, is then built at the architectural level by connecting type managers, since they explicitly represent component types. Each type manager controls which component instances are plugged into the framework and is able to extend and adapt the interaction behavior of its component instances by redirecting intercepted messages.

4.3. Aspect-based Forwarding

The implementation of the dynamic configuration process presented in Figure 5 is then completely realized at the architectural level. This implementation is based on forwarding of incoming events to component instances, in combination with a reification of the *variation point* concept, introduced in section 3. Each type manager incorporates a variation point that performs for each incoming event a run-time selection between component instances, to which the intercepted event is then forwarded. This run-time selection is determined by matching policies of the end-user application with component descriptors of component instances. Policies for invocation requests are associated as first-class objects with the events that flow through the system. As such, control flow in the system can be divided into a functional flow (consisting of invocation requests, replies, tasks,) and a non-functional flow describing the policies for their functional part. When a type manager receives an event on one of its incoming connections, its variation point selects that component instance of which the component descriptor best matches the policies that the type manager extracted from the event. For details about the matching we refer to [10].

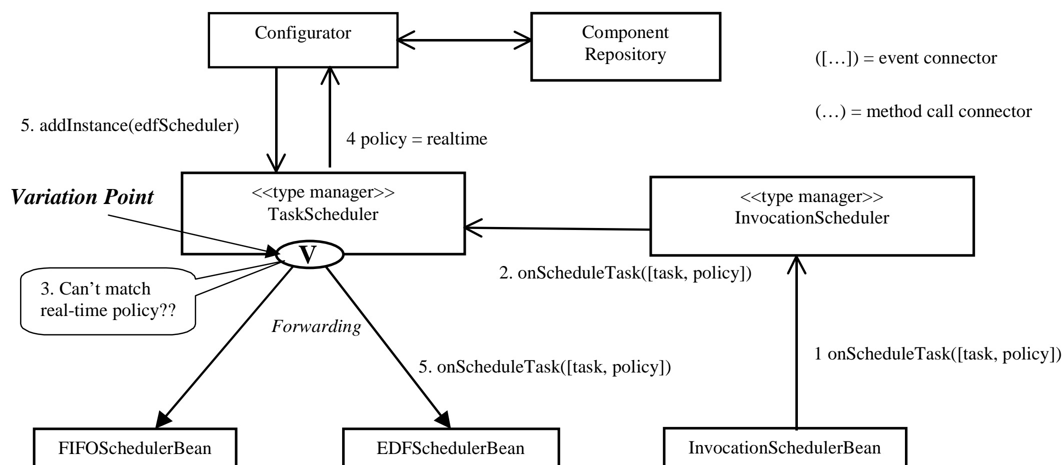

Figure 8 Aspect-based forwarding

If a match isn't found, a suitable component instance is uploaded from a component repository and added to the type manager for that component type. As discussed in the example with Figure 6, the integration of the newly introduced component instance should wisely be

performed in such a way that the old component instance is *preserved*, instead of replaced. The object-based forwarding relation between type manager and component instance makes this easy to implement by maintaining a list of component instances in the type manager.

5. Interaction refinement in middleware

Setting up a generic variation point strategy for gluing component instances that works for the general case is not practical. Instead, various variation points with different capabilities and selection strategies optimized for specific gluing purposes must co-exist. Therefore type managers offer hooks that allow different variation point strategies to be incorporated.

In this section, we illustrate a variation point that is able to dynamically integrate new component types into a running ORB that the design of the ORB's architecture is not anticipated for.

5.1. Unanticipated reconfiguration

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. Therefore, 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 [15].

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's architecture 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 [16]. For example, suppose that an encryption/decryption component must be introduced into the running ORB system of Figure 3, 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 instances of type Transport running in the system. At the sending ORB-side, an invocation request has to be encrypted, before being sent by a Transport component, and at the receiver ORB-side the invocation request must be decrypted after its receipt by a peer Transport component. Figure 9 shows the basic scheme of wrapping when handling an outgoing invocation request that must be send to a remote peer confidentially.

The wrapper in Figure 9 codifies which additional context dependency interfaces (i.e. Encryption) the type of the Transport component type must be widened with. Furthermore the wrapper also contains codified logic that implements how the interaction behavior of the existing Transport component instance(s) must be extended to incorporate the services of the newly introduced component. The on-line reconfiguration manager enables then a system administrator to "inject" and connect this wrapper on-line into the running ORB system at the chosen intercession point.

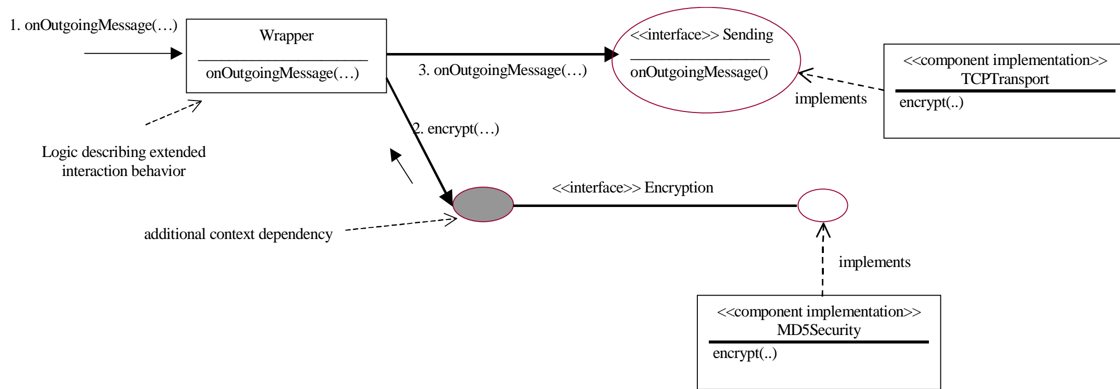

Figure 9 Wrappers

5.2. A variation point for injecting wrappers at run-time

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 [17], 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., [17].

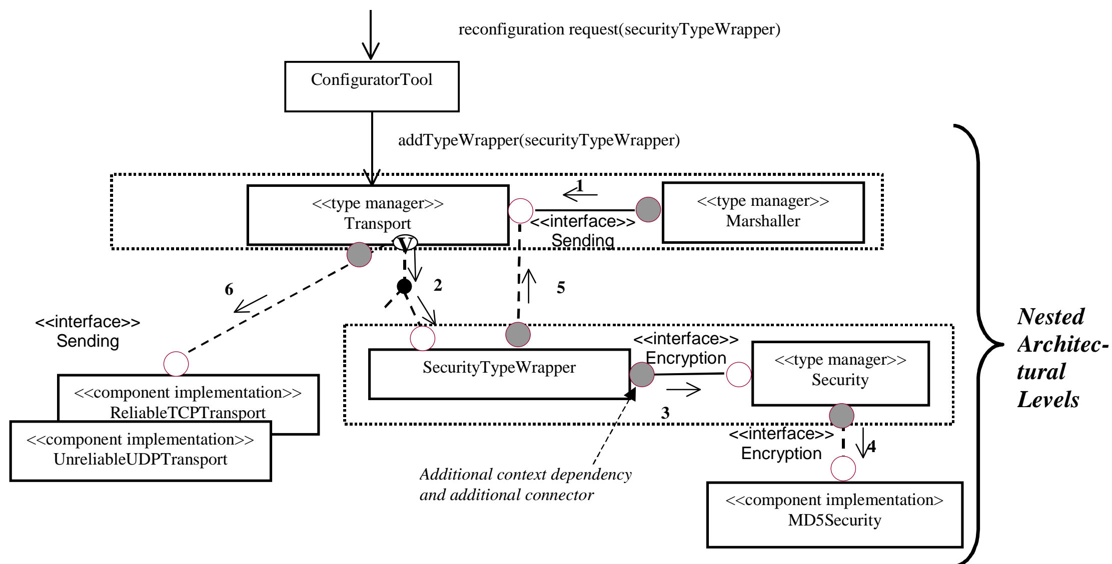

Figure 10 Run-time wrapping

Based on our reflective architecture presented in section 4.2, we can deal with this simply by incorporating a specifically designed variation point that is able to integrate wrappers at run-time by cooperating with the reconfiguration manager. Figure 10 illustrates this for the example wrapper depicted in Figure 9.

Obviously, the wrapper must first be implemented before it can be injected into the running ORB system. It is important to realize that the wrapper is implemented completely in terms of the context dependency interfaces it depends on. When the wrapper is then injected into the running ORB system the reconfiguration manager makes sure that object references for these interfaces are set.

The variation point has a method for accepting the wrapper from the on-line reconfiguration tool. Before passing the wrapper to the variation point, the reconfiguration manager first has to install a type manager for the newly introduced component MD5Security. Type information 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 reconfiguration manager first may check whether the two constraints formulated in [17] are not violated. If no type conflict is found, the injection of the wrapper into the running ORB system can effectively be performed.

The intercession point for the wrapper is the type manager of the component instances that are to be wrapped (i.e. type manager Transport). The reconfiguration manager is responsible for connecting the additional context dependency interface Encryption of 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 variation point. 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 strata* 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. As such, an object (thus a component) is able to morph its interface over the different strata at run-time.

Figure 10 shows furthermore the control flow of how the variation point applies the wrapper on the Transport component instances. The wrapper must of course only be applied on invocation requests that carry an application-specific policy specifying the need for confidentiality. To ensure this, the only thing the system integrator has to do is registering the wrapper with the appropriate component descriptor and the aspect-based forwarding mechanism will do the rest. The variation point also must know whether the wrapper must be applied before or after the wrapped components. Furthermore the variation point must apply wrappers in a certain order such that the required ORB semantics are not violated.

6. Related work

The discussion is organized accordingly to the related fields that our work touches.

Flexible ORB architectures: FlexiNet [6] is a Java-based middleware platform featuring a component based ‘white-box’ approach that supports reflection and introspection at all levels. This allows programmers to tailor the platform for a particular application domain or deployment scenario by assembling strongly typed components either at run-time or compile-time. However, reconfiguration has to be initiated from within code provided by the application programmer. Our approach allows the system itself to initiate reconfiguration based on reflection.

A reflective architecture for configurable middleware platforms is presented in [5]. The

approach relies on reflective techniques for reconfiguring the ORB to the desired behavior. Reconfiguration is done by configuring the meta-level of the reflective architecture, while the ORB implementation is residing at the base-level. However, reconfiguration is limited to application wide concerns and cannot be addressed on a per remote method invocation basis.

LegORB is a component-based ORB [15] that can be configured at run-time so that it loads just enough components to provide the middleware services required by its applications. These applications typically run on embedded devices with small memory footprints. A component library provides a range of different implementations for each ORB component that application programmers can choose from. Run-time configuration is achieved by defining pre-existing hooks for changing strategies related to concurrency, security, monitoring, and the like. This implies that only anticipated change can be handled whereas our approach is able of coping with unanticipated change as well.

Dynamic reconfiguration: To our knowledge, current research [19][20] in the area of run-time reconfiguration does not report about techniques for unanticipated run-time reconfiguration. Furthermore, they do not offer an easy usable specialization interface to application programmers that allow them to drive the reconfiguration process.

Aspect-Oriented Programming (AOP): 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 [21] and the lessons learned from building dynamical re-configurable ORBs, we argue that an explicit representation of some aspects at run-time is required to capture the dynamic preferences of an application.

Composition Filters [22] 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.

Meta-Level Architectures: In computational reflection, a Meta-Object Protocol is a generally applied technique for opening up system implementations on a system-wide scale [8]. However in general Meta-Object Protocols enforce a strict separation of concerns that makes it difficult to fine-tune the meta-level to application-specific preferences. In our approach, application-specific policies deliver a mechanism to break this separation of concerns in a disciplined and controlled way [23].

Programming languages and mechanisms: The ingredients to implement the presented component architecture efficiently within the implementation of a programming language are scattered across different research projects. The prototype-based language NewtonScript [24] provides useful object-based inheritance mechanisms that can be used for implementing the recursive tree structure. The language LAVA (a hybrid between prototype-based and class-based languages) [17] implements an object-based delegation mechanism, promoting a composition of class-based objects to a true component that owns a common-self [9]. There are also some domain-specific languages that implement support for aspect-based forwarding. For example, the language DROL implements a time polymorphic invocation mechanism, that enables programmers to express timing constraints on a per remote method invocation basis.

Researchers at North Eastern University Boston [25] have developed a programming construct that makes interactions (collaborations) between a set of classes/objects explicit, promoting these interactions as reusable components over hierarchies of classes.

Finally researchers at Vrije Universiteit Brussel [26] are developing mechanisms for extracting structural information out of object-oriented systems.

7. Conclusion

In this paper we presented a novel reflective component architecture that enables us to build ORBs that can be dynamically reconfigured to support different non-functional requirements on a per remote method invocation basis. The new component architecture solves the problems of current component architectures with respect to dynamic reconfiguration by separating architectural knowledge from purely functional behavior. This separation enables us to cope with anticipated as well as unanticipated change.

At the moment we are implementing an integrated development environment (IDE) for building component-based applications based on our component architecture. This IDE presents a virtual development and deployment workspace that is structured according to five different *views* offered to ORB architects, ORB builders, application programmers, and system integrators. The purpose of the individual views are as follows: The *software architecture view* supports the *ORB architect* in developing a generic ORB architecture by providing a visual environment for composing component types together. The ORB builder uses the *functional view* to create a new ORB implementation by providing one or more component instances together with their respective component descriptors for each component type. In applying the *non-functional view,* application programmers specify application-specific policies for the non-functional requirements that were identified during the use-cases analysis of their applications. The processes of specifying XML-compliant DTD's for component descriptors and policies are supported through an editing tool. The *component view* offers the system integrator the possibility to adapt the ORB at run-time by selecting and replacing component instances. Finally, the ability to cope with unanticipated change is supported through the *interaction refinement view*. This view allows the integration of new protocols and services that were not anticipated for during design of the ORB architecture.

On a longer perspective we intend to apply the proposed component architecture in other application domains that are subject to both anticipated and unanticipated changes.

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] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware", Design Patterns in Communications, (Linda Rising, ed.), Cambridge University Press, 2000.

[5] G.S. Blair, G. Coulson, P. Robin, M. Papathomas, "An Architecture for Next Generation Middleware", Proc. IFIP International Conference on Distributed Systems.

[6] 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.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns, Elements of Reusable Object-Oriented Software", Addison-Wesley, ISBN 0201633612.

[8] 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.

[9] C. Szyperski, "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, ISBN 0-201-17888-5.

[10] 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).

[11] T. D. Meijler, O. Nierstrasz, “Beyond Objects: Components”, in Cooperative Information Systems: Current Trends and Directions, M.P. Papazoglou, G. Schlageter (Ed.), Academic Press, Nov., 1997, pp. 49-78.

[12] F. Matthijs, “Component Framework Technology for Protocol Stacks”, Phd. Thesis, K.U.Leuven, ISBN 90-5682-224-1.

[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] I. Jacobsen, M. Griss, P. Jonsson, “Software Reuse; Architecture, Process and Organization for Business Success”, Addison Wesley 1997, ISBN 0-201-92476-5.

[15] 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).

[16] J. Brant, B. Foote, R. e. Johnson, D. Roberts, “Wrappers to the Rescue”, Proceedings of ECOOP’98. Springer LNCS 1445.

[17] G. Kniesel, “Type-Safe Delegation for Run-Time Component Adaptation”, In R. Guerraoui (Ed.): Proceedings of ECOOP99. Springer LNCS 1628.

[18] C. Atkinson, T. Kühne, C. Bunse, “Dimensions of Component Based Development”, in Proceedings of the 4th International Worshop on Component-Oriented Programming.

[19] 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.

[20] Jakob Hummes and Bernard Merialdo, “Design of extensible component-based groupware”, in Computer Supported Cooperative Work - An International Journal, 1998.

[21] 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.

[22] 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.

[23] B. Robben, B. Vanhaute, W. Joosen, P. Verbaeten, "Non-Functional Policies", In Proceedings of the Second International Conference on Metalevel Architectures and Reflection, Saint-Malo, France, July 1999.

[24] W. R. Smith, “Using a Prototype-based Language for User Interface: The Newton Project’s Experience”, in Proceedings of the 95 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. (OOPSLA’95).

[25] M. Mezini, K. Lieberherr, “Adaptive Plug-and-Play Components for Evolutionary Software Development”, in Proceedings of the 98 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’98).

[26] R. Wuyts, “Declarative Reasoning about the Structure of Object-Oriented Systems”, in Proceedings of TOOLS USA '98.