Component Architecture for Dynamic Reconfiguration of Object Request Brokers

Eddy Truyen, Bo Nørregaard Jørgensen, Frank Matthijs, Wouter Joosen, Pierre Verbaeten

Dept Comp. Sc. K.U.Leuven
Celestijnenlaan2000, 3001 Leuven, Belgium

MIP, University of Southern Denmark, Odense
Campus, DK-5230 Odense M, Denmark.

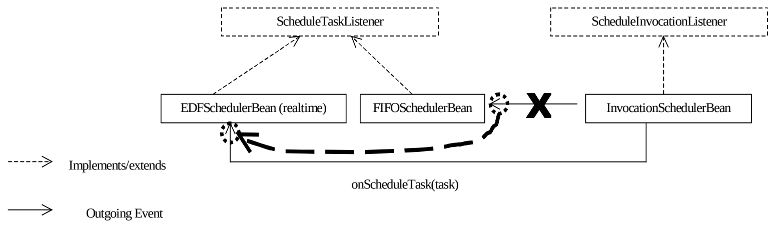
Problem Statement

Support for run-time reconfiguration of component-based systems is a necessity for achieving flexible component-based Object Request Brokers that incorporate support for various non-functional requirements on a per application use-case basis. However, current component architectures do not meet the requirements of run-time reconfiguration of component-based systems. In fact, current dynamic reconfiguration practices in component-oriented programming often result in ad-hoc solutions.

Introduction

In previous projects we build flexible communication systems and Object Request Brokers implemented in Java [1][2]. In our approach the process of customizing an ORB is driven by application-specific policies that describe QoS preferences for various non-functional requirements. An ORB implementation is then dynamically constructed by selectively integrating those ORB components that provide the expected behavior [3].

In order to implement this conceptual customization process, support for dynamic reconfiguration is needed. However, when implementing a prototype of a component-based ORB using JavaBeans, we ran into several issues that inhibited the realization of the required dynamic reconfiguration mechanisms. The basic problem was that the current component architecture of JavaBeans does not meet the requirements for dynamic reconfiguration. We will demonstrate this by giving an example. Consider an end-user application that has timeliness requirements for only a few of its use-cases. Further, suppose the used component-based ORB is initially configured with a FIFOTaskSchedulerBean that implements a non-real-time scheduling algorithm. When the InvocationSchedulerBean receives an IncomingInvocationMessage event[1] for method invocation within a specified 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 RealtimeTaskSchedulerBean.



However, when implementing this reconfiguration using JavaBeans several issues arise. 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 EDFScheduler-Bean with the InvocationSchedulerBean. However, a Java Bean is only

[1] In JavaBeans components are connected together through a simple push-event model. Event listeners are registered directly with event sources.

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 somewhere. Furthermore, 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.

The underlying source of the problems is the fact that JavaBeans fails to localize knowledge of system structure, independently of the elements being structured; and secondly, there is no infrastructure that allows you to realize atomic primitives for controlling and changing this knowledge. We refer to this knowledge as *Architectural Knowledge (AK)*. We distinguish between three categories of architectural knowledge, in relation to the JavaBeans component model. For a detailed analysis of the problems and solutions for each category, we refer to [4]. The most important architectural knowledge consists of:

- The collaboration flow between the components of the ORB system
- The management of listener references and source references: who is registered to whom?
- The component type of each component in the ORB system

Component Architecture

In [4] we present a novel component architecture that provides uniform support for run-time reconfiguration of component-based Object Request Brokers driven by application-specific policies. This component architecture is based on a localization of architectural knowledge embedded within a component-based system, making this knowledge controllable and changeable. This is realized 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 belong to a specific component type. Second there is an implementation-level that consists of component instances that implement the functionality of the system.

Type managers are themselves implemented as Java Beans that can be code-generated by a parser that inspects the component type specification of each component in the system. Type managers implement the same interfaces as its component instances. In order to observe and manipulate AK of component instances, we implemented a simple interception mechanism that allows the type manager to intercept outgoing and incoming event messages send and received by its component instances. This interception mechanism is similar to techniques applied in computational reflection, except that we don't perform 'reification' of event messages, but only forward the messages to the type manager. This is possible because each type manager implements in addition the interfaces on which its component instances explicitly depend on.

The component architecture supports selective integration of component instances based on application-specific policies. We call this technique aspect-based forwarding. Furthermore, the component architecture also copes with the integration of new component types into the system that were not anticipated for by pre-existing hooks. In order to interact with component instances of this new type, the interaction behavior of the existing component instances must be adapted. This is realized by the combination of applying a simple wrapper to existing components together with rewiring the input connections of the wrapped components to the wrapper. This rewiring is completely handled at the architectural level without type safety breaking.

References

[1] Eddy Truyen et. al., “Open Implementation of a Mobile Communication System”, In Proceedings of the ECOOP' 98 Workshop on Mobility and Replication, July 1998, Brussels, Belgium. http://www.cs.kuleuven.ac.be/~eddy/mp/smove.html

[2] B. N. Jørgensen, W. Joosen, "Dynamic Scheduling of Object Invocations in Distributed Object Oriented Real-Time Systems", ECOOP'98 Workshop Reader on Object-Oriented Technology, LNCS Springer, 1998.

[3] 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).

[4] Eddy Truyen, Bo N. Jørgensen, Wouter Joosen, "Customization of Object Request Brokers through Dynamic Reconfiguration", submitted to Tools Europe 2000.

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

[6] 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.

[7] 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.

Appendix: ORB development and deployment process

Figure 1 gives an overview of the component architecture. The component architecture is structured according to the different *views* used by developers/programmers/end-users in developing component-based systems. In ORB development and deployment we differentiate between three developer roles. These roles include the *ORB system architect* (who builds a domain-specific ORB component framework), the *ORB developer* (who builds an ORB as a specialization from the framework by plugging in component instances), and *end-user/application programmers* (who customize the ORB through policies). We differentiate between five views: the software architecture view, the component view, the functional view, the non-functional view, and finally the interaction refinement view.

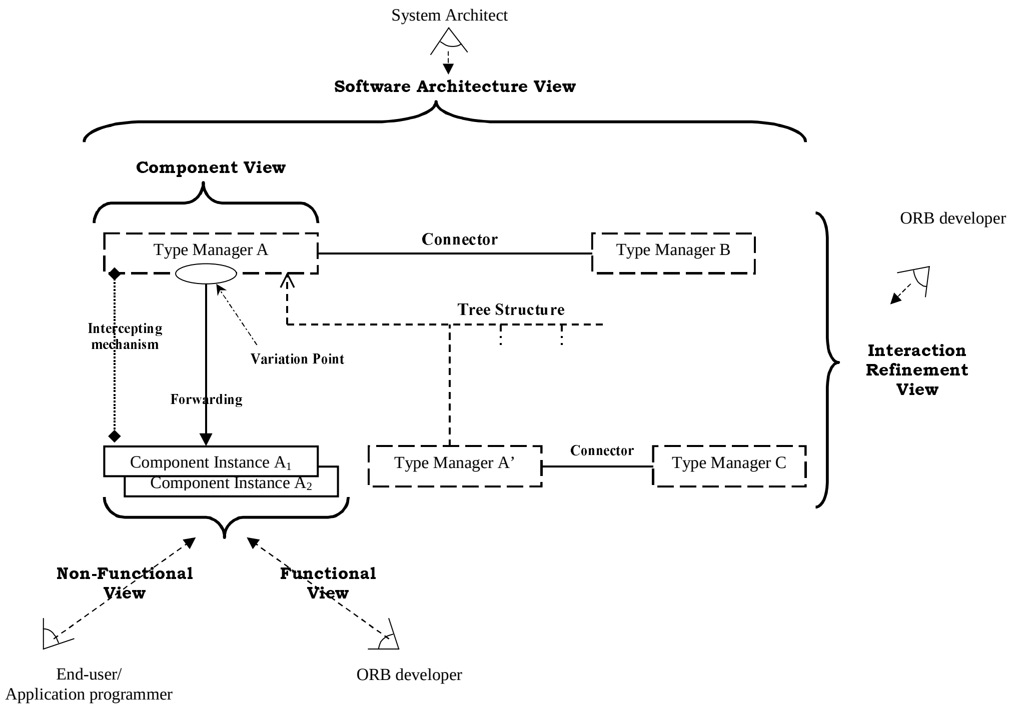


Figure 1 Component Architecture

Typically ORB implementations are constructed as a specialization of a domain-specific ORB component framework that is tailored to a specific application domain, such as E-commerce, robotics control etc. In designing this framework, the ORB architect anticipates ‘hot spots’ and encapsulates them within component types. In applying the *software architecture view*, an ORB architecture is then set up by selecting component types and connecting their interfaces together with connectors. Since a component type is explicitly represented by a type manager at the architectural level, the collaboration control flow between components is completely controlled at the architectural level. Furthermore in ORB development connectors typically implement the concurrency model of the ORB (how threads are flowing through the ORB)[5], keeping this difficult aspect away from the implementation level. This provides the ORB developer with a pure functional view that allows him to focus solely on the functionality that his component instances have to provide. From this *functional view*, the ORB developer creates an ORB implementation as a specialization of the component framework, by providing one or more component instances for each component type. Component instances vary in their support for non-functional requirements.

In applying the *non-functional view*, application programmers specify application-specific policies for the non-functional requirements that are necessary for the different use-cases of their applications. Application-specific policies are expressed using a specific aspect language that is specifically designed for a certain non-functional requirement. This is inspired by Aspect-Oriented Programming (AOP)[7]. AOP 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. However AOP is a static approach. Based on earlier experiences [6] and the need for dynamic reconfigurability of ORBs, we argue that an explicit representation of some aspects at run-time is required to capture dynamic preferences of an application. As such, customization of an ORB should better be performed by *run-time weaving* of policies into the ORB implementation. Run-time weaving is based on a matching between policies and alternative component instances [3].

The *component view* realizes the notion of a component as the dynamic construct of a component type (implemented by a type manager) and a set of alternative component instances.

Finally, the *interaction refinement view* copes with unanticipated changes not foreseen by the system architect. The ability to cope with unanticipated change is a necessity in application domains where 7x24 availability is required.