Customization of Object Request Brokers by Application Specific Policies

Bo Nørregaard Jørgensen[♣], Eddy Truyen[♠†], Frank Matthijs[♠], 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.Truyen, Frank.Matthijs, Wouter.Joosen}@cs.kuleuven.ac.be

Abstract. This paper presents an architectural framework for customizing Object Request Broker (ORB) implementations to application-specific preferences for various non-functional requirements. ORB implementations are built by reusing a domain-specific component-based architecture that offers support for one or more non-functional requirements. The domain-specific architecture provides the mechanism that allows the ORB to reconfigure its own implementation at run-time on the basis of application-specific preferences. This mechanism is based on a run-time selection between alternative component implementations that guarantee different service-levels for non-functional requirements. Application-specific preferences are defined in policies and service-level guarantees are defined in component descriptors. Policies and component descriptors are expressed using descriptive languages. This gives application programmers an easy and powerful tool for customizing an ORB implementation. To validate the feasibility of our architectural framework we have applied it in the domain of robotic control applications.

1 Introduction

The success of distributed object technology in time-critical distributed systems, such as robotic manufacturing systems, depends on the advent of Object Request Brokers (ORBs) that integrate support for 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

[♣] This research was supported in part by the A. P. Møller and Chastine Mc-Kinney Møller Foundation, The Danish National Centre for IT Research and …
[†] … by a grant from the Flemish Institute for the advancement of scientific-technological research in the industry (IWT).

characteristics that the application should have. In industrial settings such additional requirements include reliability and real-time.

In robotic manufacturing systems various non-functional requirements have effect on the exchange of control messages. Control messages can be simple activation and deactivation commands or commands containing isochronous data. Isochronous data is characterized by being equidistant in time and requiring processing at equal time intervals. In advanced model-based robotics [1] motion planning and joint control result in control messages that contain isochronous data. Distributing these messages in a timely manner requires real-time support from the ORB. By now, it is well known that conventional ORBs like CORBA [2], DCOM [3], Java RMI [4] are not designed to cope with real-time requirements [5].

The development of ORBs that support vertical integration of non-functional requirements from the application level all the way down to the network layer is crucial for successful application of distributed objects in robotic manufacturing systems. To deal with the wide range of non-functional requirements, 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 its implementation. Previous work has shown that meta-level architectures are a powerful technique for opening the ORB's implementation to the application programmer [6]. However, full-scale meta-level architectures make the customization process more complex than most application programmers can comprehend. This results from the inherited complexity of reflective systems and the non-trivial protocols and algorithms used to implement an ORB. As a result, it is very hard for application programmers, who are typically not experts in meta-level architectures or ORB development, to create specialized ORBs that satisfy their needs [7]. One way of solving this problem is to provide tools that allow the application programmer to customize an ORB without requiring him to understand the inner working of an ORB.

The approach we propose is based on architectural support for dynamic reconfiguration of ORB implementations. Reconfiguration of the ORB is based on policies that describe the non-functional requirements specific to the application and descriptors that specify how non-functional requirements are supported by the alternative implementations available for each ORB component. Policies and component descriptors are defined with a specific language for expressing non-functional requirements. At run-time the ORB interprets the policies and descriptors to select the right components for configuring its implementation.

An important characteristic of our approach is that it does not enforce a particular ORB architecture on the ORB developer but allows him to create the ORB architecture that is most appropriate for his specific application domain. The rationale behind this thought is that one size does not fit all. On the contrary, some application domain may require some ORB components that are not present in other domains. For example, embedded systems need compact ORBs with a small footprint, while e-commerce applications need ORBs that support data integrity, authentication, and authorization of remote method invocations.

This paper is organized as follows: section 2 gives an overview of the proposed approach. Section 3 describes each of the elements of our architectural framework. In section 4 we exemplify our architecture by showing how it can be used to create a

customized ORB for a robotic control system. Related work is described in section 5. Finally, section 6 concludes.

2 Overview of Our Approach

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 the system. However, non-functional requirements, such as reliability, availability, performance, security, or real-time are equally important for establishing a system that can deliver the expected quality of service. Non-functional requirements should be dealt with during analysis and design and should not be postponed until the implementation phase. During use-case analysis some considerations about non-functional requirements often come up. For instance, when describing the use-case for an ATM Cashier system the domain expert may very well ask himself whether or not the transaction responsible for money withdrawal should use a secure line to the main server. By extending the use-case analysis phase to include the specification of non-functional requirements, the domain expert can record non-functional requirements together with the functionality they apply to.

This paper presents an architecture that offers an easy but powerful way for integrating those non-functional requirements into ORB implementations. In the rest of this section we describe the important features of our work.

2.1 Architectural Framework for Domain Specific ORBs

Since distributed technologies are nowadays applied in almost every application domain, one general ORB architecture that is put forward as a fit for all applications, is not realistic. Instead ORBs should be developed for a specific application-domain or for a family of applications, incorporating support for only those non-functional requirements that are relevant for that specific application domain or family of applications.

Our approach supports this idea by providing the ORB developer with a domain-specific component framework that defines a basic ORB architecture that is tailored for a specific application domain or family of applications. The ORB developer uses this basic architecture to build an ORB implementation that realizes all the non-functional requirements recorded during the use-case analysis phase. However, since the non-functional behavior is not necessarily the same for all parts of the application, it is essential that the ORB implementation can be dynamic reconfigured with respect to how non-functional requirements are realized. To enable this, we define the basic ORB architecture in terms of architectural entities that abstract away from concrete implementation details. This is possible by differentiating between the notions of *component types* that constitute such architectural entities and *component instances* that realize implementations of component types. Each component type defines a set of contractually specified interfaces that describe the external characteristics of the

architectural entity, without stating anything about its implementation. The architecture of an ORB is then defined as a static composition of component types that are connected appropriately through their respective interfaces. A component instance provides a specific implementation for a specific component type. There can be more than one component instance per component type: various component instances can differentiate in the non-functional requirements they support and how this support is implemented.

Building an ORB implementation that realizes flexible support for the subset of non-functional requirements, identified in use-case analysis, is then a matter of instantiating the basic architecture with those component instances that provide the expected service-level for each non-functional requirement.

2.2 Policies and Component Descriptors for QoS Specification

A second feature of our approach is that we use a descriptive language for specifying Quality of Service (QoS) expectations of applications and QoS guarantees delivered by component instances. QoS expectations reflect application specific preferences to how well the system must perform with respect to a specific non-functional requirement. QoS guarantees describe the service-level of a component instance for one non-functional requirement. In our approach, QoS expectations are defined in a policy, while QoS guarantees are specified in a component descriptor. For each non-functional requirement, a separate descriptive language is used. Hence policies and component descriptors are defined per non-functional requirement.

The application programmer defines specific policies for each method that takes part in realizing the use-cases. An application specific policy specifies how the non-functional requirement should be handled for the method that it applies to. Hence, an application specific policy will be enforced per remote method invocation. Similarly the ORB component developer defines component descriptors for the component instances that he implements.

2.3 Dynamic Reconfiguration of ORB Implementation

Dynamic reconfiguration of an ORB implementation is supported by a run-time selection mechanism between alternative component instances. This mechanism is implemented within the component type as a generic variation point. A detailed discussion of variation points can be found in [8]. The variation point performs the selection on a per method invocation basis by comparing policies with component descriptors. In our approach, method invocations are reified as typed objects that offer introspection facilities for accessing parameters, method names, destination, invocation context attributes, etc. This is done by stub objects, which also attach to the reified method invocation the policies for that method. This provides the variation point with the information it needs to make appropriate tradeoffs when selecting between alternative component instances. Each variation point bases its choice of component instance on a comparison between the application specific policies

associated with the remote method invocation and the component descriptors provided by the alternative component instances.

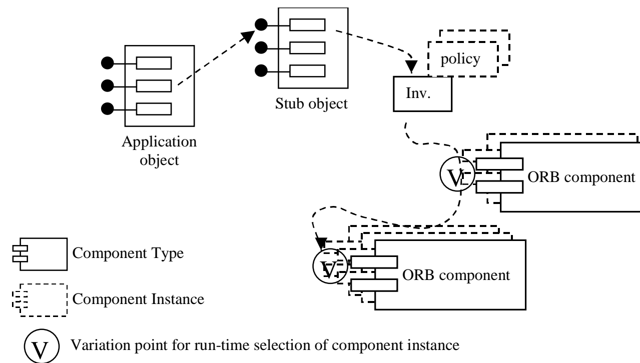

Fig. 1. Flow of reified invocation and policy object

Consequently, the customization of the ORB is controlled at runtime by the application specific policies associated with the methods in the application objects. Fig. 1 shows how the remote method invocation reified by the stub object traverses through the ORB together with its associated policy objects. Propagating the policy objects along with the reified remote method invocation allows all variation points to make the proper choices with respect to the selection of the suitable component instance.

3 Architecture for Customization of Object Request Brokers

In this section we discuss each of the basic building blocks of our architectural framework in detail.

3.1 Variable Features in ORB Design

In general, the implementation of an Object Request Broker can be described as a collection of features. A feature corresponds to an identifiable part of the ORB functionality. Examples of such features are marshalling, invocation scheduling, routing of invocations, etc.

In conventional ORB design, ORBs are viewed as black boxes. This information hiding principle helps during ORB development, but it locks in decisions that can effect QoS, after which those decisions are not readily reexamined. For instance, in robotic manufacturing systems, some remote method invocations have strict timing requirements. Hence the choice of scheduling algorithm in the ORB can effect whether the ORB implementation is acceptable for such systems. In our opinion, an ORB implementation must be designed for change by allowing different variants of the implementation for one or more of its specific features. In the rest of this section

we give a non-exhaustive list of features that we believe are subject to variability, limiting the scope to those features that are related to the implementation of remote method invocation. The list of features covers ORBs as well as protocol stacks.

Invocation dispatching Invocation dispatching refers to the process of calling the method corresponding to the invocation on the servant that implements the remote object. Dispatching provides an interception point for reflecting on invocations at the server side.

Marshalling Refers to the process of taking a collection of objects and assembling them into a form suitable for transmission in a message. During marshalling objects can be replaced, like it is the case with stubs in Java RMI. Furthermore marshalling can be extended to perform data compression or encryption.

Unmarshalling This is the reverse process of marshalling. Unmarshalling is the process of disassembling a marshalled message to produce an equivalent collection of objects at the destination. When resolving an object during unmarshalling, it can have its attributes modified or it can be replaced with an equivalent object.

Invocation Context This is a reification of the runtime context in which the invocation takes place. The invocation context is often used to associate non-functional requirements with the invocation, such as security context, priority, user preferences, etc. In the CORBA specification the functionality of an invocation context is provided by the Context object abstraction [2].

Invocation semantics In distributed systems asynchronous invocation semantics can be preferable, since synchronous invocation semantics can result in unnecessary delay at the caller side. Therefore both synchronous and asynchronous invocation semantics should be supported by the ORB.

Invocation scheduling The decision on whether or not an invocation is to be executed may depend on different factors, such as, the state of the servant (preconditions), the priority associated with the invocation if any, the CPU load of the node (resource admission control), etc.

Threading The number of threads available for executing object invocations determines the degree of concurrency within the system. If only one thread is available for executing object invocations, a purely sequential system is the result. In contrast, multiple threads result in a truly concurrent system.

Channel Handle the responsibility of maintaining a session between two address spaces. Session management comes in many flavors, for example object to object, node to node (multiplexed), one per invocation.

Reliability The kind of transport protocols available for transferring the invocation may vary depending on the underlying network technology or according to application domain specific requirements.

Routing According to the non-functional requirements of the application certain types of network technologies may be preferable. This includes Ethernet, ATM, Firewire, Canbus, etc. The availability of network technology is strongly dependent on the application domain. For instance, the use of Canbus is common in industrial automation.

3.2 Architectural Reuse in ORB Design

To facilitate the ORB development for a specific domain, a component framework is used that offers a component-based ORB architecture that is tailored for that specific domain. The architecture defines a set of component types and how these component types cooperate together. In order to support variability, each component type reflects upon a particular variable feature of an ORB in an implementation-independent manner. In the implementation of the component type the variable feature is exploited at a variation point. For each component type, the ORB developer selects one or more component instances. The alternative component instances are characterized by different service-levels for each non-functional requirement.

For example, in the context of a robot control project [1] we have build an ORB component framework that defines an architecture tailored to real-time applications. This architecture is explicitly represented by a composition of Java Beans, where each component type is implemented as a separate Java Bean and component types cooperate together through various classes of events. Fig. 2 gives an overview of the architecture. It consists of the following component types:

ReferenceBean provides the support for the synchronous and asynchronous invocation semantics.

MarshallerBean is responsible of marshalling outgoing invocations and replies, and unmarshalling incoming invocations and replies.

ChannelBean is responsible for session management between address spaces.

TransportBean transmits messages containing invocations and replies between address spaces.

InvocationSchedulerBean determines the order in which to dispatch incoming invocations on the corresponding servant objects.

TaskSchedulerBean controls the threading strategy for executing all computations within the system. This includes computations related to the basic functionality of the

ORB (e.g. listening for incoming requests) as well as computations related to method execution on servant objects.

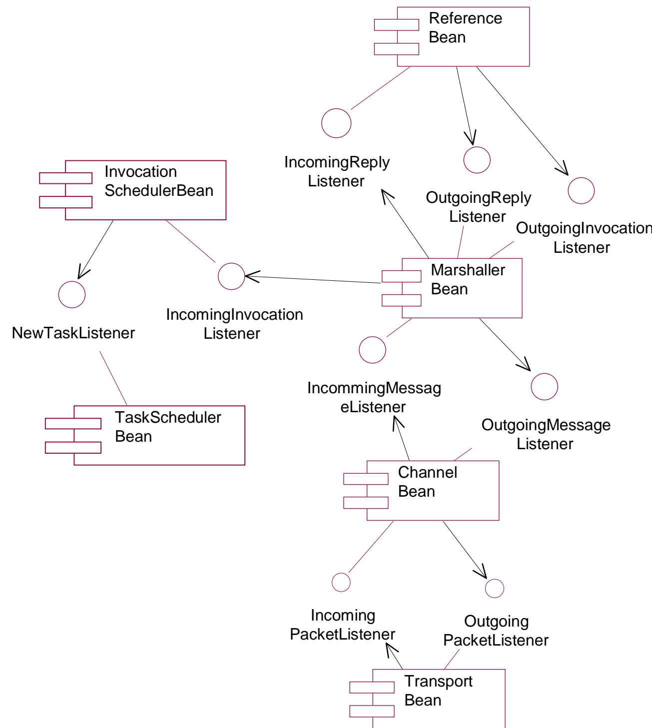

Fig. 2. Bean model of the ORB architecture.

A real-time ORB implementation is constructed as an instantiation from this architecture. The ORB developer just provides the component instances that have to be available for each component type and glue code within each component type connects a component instance - once selected - into the ORB implementation.

Component instances are also implemented as separate Java Beans that implement one feature of the ORB. The design decisions made by the component developer determine its QoS-level for the different non-functional requirements.

3.3 ORB Customization Through Descriptive Languages

ORBs have to take application-specific information into account, to achieve optimal performance. In our approach, application-specific QoS expectations with respect to the implementation of a specific non-functional requirement are defined in a policy. The ORB implementation tries to offer the requested QoS expectations by integrating those components that guarantee the expected service level for that non-functional

requirement. In this way, by choosing appropriate individual components, the overall ORB implementation is tailored to the application-specific QoS expectations. QoS guarantees provided by a specific component are defined in component descriptors that are packaged together with the component. As for policies, component descriptors are specified per non-functional requirement.

The definition of policies is the task of the application programmer, whereas the definition of component descriptors is the task of the ORB component developer. However, they are both declared at a high level of abstraction in the same specialized language. The vocabulary of such a language is defined by the ORB developer as a general template. Application-specific policies and component descriptors are then defined using this template. This means that their interpretation is done in terms of the vocabulary defined by the template. The use of templates keeps the variation point independent of specific characteristics of non-functional requirements, as well as component instance implementations. As a consequence, a generic mechanism for realizing variation points can be offered to the ORB developer. Note that the ORB developer has to define a template for each non-functional requirement that he wants to take into account.

3.3.1 Defining Templates, Policies and Component Descriptors

A template defines the vocabulary of a language for describing one specific non-functional requirement. The vocabulary is defined as a set of parameters that can be used to specify QoS expectations and guarantees for one non-functional requirement. The possible service-levels available for each parameter are defined as an enumeration. Each service-level can be further refined by associating it with a number of attributes. For example, for reliability you could define a parameter called *tolerance* that can have three different service-levels: “NONE”, “NOT_TRANS-PARENT”, “TRANSPARENT”. The service-level “TRANSPARENT” has an attribute for specifying the number of faults that are allowed. Another parameter is the *fault type*. For this parameter three different service-levels can be defined: “FAIL_STOP”, “BYZANTINE”, and “TIME”.

Defining a policy from a template consists of specifying the service-level for one or more parameters and setting the associated attributes. A policy only has to define the number of parameters from its template that are necessary to specify the QoS expectations of its associated application method. Parameters that are not defined are assigned a default service-level, by default this is the first service-level from the parameter’s enumeration in the template. The process for defining a component descriptor is similar. Each component implementation can have more than one component descriptor, since it can have been built to support more than just one specific non-functional requirement. For instance, a marshalling component can provide support for real-time requirements as well as security requirements, but it doesn’t has to do so. Examples for defining policies and component descriptors are given in section 4.

Policies and component descriptors are transformed into objects by parsing their definitions. The corresponding class diagram is shown in Fig. 3.

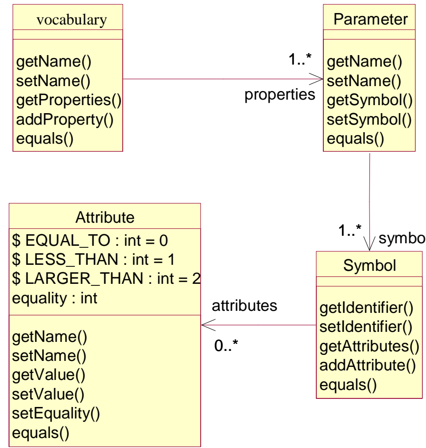

Fig. 3. Class diagram for representing policies and descriptors

3.3.2 Matching Policies with Component Instances

At run-time, all application specific policies that apply to a method are grouped together. The variation points within the ORB component types traverse this group to find the policy objects that influence their choice of implementation.

To make the best match the variation points apply a mapping function to the application specific policies provided for the method and the component descriptors of the component instances of the implicated component type. At each variation point that component instance is selected whose component descriptors make the "best match" with the application specific policies. The best match for an application specific policy is the component descriptor that has the most parameters that match the parameters in the policy. Matching is based on the notion of equality; that is, a component descriptor matches a policy if the attributes associated with the service-levels of its parameters equal with the corresponding attributes of the service-levels defined by the policy. For instance, assume that the value of an attribute in a component descriptor always has to be larger than the value of the attribute in the corresponding policy. This relationship is expressed by setting the equality relation of the attribute in the component descriptor to LARGER_THAN and the equality relation of the attribute in the policy to LESS_THAN. The matching function will then verify that the relation holds for the actual attribute values. An appropriate set of keywords is provided for specifying such equality relations when defining policies and descriptors.

The advantage of the 'best match' strategy is its generality. It is implemented once and for all in one variation point and this variation point can immediately be used in any component type. However, when dealing with more complex cases, the 'best match' strategy may not be sufficient. Examples include the cases where none or more than one component instance fails to completely match all QoS parameters for a non-functional requirement. In these cases, different selection strategies could be

preferable over the ‘best match’ strategy. For instance, when no component instance makes a complete match, the selection strategy could prefer an instance that performs weakly on some parameters rather than selecting one which fails completely on one parameter.

Another complex situation occurs when combining non-functional requirements that are not orthogonal. Non-functional requirements that are not orthogonal introduce constraints that have to be taken into consideration by specialized variation points that are able to enforce these constraints. For instance, when policies for real-time and security are applied simultaneously for a method, an invocation of the method may miss its deadline due to the additional overhead induced by encryption and decryption. This constraint can be taken into account if the framework architect constructs a third template that defines the vocabulary for expressing such a constraint. Using this template the application programmer can for example specify a desired upper limit for encryption overhead, leading to the definition of a third 'overlapping' policy. This provides the variation point with the information it needs to make a good choice between component instances, without breaking the constraint. Here again, different specialized selection strategies can be used. For example, one variation point could decide - when the constraint is violated - to decrease the required security level in favor of the timeliness requirement. One could also consider a variation point that is able to customize its component instances by forwarding the constraint information via a contractually specified meta-interface that the component instances export. Which selection strategy is best, is however often not determinable until the time of instantiating a specific ORB implementation from the ORB component framework. Hence, component types must offer hooks that allow different variation point implementations to be plugged in.

4 Applying the Approach to a Time-Critical Application

In this section we show how our approach can be used to customize an ORB for a distributed robot controller application. The robot controller is part of the SmartController project that addresses the development of a generic robot controller for arbitrary robotic manipulators [1]. The robot controller is built as a component framework, based on an extension of the JavaBeans model [9].

Basically, there are two primary functional aspects that a non-trivial robot controller should take care of. First, there is the task of generating collision free motion for the robot within the work cell. The robot should not collide with itself or with the work piece. Secondly, there is the planning of the work that the robot has to perform on the work piece. This work is described by a process description that specifies the speed by which the robot should move over the surface of the work piece to perform the work correctly. Deviation from the specified speed will have an impact on the quality of the performed work. For instance in spray painting, deviation in the speed by which the spray gun is moved over the surface of the work piece will either result in a thinner or ticker layer of paint.

4.1 Defining a Template for Temporal Behavior

Object interactions within a real-time system can be characterized along the dimensions of timeliness, temporal behavior, and invocation precedence. Timeliness expresses whether an object interaction is time constrained. Temporal behavior specifies how often an object interaction occurs. Finally, invocation precedence specifies whether the next invocation of a method by the same caller is more important than the present one. Invocation precedence is useful when old information becomes obsolete as soon as new information is available. One example is proximity sensors. In robotics, proximity sensors provide information about the distance to nearby obstacles. The actuality of this information is crucial for collision avoidance.

In the context of this paper, timeliness of object interactions is classified by the two values:

REALTIME Response must be timely; that is, within a specified deadline. A late response will have undesirable consequences in the application domain.
NEUTRAL No timing constraint is imposed on the object interaction.

Timeliness says nothing about the magnitude of a timing constraint; it can be microseconds or weeks. The temporal behavior of object interactions is classified as:

PERIODIC Object interactions that take place at regular time intervals and that execute for a fixed amount of time. Each interaction has to finish before the end of its period.
SPORADIC Object interactions triggered by external events or internal state changes.

The precedence of subsequent invocations of the same method is classified as:

NEXT The next invocation has precedence over the present. The present invocation can be skipped if it has not begun execution before the next one arrives.
CURRENT The current invocation has to be finished before the next one is allowed to execute.

Based on these classifications we can define a template for specifying application specific policies for temporal behavior. Fig. 4 shows the template definition.

```
template TemporalBehavior {
   parameter timeliness enum NEUTRAL,REALTIME;
   parameter temporal enum SPORADIC,PERIODIC;
   parameter precedence enum CURRENT,NEXT;
   REALTIME attributes DEADLINE Long;
   PERIODIC attributes PERIOD Long;
}
```

Fig. 4. Template for specifying temporal behavior

4.2 Defining an Application Specific Policy for Temporal Behavior

To illustrate how application specific policies are instantiated from the temporal behavior template we apply it to two methods of the JointController component from our robot controller framework. The result is shown Fig. 5. The JointController component is responsible for applying the forces that describe the robot motion to the robot's joint actuators.

```
TemporalBehavior JointController.
   addSensorDataSubscriber(SensorDataSubscriber) {
   timeliness NEUTRAL;
   temporal SPORADIC;
   precedence CURRENT;
}
TemporalBehavior JointController.onForceReady(Force) {
   timeliness REALTIME attribute DEADLINE 100;
   temporal PERIODIC attribute PERIOD 100;
   precedence NEXT;
   relation DEADLINE larger than;
   relation PERIOD larger than;
}
```

Fig. 5. Temporal policy applied to an application class

The first part of the policy specifies that the method `addSensorDataSubscriber` is only invoked sporadically and that there is no timing constraint on the execution of the method. The second part specifies that the method `onForceReady` is invoked periodically and that the execution of each invocation is constrained in time. The precedence parameter tells that new force values are preferable over old ones. The equality relation for the deadline and the period attribute specifies that it must always be larger than the corresponding attribute provided by a component descriptor.

4.3 Defining Component Descriptor for Temporal Behavior

The ORB components that directly influence the temporal behavior of a distributed application are the components responsible for executing and transmitting remote method invocations. In our case these components are the TaskSchedulerBean and the TransportBean. Before we show the descriptors for these components, we describe each component in more detail to give the basis for understanding the meaning of these descriptors.

4.3.1 TaskSchedulerBean

Predictions about the system's temporal behavior can only be made if the execution of all computations within the system is coordinated. Coordination ensures that all computations advance, as they are required to. Introducing the concept of a task enables this. A task represents the basic unit of computation. Examples of tasks within

the ORB include listening for and receiving messages from the network, dispatching invocations to servant objects, etc. Hence, all execution within the ORB and its application is represented as tasks that are scheduled by the TaskSchedulerBean. The application programmer is not allowed to create threads within the application, since they will interfere with the scheduling done by the TaskSchedulerBean. Component instances of the TaskSchedulerBean can provide different scheduling guarantees. One component instance can be used for tasks that only require best-effort scheduling and execution; whereas, another component instance can be used for tasks that require real-time scheduling. Here real-time scheduling refers to either Early-deadline-first or Rate-monotonic scheduling [10].

The rationale for encapsulating the threading feature in the TaskSchedulerBean is the fact that if a time-critical application is built on top of an ORB that does not apply any strategy for coordinating the execution of threads, it can result in missed deadlines for important operations. In the task-based approach this situation is avoided by using a component instance which implements a real-time scheduling algorithm for executing time constrained tasks and a component instance that implements a non real-time scheduling algorithm for executing tasks that only require best-effort service. The execution of tasks scheduled by the real-time scheduling algorithm will be done in a thread given a real-time priority whereas execution of non real-time tasks will be done in a thread with normal priority. Component descriptors for two component instances of the TaskSchedulerBean that implements these different scheduling strategies are shown in Fig. 6. In our current prototype, the RealtimeTaskSchedulerBean uses a thread running at the highest priority.

```
TemporalBehavior FifoTaskSchedulerBean {
   timeliness NEUTRAL;
   temporal SPORADIC;
   precedence CURRENT;
}
TemporalBehavior RealtimeTaskSchedulerBean {
   timeliness REALTIME attribute DEADLINE 10;
   temporal PERIODIC attribute PERIOD 10;
   precedence NEXT;
   relation DEADLINE less than;
   relation PERIOD less than;
}
```

Fig. 6. Component descriptor for TaskScheduler component instances

4.3.2 TransportBean

The TransportBean is responsible for transferring object invocations to a different address space. The most interesting case is when the source and destination address spaces are located on different hosts. In that case, the object invocations are sent over the network, which is an important factor affecting the overall QoS guarantees an ORB is able to make. The TransportBean is implemented as a specialized instantiation of our DIPS protocol stack framework [11]. The framework can

instantiate dynamic protocol stacks which can cope with variability in much the same way as the global ORB architecture. The TransportBean is in itself a component framework with its own variation points. This nested structure has the advantage that each variation point inside this Bean can be individually described, while the TransportBean still fits in the global ORB architecture as one component which supports the global QoS concerns. The designers of the TransportBean have to determine which non-functional concerns they will support. For each non-functional aspect they support, they have to provide a component descriptor. In the case of our example, they will support the TemporalBehaviour template. The example of the TransportBean is interesting in that it shows what happens when an ORB component is itself built from components. In this case, the TransportBean consists of two component types, namely the RoutingBean and the ReliabilityBean. The rest of the TransportBean structure and functionality is not relevant for the discussion in this paper. We like to stress, though, that as the TransportBean is built with a flexible protocol stack framework, new versions can be built which expose additional internal component types, should the need arise.

We now describe the function of each of the two nested component types in more detail.

RoutingBean The RoutingBean is responsible for selecting the underlying network technology. A selection is made based on the application requirements and on the capabilities of the underlying communication technology. This resource-aware routing is a variation point in the TransportBean, therefore the TransportBean exposes the RoutingBean type. Depending on the type of remote method invocation that has to be sent over the network, a specific RoutingBean instance will be chosen. For example, a real-time invocation with a short deadline will be sent over a communication channel which can guarantee timely delivery, such as IEEE 1394 Firewire. Neutral invocations are sent over another channel if possible, for example, a cheap Ethernet connection, to avoid unnecessary usage of precious resources. The cases discussed in this example are handled by the FirewireRoutingBean and the EthernetRoutingBean component instances, respectively.

ReliabilityBean Many remote method invocations require reliable transmission. The TransportBean therefore includes a ReliabilityBean for managing acknowledgements and retransmissions. The retransmission strategy is an important ingredient of this component which has a strong impact on the ability of the TransportBean to provide QoS. Therefore, the TransportBean includes a variation point for the retransmission strategy by exposing the ReliabilityBean component type. As a result, a specific ReliabilityBean instance is chosen depending on the properties of the object invocation at hand. For example, for a periodic remote method invocation for which the precedence parameter has the value “NEXT”, the strategy takes into account the period of the invocation and it does not perform retransmissions when the next invocation is imminent. For sporadic invocations, a retransmission strategy such as the one included in TCP is more suitable. These cases are handled by the PrefernextReliabilityBean and the NormalReliabilityBean component instances, respectively.

In order to support the automatic component instance selection, the TransportBean component developer has to provide a component descriptor for every instance of both the RoutingBean and the ReliabilityBean. See Fig. 7 for the descriptors of the component instances from our example. The FirewireRoutingBean can cope with remote method invocations with real-time constraints, whether they are sporadic or periodic. The EthernetRoutingBean can only handle neutral sporadic invocations. The ReliabilityBean instances can each handle a different kind of precedence.

```
TemporalBehavior FirewireRoutingBean {
   timeliness REALTIME attribute DEADLINE undefined;
   temporal SPORADIC,PERIODIC attribute PERIOD undefined;
   relation DEADLINE less than;
   relation PERIOD less than;
}
TemporalBehavior EthernetRoutingBean {
   timeliness NEUTRAL;
   temporal SPORADIC;
   precedence CURRENT;
}
TemporalBehavior PrefernextReliabilityBean {
   precedence NEXT;
}
TemporalBehavior NormalReliabilityBean {
   precedence CURRENT;
}
```

Fig. 7. Component descriptors for the RoutingBean and the ReliabilityBean

In addition, the TransportBean itself needs a descriptor that describes its capabilities to the rest of the ORB. This descriptor is simply the combination of the capabilities of all internal component instances that make up the TransportBean. The result is given in Fig. 8.

```
TemporalBehavior TransportBean {
   timeliness NEUTRAL, REALTIME attribute DEADLINE undefined;
   temporal SPORADIC, PERIODIC attribute PERIOD undefined;
   precedence CURRENT, NEXT;
   relation DEADLINE less than;
   relation PERIOD less than;
}
```

Fig. 8. Component descriptor for TransportBean component implementation

This descriptor basically means that our TransportBean instance can cope with all kinds of remote method invocations. Its internal variation points and nested components will take care of it. Note that the values for the deadline and period

attributes are left out for the TransportBean. This means that the values are dynamically determined at run-time by the component instances.

4.4 Mapping Temporal Behavior to Component Instances

To exemplify how reconfiguration of the ORB works, we will discuss the invocation of two methods with different temporal behavior. Both methods belong to the JointController component from our SmartController framework.

When the method `addSensorDataSubscriber` is invoked on the JointController stub, the invocation is reified and the application specific policies associated with the method are retrieved. In the present case only one application specific policy has been associated with the method, namely an instance of the TemporalBehavior template. Its policy object is now propagated along with the invocation down through the ORB. When the invocation arrives at the TransportBean at the client side, the RoutingBean and the ReliabilityBean within the TransportBean decide to transmit the invocation to the destination address space using the component instances EthernetRoutingBean and NormalReliabilityBean, respectively. The TransportBean makes this decision based on the value of the timeliness parameter, which is "NEUTRAL". At the server side, the TaskSchedulerBean assigns the Task responsible for executing the invocation is the component instance FifoTaskSchedulerBean. This assignment is based on the same reasoning.

Invocation of the method `onForceReady` leads to different choices within the TransportBean and the TaskSchedulerBean, due to the different values of the policy properties. Now the TransportBean chooses to use the component instances FirewireRoutingBean and PrefernextReliabilityBean for transmitting the invocation. This choice is made because the value of the timeliness parameter is "REALTIME". Similarly, the TaskSchedulerBean, at the server side, chooses to use the RealtimeTaskSchedulerBean component instance. In general terms the temporal nature of the method `onForceReady` can be characterized as isochronous. This is specified by setting the timeliness parameter to "REALTIME", the temporal parameter to "PERIODIC", and the precedence parameter to "NEXT". The retransmission algorithm within the PrefernextReliabilityBean can utilize this information to optimize its performance for transferring the force information. Here optimization consists in skipping retransmission of the current invocation in case of transmission failures if the next invocation has become available in the meanwhile.

This example illustrates that the temporal behavior of a method depends on its function within the application. Accordingly the ORB can not just handle all method invocations equally. Run-time reconfiguration of the ORB is necessary to meet the requirements of different methods.

5 Related Work

Related projects investigate ways of adding support for non-functional requirements to distributed object systems, although many of them are concentrating on specific

application domains, such as ReTINA [12] (telecommunications), or restrict themselves to non-functional requirements only concerning bandwidth and throughput, such as TAO [5]. The ReTINA project has developed a distributed processing environment for telecommunication applications that complies with the Telecommunications Information Networking Architecture (TINA) standard. ReTINA provides real-time audio and video, and network QoS guarantees. TAO is a real-time CORBA compliant ORB that provides end-to-end QoS by vertically integrating the ORB middleware with communication protocols and network subsystems. TAO is the first real-time ORB supporting end-to-end QoS over COTS platforms and ATM networks. Our research goals are much broader than the goals of those projects, since we believe the dynamic reconfiguration offered by our framework is applicable to a wide range of non-functional requirements.

Other related work, are projects where reflection and component-based techniques are used to achieve open Object Request Brokers. Researchers at APM have developed an experimental middleware platform called FlexiNet [13]. This platform allows the programmer to tailor the underlying communications infrastructure by inserting/removing meta-level components. Researchers at Illinois have developed dynamicTAO [14], a CORBA compliant reflective ORB that supports run-time reconfiguration. DynamicTAO maintains an explicit representation of its own internal structure and uses it to carry out dynamic reconfiguration. Reconfiguration is implemented by a so called TAOConfigurator that contains hooks to which implementations of dynamicTAO strategies are attached. In our opinion, policies can here be used to drive the configuration process implemented within the TAOConfigurator. At Lancaster University researchers conduct research about a generic reflective architecture for constructing open middleware platforms [6]. They define three distinct meta-object protocols that reify specific aspects of the middleware architecture: encapsulation, composition and environment. In their approach they build an ORB at the base-level, that can customize itself through the deployment of these three MOPs. However, a general problem of applying meta-level programming for application-specific customization, is that it's too complex for the average application programmer to comprehend [7]. We address this problem by introducing application-specific policies together with a reconfigurable ORB architecture. However, we think that our work is also complementary to this related work, since policies can be applied there as well.

Researchers at HPLabs have developed a general-purpose language for QoS, called QML [15]. QML has three main abstraction mechanisms for QoS specification: contract type, contract and profile. Although these abstractions are more generic than ours, the first abstraction is similar to a template and the last two abstractions are captured by a policy. They show how QML can be used to build a QoS-based trader that matches client requirements with QoS properties of services [16]. In this case, they don't use QML for customization of the underlying distributed platform.

6 Conclusion

In this paper we presented an architectural framework that can be used to implement domain specific ORBs that can be dynamically configured to support different quality of service levels for non-functional requirements. An important characteristic of the ORB architecture is that it allows each remote method invocation to be treated differently. This is particularly important in time-critical applications where some remote method invocations have timing constraints and others don't. For other applications, such as e-commerce the situation is similar. Here security is an issue for some remote method invocations and for others it is not. This variation between methods is easily expressed by defining policies that describe how non-functional requirements impact the invocations of each method. Our experiences with applying policies for constructing open communication systems [17][11], and customizable metalevel programs for non-functional requirements [7] have given us confidence in the feasibility of our architecture.

To validate our ORB architecture we developed a prototype that integrates the protocol stack with the ORB. We chose not to differentiate between the ORB and the protocol stack like conventional ORB implementations, where the protocol stack is a black-box because of the real-time requirements of our robot controller application. Only by integrating the protocol stack with the rest of the ORB can we be sure that the real-time requirements of the robot controller application are effectively enforced at all levels.

The approach we have presented provides a simple but powerful tool for customizing ORBs to support non-functional requirements. The approach divides the responsibility for the different parts of the customization process to the people who have the necessary knowledge to perform it. In future work we will investigate how XML can be used for expressing policies and component descriptors. Using XML will make our architecture more accessible since XML is becoming the universal language for specifying meta-data.

References

1. Joosen W., Jørgensen B.N., Linder S.M., Olsen M.M., Perram J.W., Petersen H.G., Ruhoff P.T., Sørensen A., Sørensen A.S. and Wagenaar J.M., “Towards a generic controller for arbitrary robotic manipulators”, Submitted to ICRA2000.
2. Object Management Group, “The Common Object Request Broker: Architecture and Specification”, 2.2 ed., Feb. 1998.
3. D. Box, “Essential COM”, Addison-Wesley, Reading, MA, 1997.
4. A. Wollrath, R. Riggs, and J. Waldo, “A Distributed Object Model for the Java System”, USENIX Computing Systems, vol. 9, November/December 1996.
5. Douglas C. Schmidt, David L. Levine, and Chris Cleeland, “Architectures and Patterns for High-performance, Real-time ORB Endsystems”, Advances in Computers, Academic Press, Ed., Marvin Zelkowitz, to appear in 1999.
6. Blair, G.S., Coulson, G., Robin, P., Papathomas, M., "An Architecture for Next Generation Middleware", Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98), pp 191-206, Springer, 1998.

7. Bert Robben, Bart Vanhaute, Wouter Joosen, Pierre Verbaeten, “Non-Functional Policies”, In Proceedings of the Second International Conference on Metalevel Architectures and Reflection, Saint-Malo, France, July 1999.
8. Jacobsen I., Griss M., Jonsson P., “Software Reuse; Architecture, Process and Organization for Business Success”, Addison Wesley 1997, ISBN 0-201-92476-5.
9. B. N. Jørgensen, W. Joosen, “Classifying Component Interaction in Product-line Architectures”, Proceedings of TOOLS PACIFIC 99, Melbourne, Australia, IEEE, 1999.
10. Liu C., Layland J., ”Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”, JACM, vol. 20, pp. 46-61, January 1973.
11. Frank Matthijs, “A framework for the domain of protocol stacks: methodology, conception and applications”, PhD thesis, Katholieke Universiteit Leuven, 1999.
12. Bosco, P.G., Dahle, E., Gien, M., Grace, A., Mercouroff, N., Perdigues, N., and Stefani, J.B., “The ReTINA project: an overview”, ReTINA Technical Report, 1996.
13. Hayton R., "FlexiNet Open ORB Framework", APM Technical Report 2047.01.00, APM Ltd., Poseidon House, Castle Park, Cambridge, UK, October 1997.
14. Manuel Román, Fabio Kon and Roy Campbell, ”Design and Implementation of Runtime Reflection in Communication Middleware: the dynamicTAO Case”, in proceedings of the ICDCS'99 Workshop on Middleware. Austin, Texas. May 31 - June 5, 1999
15. Svend Frølund, Jari Koistinen, “Quality-of-Service Specification in Distributed Object Systems”, in Distributed Systems Engineering Journal, volume 5, number 4, December 1998.
16. Svend Frølund, Jari Koistinen, “Quality-of-Service Aware Distributed Object Systems”, in proceedings of the 1999 USENIX Conference on Object-Oriented Technologies and Systems (COOTS).
17. 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