

# Formalizing Meta-Programming Techniques to Reconcile Heterogeneous Scheduling Disciplines in Open Distributed Real-Time Systems

Angelo Corsaro, Chris Gill, and Ron Cytron  
Department of Computer Science  
Washington University, St. Louis, MO 63130, USA  
{corsaro,cdgill,cytron}@cs.wustl.edu

Douglas C. Schmidt  
Electrical and Computer Engineering Dept.  
University of California, Irvine, CA 92697, USA  
schmidt@uci.edu

## Abstract

*In open distributed real-time and embedded (DRE) systems, different ORB endsystems may use different scheduling disciplines. To ensure appropriate end-to-end application behavior in an open architecture, DRE systems must enforce an ordering on activities originating in an endsystem and activities that migrate there, based on the relative importance of these activities. This paper describes the meta-programming techniques applied in Juno, which extends Real-time CORBA to enhance the openness of DRE systems with respect to their scheduling disciplines by enabling dynamic ordering of priority equivalence classes. We use the forthcoming OMG Real-Time CORBA 2.0: Dynamic Scheduling Joint Final Submission (RT-CORBA 2.0 JFS) to illustrate our techniques.*

**Keywords:** Real-Time and Distributed Systems, CORBA, Dynamic Scheduling, Meta-programming Architectures.

## 1. Introduction

**Emerging challenges:** Distributed object computing (DOC) middleware, such as CORBA, COM+, and Java RMI, shields developers from many complexities associated with developing distributed systems. For example, DOC middleware allows applications to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware [7]. The maturation of DOC middleware specifications and implementations over the past decade have greatly simplified the development of open distributed systems with complex *functional* requirements.

More recently, the emergence of quality of service (QoS)-enabled DOC middleware, such as Real-time CORBA 1.0 (RT-CORBA 1.0) [13], Real-Time Java [15], [2] and Distributed Real-time Java [9], will simplify open distributed real-time and embedded (DRE) systems with complex QoS requirements, such as stringent latency, jitter, and dependability. For example, future combat systems will involve heterogeneous collections of mobile

autonomous vehicles that must collaborate to perform coordinated maneuvers in support of time-critical missions, such as reconnaissance, perimeter defense, and suppression of enemy air defenses. Likewise, QoS-enabled DOC middleware will benefit commercial DRE systems, such as distributed virtual reality applications, distributed multimedia collaboration systems, and massively-multiplayer online persistent-world games.

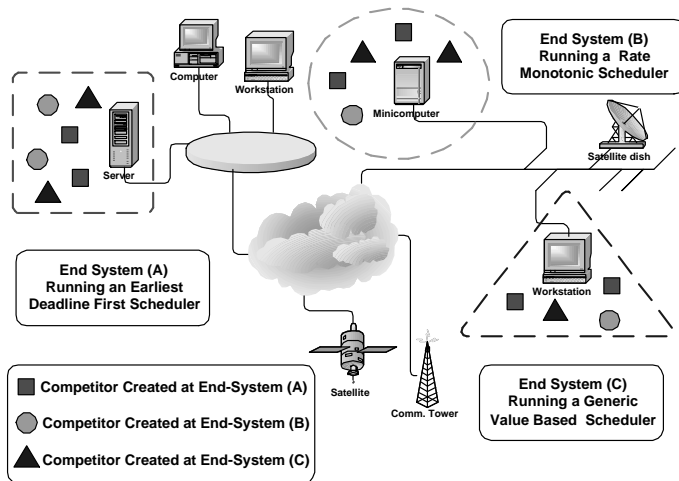
Key challenges arising in these types of DRE systems involve communicating and enforcing the relative importance of various *competitors* (such as threads or operations on CORBA objects) to ensure appropriate scheduling of system *resources* (such as memory, CPU time, and network bandwidth) at a given point in time. Resolving these challenges is essential to building DRE systems that are simultaneously:

1. *Open*, *i.e.*, system components can connect and interoperate in a flexible manner without having to be pre-configured statically; and
2. *Dependable*, *i.e.*, the system can preserve key end-to-end QoS properties, such as timeliness and resource constraints.

For example, mobile autonomous vehicles should be able to collaborate in a dependable and efficient manner, despite the heterogeneity of their scheduling disciplines and implementations. The forthcoming Real-Time CORBA 2.0: Dynamic Scheduling Joint Final Submission (RT-CORBA 2.0 JFS) [14] addresses some aspects of the challenges outlined above. For example, the RT-CORBA 2.0 JFS defines a *distributable thread* mechanism that has the following properties:

- It can extend and retract its *locus of execution*<sup>1</sup> to transition among ORBs while servicing an operation request.
- It contends with other competitors for the use of different resources (such as CPU time, memory, or network bandwidth) in the various ORBs it traverses through a dynamic call graph.
- It contains certain scheduling information carried across ORBs embedded in a GIOP service context and

<sup>1</sup>The *locus of execution* of a distributable thread represents the ORBs visited by the thread while servicing a remote method invocation.



**Figure 1. DRE Systems with Competitors that Migrate from System to System.**

used by ORBs visited by a distributable thread to ensure that the thread is processed at the appropriate priorities end-to-end.

For example, Figure 1 illustrates a representative DRE system in which three endsystems are running three ORBs configured with three different scheduling disciplines. Threads are distributed across endsystems as a result of remote operation invocations or distributable thread migration.<sup>2</sup> As a result, competitors originating on different endsystems contend for the same set of resources on each ORB endsystem. To adjudicate this competition, some type of scheduling is required.

RT-CORBA 1.0 specifies a Scheduling Service to relieve application programmers of the tedious and error-prone task of configuring scheduling properties on each end-system. This service is an optional part of the RT-CORBA 1.0 specification, however, so it may not be available for all RT-CORBA 1.0 ORBs. Moreover, the RT-CORBA 1.0 Scheduling Service deals only with priorities, which under-specify mappings of more complex scheduling properties (such as deadline) into an ordering of competitor execution eligibilities.

The RT-CORBA 2.0 JFS—and the RT-CORBA 1.0 specification upon which it builds—are the most advanced open standards that address static and dynamic scheduling in the context of open, QoS-enabled middleware for DRE systems. Neither specification, however, fully addresses the *interoperability* aspect of the challenges outlined above, due to under-specification in the areas of:

<sup>2</sup>Threads at each endsystem are shown with a different shape, depending on the endsystem on which each originated.

**1. Mapping of scheduling parameters:** The RT-CORBA 2.0 JFS does not define the mapping of scheduling parameters when distributable threads pass through ORBs that are configured with

- Heterogeneous scheduling disciplines or
- Different scheduling parameters for the same scheduling discipline.

For example, during a request's traversal through the dynamic call graph formed by a distributed thread execution, one of the visited ORBs could be configured using an *earliest deadline first* (EDF) [12] scheduling discipline. An EDF scheduler orders competitors according to the propinquity of their deadlines. Another ORB in the traversal could use a *value-based* scheduling discipline [8], where every competitor is characterized by a time-dependent function that describes the value associated with the competitor at a given point in time. A value-based scheduler tries to maximize the value gained by the system using information this function provides.

In the RT-CORBA 2.0 JFS, when a distributable thread traverses endsystems, its corresponding scheduling information must be understood at each endsystem. The composition of scheduling disciplines used along the chain of endsystems must therefore be *semantically coherent*, even if the result is non-optimal. There is no existing standard, however, that specifies *how* to provide interoperability between heterogeneous (but composable) schedulers. This omission limits the openness of DRE systems using RT-CORBA 2.0 JFS middleware.

**2. Scheduling information propagation:** Another relevant issue that neither the RT-CORBA 1.0 specification nor the RT-CORBA 2.0 JFS addresses is whether to update scheduling information propagated on a hop-by-hop basis through a distributed call graph. Although this issue is not related directly to interoperability, the solution described in this paper to enable interoperability can be used to propagate *and* update scheduling parameters end-to-end.

In this paper, we present a solution to the problems outlined above by

- Formalizing the problem of interoperability in the context of open DRE systems
- Defining formalisms to express different instances of the problem precisely and
- Providing a meta-programming architecture [20] that maps the formalized abstractions to a software architecture based on RT-CORBA.

Figure 1 outlines our solution approach in the context of CORBA. As shown in this figure, three endsystems are configured with three different scheduling disciplines. The competitors initiated at endsystem (A) are *square*, those initiated at endsystem (B) are *circular*, and those initiated at

endsystem (C) are *triangular*. To preserve the QoS properties requested by the competitors, we apply techniques that reconcile

1. The properties used by each scheduler to enforce QoS; and
2. The properties used by each competitor to express its QoS requirements.

Our techniques enable an open architecture in which competitors can traverse endsystems without concern for how QoS requirements are expressed. We also allow each ORB endsystem to schedule competitors—including those initiated remotely—by adapting the competitors’ properties for use by the ORB’s local scheduler. We use a meta-programming architecture based on a two-level reflective middleware model [20, 18, 17] to implement the solution presented in this paper.

**Paper organization:** The remainder of this paper is organized as follows: Section 2 defines a formal model for reconciling heterogeneous scheduling disciplines in open distributed real-time systems; Section 3 presents Juno, which is our meta-programming architecture for enhancing the openness of DRE middleware and illustrates briefly how Juno implements the formal model defined in Section 2; Section 4 compares our approach with related work; and Section 5 presents concluding remarks and outlines our future research directions.

## 2. Terminology and Formalisms

This section defines the terminology used throughout the paper and motivates the assumptions that underly our work. The formalisms presented in this section are applicable to any open DRE system. For concreteness, however, we focus the examples in the context of RT-CORBA 1.0 [13] and the RT-CORBA 2.0 JFS [14].

### 2.1. Properties, Competitors, and Schedulers

We model an open DRE system as consisting of *properties*, *competitors*, and *schedulers*, which are defined informally as follows:

- *Properties* describe QoS attributes, such as a criticality level, a deadline, or a constraint on jitter. We do not restrict the domain of the properties, *i.e.*, a property can be a function, which allows value and/or quality functions to be expressed as properties.
- *Competitors* denote entities that can contend for common system resources. Competitors expose properties that describe their *features*, such as their importance, or *QoS requirements*, such as deadline or worst-case execution time.

- *Schedulers* grant competitors access to shared resources. The order in which competitors can access a resource depends on scheduler disciplines and competitor properties. Scheduling disciplines are formulated in terms of the properties they use to determine the ordering of competitors. These properties can therefore be viewed as an abstraction of the competitors for the purpose of scheduling. Since we focus on dynamic systems, all our schedulers operate on-line [4], rather than off-line [16].

The remainder of this section presents a formal model for properties, competitors, and schedulers. The advantage of creating a formal model is to enable heterogeneous ORB endsystems to exchange precise information about the properties associated with individual competitors and schedulers. This information allows each endsystem to transform competitors’ properties and reconcile them for each ORB endsystem’s scheduler.

#### 2.1.1 Properties

**Definition 2.1** *Let  $\Pi$  be the Universe of Properties. A generic element of  $\Pi$  is denoted by  $\pi$  and is called a property type, or simply a property. Each property  $\pi \in \Pi$  is associated with the following tuple:*

$$\langle D_\pi, d_\pi \rangle$$

Where:

1.  $D_\pi$  is the domain of the property.
2.  $d_\pi \in D_\pi$  is the default value for the property.

That is, given any property  $\pi \in \Pi$  we denote its associated domain by  $D_\pi$ , and its associated default value by  $d_\pi$ .

Moreover, given a property  $\pi \in \Pi$  we define a Tagged Domain ( $T_\pi$ ) of  $\pi$  as:

$$T_\pi = \{\pi\} \times D_\pi = \{(\pi, u) : u \in D_\pi\}$$

Given any  $e \in T_\pi$  we denote the property of the tagged element by  $e.\pi$  and its value by  $e.value$ .

*RT-CORBA*<sup>3</sup>  $\Rightarrow$  An example of a property in RT-CORBA is the deadline of a distributable thread. In this case, the *domain* of the property is the time, which in RT-CORBA is represented as the integral type `TimeBase::TimeT`. Other examples of properties in RT-CORBA include *criticality* (which distinguishes classes of real-time competitors) and the periodicity of activities.

**Definition 2.2** *Given a set of  $n$  properties,  $n > 0$ :*

<sup>3</sup>Henceforth, our use of the term “RT-CORBA” connotes both static and dynamic scheduling capabilities.

$$P = \{\pi_1, \pi_2, \dots, \pi_n\}$$

We define the **Compound Property Domain (CPD)** as:

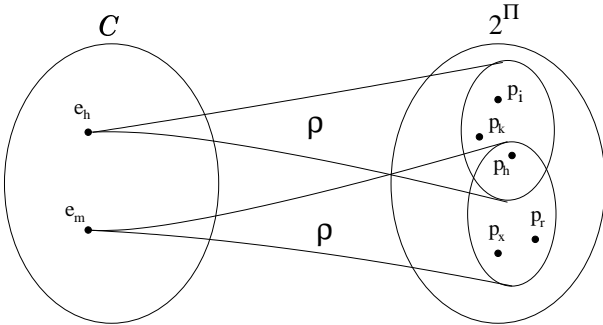
$$CPD = \{E : |E| = n \text{ and } |E \cap T_{\pi_i}| = 1, 1 \leq i \leq n\}$$

The compound property domain is a set of sets, each having size  $n$ . Each set has exactly one element from each tagged domain associated with each property in  $P$ . Note that the definition of CPD does not impose any ordering on the properties.

*RT-CORBA*  $\Rightarrow$  The *Compound Property Domain* can be viewed as a generalization of the *RT-CORBA 2.0 JFS* concept of *scheduling parameter types*. A given scheduling parameter type (e.g., the EDF scheduling parameters defined in the *RT-CORBA 2.0 JFS*) is a collection of typed properties, where a type defines a domain for the property. The *RT-CORBA 2.0 JFS* focuses on the identity of the aggregate, treating each kind of scheduling parameter as a different type. In our definition we stress the identity of single properties, so each scheduling parameter is treated as a collection of properties, rather than as a typed aggregate of properties.

### 2.1.2 Competitors

Let  $\mathcal{C}$  be the Universe of Competitors.<sup>4</sup> We assume that each competitor exposes a set of properties, as shown in Figure 2.



**Figure 2. Association Between Competitors and Properties**

**Definition 2.3** We define the following function:

$$\rho : \mathcal{C} \mapsto 2^\Pi$$

that when given a competitor  $c \in \mathcal{C}$ , maps it to the set  $\rho(c)$  of properties it exposes.

<sup>4</sup>In our case, the universe of discourse is those entities that can compete for the use of resources, and are thus subject to scheduling.

At any point in time, any competitor  $c$  has associated with it the current value of its properties. This value is actually an element of the Compound Property Domain of  $\rho(c)$ , and will be indicated with  $c.pval$ .

Figure 2 shows schematically how the relation  $\rho$  works. In this figure,  $\mathcal{C}$  represents the Universe of Competitors,  $2^\Pi$  is the power-set of the Universe Of Properties,  $e_h$  and  $e_m$  represent generic competitors, and  $\rho(e_h)$  and  $\rho(e_m)$  represent the property sets (contained in  $2^\Pi$ ), respectively.

*RT-CORBA*  $\Rightarrow$  Competitors in *RT-CORBA* can be

- Distributable threads that compete for CPU time on ORB endsystems
- Events in an event channel [6] that must be delivered to consumers that have subscribed for particular events or
- GIOP requests that compete for network/bus resources.

If competitor  $c$  is a distributable thread in the context of *RT-CORBA*, then  $\rho(c)$  can be the set of properties containing the elements *deadline*, *importance*, and *laxity*. In this case, the  $c.pval$  would be the value of the deadline, importance, and laxity at a particular point of time.

### 2.1.3 Schedulers

**Definition 2.4** We define an **Ordering of Classes of Equivalence (OCE)** over the set of properties  $P \subseteq 2^\Pi$  as consisting of the following tuple:

$$\langle =_{OCE}, <_{OCE} \rangle$$

where:

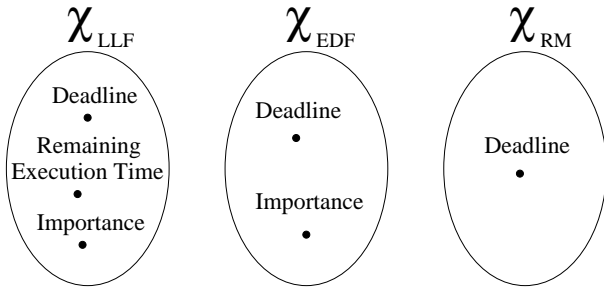
1.  $=_{OCE}$  is an equivalence relation over the CPD of  $P$ .
2.  $<_{OCE}$  is a total ordering over the set  $\{[a] : a \in CPD\}$

$[a]$  represents the equivalence class to which the element  $a$  belongs. Based on this definition, the OCE provides a partition of equivalence classes over CPD and also provides a total order of equivalence classes.

Note that the ordering of equivalence classes is defined over a set of properties. Property ordering therefore has no effect on the structure of the equivalence classes, nor on equivalence class ordering. The ordering of equivalence classes depends only on the *value* and *type* of properties. Conversely, due to run-time changes in system configuration or scheduler operation mode, the ordering of equivalence classes can depend on time. The time dependency of equivalence classes—and of their ordering—can also occur when schedulers refer to time-dependent properties, such as value functions.

**Definition 2.5** A **Scheduler** is an *Ordering of Classes of Equivalence (OCE)* over a set of properties. The set of properties on which a scheduler imposes an OCE is called its *Characteristic Set*, which expresses the properties used by a scheduler to impose an ordering on competitors. Of properties exposed by a competitor, a scheduler only considers those in its characteristic set. Given a scheduler  $S$ , we indicate its characteristic set with  $\chi_S$ .

RT-CORBA  $\Rightarrow$  Figure 3 shows the characteristic sets the RT-CORBA 2.0 JFS defines for the least laxity first (LLF)<sup>5</sup>, EDF, and rate monotonic (RM) scheduling disciplines. If



**Figure 3. Characteristic Set for Least Laxity First (LLF), Earliest Deadline First (EDF), and Rate Monotonic (RM) Scheduling Disciplines**

we consider the RT-CORBA 2.0 JFS EDF scheduler, the properties in the scheduler’s characteristic set are the *deadline* and the *importance*.<sup>6</sup> The equivalence classes in this case are therefore represented by the set containing these two properties. The equivalence classes are ordered so that the importance and deadline ( $i, d$ ) associated with each equivalence set are ordered. An example of such an ordering could be the following expression:

$$(i_1, d_1) < (i_2, d_2) \text{ iff } (i_1 < i_2) \text{ or } (i_1 = i_2, d_1 > d_2)$$

In this example, the ordering of the importance and deadline are both the ordering of integral values. RT-CORBA 2.0 JFS defines the importance as a `long` type, and deadline as a `TimeBase :: TimeT` type.

Based on the definitions presented above, we can treat any scheduler as an ordering of equivalence classes over a set of properties used by a scheduler. These properties are associated with a competitor by the relation  $\rho$ . Note that the scheduler partitions the full *Compound Property Domain* of its characteristics into a series of equivalence classes and

<sup>5</sup>An LLF scheduler determines the execution eligibility based on *laxity*, which is defined as the difference between the deadline, the current time, and the estimated remaining computation time.

<sup>6</sup>In the canonical EDF definition [12] there is no concept of “importance” but in the RT-CORBA 2.0 JFS there is.

then orders these classes. Also note that the property values associated with competitors can change over time; a potential effect of this change is to move a competitor from one equivalence class to another.

Finally, we assume that all schedulers in DRE systems are *well-behaved*, which means that schedulers on different ORB endsystems try to enforce real-time QoS over the properties used to characterize the competitors. Specifically, we do not consider pathological cases where schedulers do not work to improve QoS in at least some dimension. For example, a rate monotonic scheduler (RMS) [12] and an EDF scheduler will use different orderings of operations, but they will both work to *improve* the deadline feasibility of operations they schedule.

## 2.2. Adapters

### 2.2.1 Core Adapter Concepts

Having formally defined the terms property, competitor, and scheduler, we can now address problems arising when establishing an ordering of competitors with sets of properties that differ from the *Characteristic Set* of a scheduler. Below, we address the different cases that can arise.

**Definition 2.6** Given two set of properties:

$$P_1, P_2 \subseteq 2^{\Pi}$$

then an **Adapter** from  $P_1$  to  $P_2$  is a function of the type:

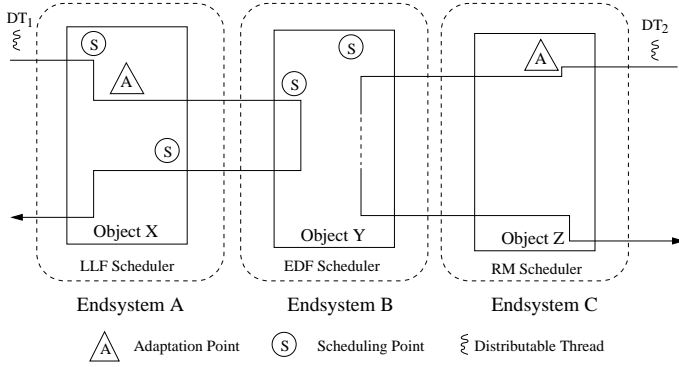
$$A_{P_1 \mapsto P_2} : CPD_{P_1} \mapsto CPD_{P_2}$$

Thus, an *Adapter* is defined as a function that transforms one set of properties into another. The definition given above is quite general, *i.e.*, no assumption are made about the mapping performed by an *Adapter*. In practice, some *Adapters* make more sense than others.

RT-CORBA  $\Rightarrow$  Figure 4 depicts a scenario in which three endsystems are each running an ORB with a different scheduling discipline. Two distributable threads,  $DT_1$  and  $DT_2$  are moving across endsystems.  $DT_1$  originated at endsystem A, where it executed an operation on the object X. It migrates from endsystem A to endsystem B after invoking an operation on object Y. In contrast,  $DT_2$  originated at endsystem B, where it executed an operation on object Z. It migrates from endsystem A to endsystem B after invoking an operation on object Y.

Three different schedulers are used by the ORBs in Figure 4, (endsystem C has a static RM scheduler). As shown in Figure 3 these schedulers have different *Characteristic Sets*. As a result, some adaptation will be required when a distributable thread crosses a *scheduling domain*.<sup>7</sup> The

<sup>7</sup>A scheduling domain is a collection of ORB endsystems using the same scheduling algorithm and properties.



**Figure 4. A Distributable Thread traversing endsystems that have different Scheduling Disciplines**

claim of this paper is that the proper type of *Adapter* can handle this adaptation. In addition, Figure 4 shows the point at which schedulers are executed, and the place at which distributable thread property adaptation can occur, *i.e.*, the place at which the right adapter is executed.

Figure 4 also shows  $DT_2$  is preempted by  $DT_1$  while executing on the endsystem B's ORB. This case shows that the dynamic priority of  $DT_1$  must be higher than that of  $DT_2$ . In general,  $DT_1$  and  $DT_2$  would be non-comparable unless adaptation is performed to make sure that their properties can be expressed in a manner comprehensible by endsystem B's ORB scheduler. Such adaptation and reconciliation of the distributable threads (*i.e.*, competitor) properties can be achieved by means of *Adapters*.

### 2.2.2 Reconciling Properties Through Adapters

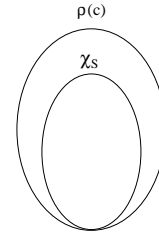
Now that we have defined our terminology and formal model, we show how these formalisms can be used to reconcile properties to support interoperability between heterogeneous RT-CORBA schedulers. Below, we examine three relevant cases that can occur and outline a solution for each of them.

**Case 1:** Figure 5 shows a scheduler  $S$  with a non-empty *Characteristic Set*  $\chi_S$  and given competitor  $c$  with  $\rho(c) \supset \chi_S$ . To map the properties exposed by the *Competitor* into the Ordering of Classes of Equivalence created by the scheduler over  $\chi_S$  we can apply the following *Restriction Adapter*:

$$RA : CPD_{\rho(c)} \mapsto CPD_{\chi_S}$$

defined as:

$$\forall a \in CPD_{\rho(c)}, RA(a) = \{e \in a : e.\pi \in \chi_S\}$$

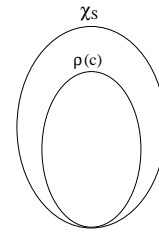


**Figure 5. The Properties used by a Scheduler are a Subset of Properties Exposed by a Competitor**

A *Restriction Adapter* drops the properties exposed by a competitor that do not belong to the scheduler's *Characteristic Set*.

*RT-CORBA*  $\Rightarrow$  For example, if we consider the case shown in Figure 4, a *Restriction Adapter* could be applied to  $DT_1$  immediately before leaving its ORB or when arriving at endsystem B's ORB. What the *Restriction Adapter* does in this case is map the property exposed by  $DT_1$  from a set containing *deadline*, *importance*, and *remaining execution time*, to the set containing just *deadline* and *importance*. Moreover, a *Restriction Adapter* implementation should also express the property being adapted in a form that can be manipulated efficiently by the scheduler. This form is generally scheduler-dependent because properties are exposed uniformly by competitors, *e.g.*, a distributable thread in this context.

**Case 2:** Figure 6 shows a scheduler  $S$  with a *Characteristic Set*  $\chi_S$  and given competitor  $c$  with a non-empty  $\rho(c) \subset \chi_S$ . To map the properties exposed by the *Competi-*



**Figure 6. The Properties Exposed by the Competitor are a Subset of Properties used by the Scheduler**

*tor* into the ordering of equivalence classes created by the scheduler over  $\chi_S$  we can use the following *Default Extension Adapter*:

$$DEA : CPD_{\rho(c)} \mapsto CPD_{\chi_S}$$

which is defined as:

$$\forall a \in CPD_{\rho(c)}, \\ DEA(a) = \{a\} \cup \{e.\pi.d_\pi : e \in \chi_S - \rho(c)\}$$

A variation of the *Default Extension Adapter* is one that considers specific values for extending the given set of properties. Using this, we can then define the *Extension Adapter* as the tuple:

$$\langle E, EA \rangle$$

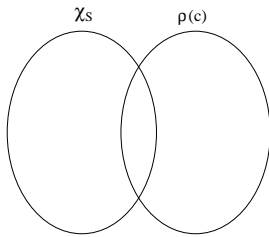
where:

1.  $E \in (CPD_{\chi_S} - CPD_{\rho(c)})$  is the set of default values.
2.  $EA : CPD_{\rho(c)} \mapsto CPD_{\chi_S}$   
 $\forall a \in CPD_{\rho(c)}, EA(a) = \{a\} \cup E$

An *Extension Adapter* can be used to extend the set of properties exposed by a competitor, so they are at least the same as those present in the scheduler's *Characteristic Set*.

*RT-CORBA*  $\Rightarrow$  Again using the example in Figure 4, a *Default Extension Adapter* or *Extension Adapter* could be applied to  $DT_2$  right before leaving its ORB or upon its arrival on endsystem B's ORB. The *Extension Adapter* would map the property exposed by  $DT_2$  from a set containing only the *deadline* to the set containing *deadline* and *importance*. As with the earlier cases, an adapter can express the property being adapted into a form that can be manipulated efficiently by a scheduler. Moreover, an *Adapter* can enable a statically scheduled ORB to interoperate with a dynamically scheduled ORB.

**Case 3:** In general, given a scheduler  $S$  with a non-empty *Characteristic Set*  $\chi_S$  and given competitor  $c$  with  $\rho(c)$ , there could be no particular relation between the two set of properties  $\chi_S$  and  $\rho(c)$ , as shown in Figure 7. In this



**Figure 7. No Assumption about the Properties used by the Scheduler and the Properties Exposed by the Competitor**

case a *Generalized Adapter* should be used. Unlike Case 2, however, this type of *Adapter* does not guarantee the value of the properties shared by the two sets  $\rho(c)$  and  $\chi_S$  will remain unchanged. A *Generalized Adapter* is defined as a transformation of the type:

$$GA : CPD_{\rho(c)} \mapsto CPD_{\chi_S}$$

A *Generalized Adapter* contains the adapters described thus far as a special case. We introduce the concept of a *Generalized Adapter* to define custom adaptation between property sets, thereby allowing extra flexibility and control over how adaptation occurs. While *Extension* and *Restriction Adapters* can be created dynamically by a *Meta\_Adapter* (as described in Section 3.1), *Generalized Adapters* must be provided by users or applications.

*RT-CORBA*  $\Rightarrow$  For example, consider a case in which a distributable thread transitions from

- An ORB configured with a MUF scheduler that uses the importance property to isolate different classes of competitors (e.g., statistical real-time vs. deterministic real-time) to
- An ORB with an EDF scheduler that does not consider the importance property.

In this case, if we simply use a *Restriction Adapter* we will lose information contained in the importance property associated with the distributable thread. One way to handle competitors having the same deadline—but different relative importance—is to boost the deadline of the more important competitor via an *ad hoc* transformation, which could be performed via a *Generalized Adapter*.

As shown in the three cases examined above, *Adapters* provide a way to transform and reconcile the properties of competitors to properties used by a scheduler. For most cases that occur in practice, an *Adapter* that perform the right transformation can be generated at run-time by the system or provided by the users or applications.

### 3. Juno: A Meta-Programming Architecture for Heterogeneous Middleware Interoperability

This section shows how the formalisms described in Section 2 can be used as the conceptual foundation for building DRE middleware that supports interoperability between heterogeneous scheduling disciplines. We outline the requirements imposed on a DRE middleware meta-programming framework called *Juno*, which implements the formalisms described in Section 2. We then briefly show how this architecture can be implemented in an ORB.

#### 3.1. Overview of the Juno Meta-Programming Architecture

To assure scheduler interoperability, a DRE system that implements the formalisms introduced in Section 2 must determine how to map properties, competitors, the function  $\rho(c)$ , schedulers, each scheduler's characteristic set  $\chi_S$ , and the necessary *Adapters* onto a meta-programming software

architecture. Juno’s architecture has been designed based on the observation that the function  $\rho(c)$  and the characteristic set  $\chi_S$  can be treated as operators that “reflect” the properties exposed by competitors and schedulers.

The degree of control and introspection needed to implement the formalisms introduced in Section 2 can be obtained via the Juno meta-programming architecture shown in Figure 8. As shown in this figure, competitors and

**Adapter:** This class provides an abstract interface for all *Adapter* implementations. As defined in Section 2, an *Adapter* converts one set of properties to another. In the context of RT-CORBA, an *Adapter* object can, for example, convert the properties needed by an earliest deadline first (EDF) scheduler into those needed by a maximize accrued utility (MAU)<sup>8</sup> scheduler.

**Meta\_Property:** This meta-class associates a *Property* base-object with a *Property* type. As discussed above, the *Property* base-object represents an element of the *Property Domain* defined in Section 2. This value must be associated with a property type. As shown in Section 2, this association was achieved by using the *Tagged Domain* of a *Property*. In Juno, we achieve this association by tying each property base object to its meta-object *Meta\_Property*.

A *Meta\_Property* provides access to the default value of a property, and a factory method creates its associated property. This meta-class reconciles property representations that might differ from ORB endsystem to endsystem. For example, a property that represents time could be expressed using different time scales on different systems.

**Meta\_Competitor:** This meta-class manages the transformations required whenever a competitor has properties that do not match a scheduler’s characteristic set directly. Juno encapsulates the logic that performs property reconciliation in the meta-class *Meta\_Competitor*. This meta-class shields developers from the complexities of interoperability.

A *Meta\_Competitor* selects the *Adapter* that performs the most suitable transformation, depending on the following factors:

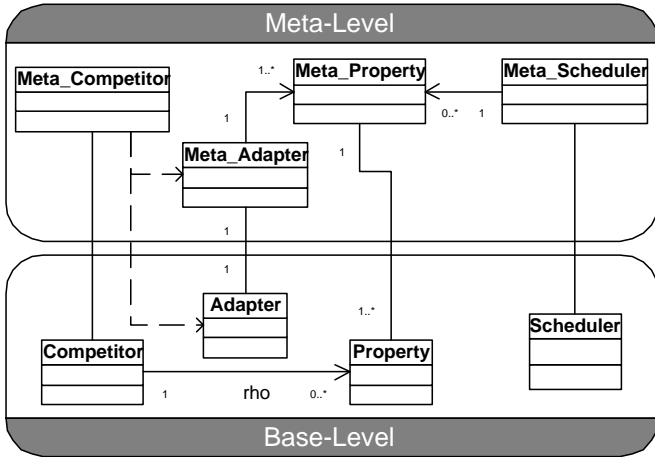
- The properties that are associated with its competitor; and
- The properties that characterize the endsystem scheduler, which are accessed via the *Meta\_Scheduler* defined below.

Juno provides an explicit meta-object protocol that enables base-objects to configure the way in which property adaptation can occur, and to restrict the types of property adaptation.

**Meta\_Scheduler:** This meta-class provides an interface that enables introspection of the properties used by its associated base-object *i.e.*, a *Scheduler*. It implements an interface to the characteristic set of a scheduler by providing an explicit MOP to introspect the characteristic set.

**Meta\_Adapter:** This meta-class provides a way to introspect the signature associated with its base-object, *i.e.*, an

<sup>8</sup>A maximize accrued utility (MAU) scheduler, associates each competitor with a value function and the scheduler tries to maximize the value of this function.



**Figure 8. UML Class Diagram for Juno’s Meta-Programming Architecture**

properties are first class entities, along with adapters and schedulers. Moreover, the function  $\rho(c)$  is represented by the association between competitors and properties, which are treated as first class entities. The characteristic set of a scheduler  $\chi$  is exploited by the association between the scheduler meta-object and the property meta-object. The roles of components in Figure 8 are summarized below:

**Property:** This class provides an abstraction for the representation of a value of the property domain, as defined in Section 2. The combination of the *Property* and its meta-object provides the same information as an element of the *Tagged Domain*. For example, in the context of RT-CORBA the deadline and the period might map to the same *Property* class, but their meta-objects would contain the information needed to distinguish the two properties.

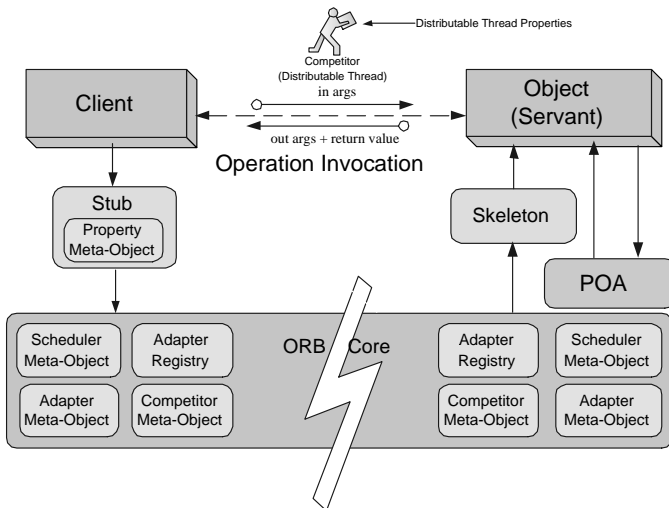
**Competitor:** This class provides an abstraction for entities that can be scheduled. For example, in the context of the RT-CORBA 2.0 JFS a distributable thread can be implemented as a specialization of this class.

**Scheduler:** This class represents the abstraction for an ORB endsystem scheduler. It provides an interface for adding and removing competitors, and for testing their feasibility. In the context of RT-CORBA, a concrete implementation of this class could be an EDF scheduler or an LLF scheduler.

Adapter. By “signature” we mean the two sets of properties that represent the domain and the co-domain for an Adapter. A `Meta_Adapter` also provides a factory method to create an `Adapter` that matches a given signature. Describing an `Adapter` in terms of the adaptation of properties it performs is essential to enable the activities of a `Meta_Competitor`.

### 3.2. Applying Juno to RT-CORBA

Juno’s meta-programming architecture described in Section 3.1 can be used as a reference model to realize interoperable RT-CORBA ORBs. Some of the meta-objects present in the meta-layer outlined in Figure 8 can be directly embedded inside the ORB Core, whereas other meta-objects can be associated with stubs and skeletons. As shown in Figure 9, a CORBA ORB can incorporate certain meta-objects present in the meta-layer outlined in Figure 8 directly inside the ORB Core, whereas other meta-objects can be associated with stubs. Each time a distributable



**Figure 9. An Open and Interoperable RT-CORBA Implementation**

thread transitions from one ORB to another, the properties it exposes must be reconciled with the *Characteristic Set* of the foreign ORB endsystem’s scheduler, as described in Section 2.2.

In the context of an ORB, it is necessary to determine where and when property reconciliation occurs. As Figure 4 suggests, property reconciliation must occur before a scheduler can order competitors. It must therefore occur either right before the distributable thread leaves its home ORB (client) or right after the arrival at the foreign ORB (server).

## 4. Related Work

Our research on meta-programming mechanisms has been influenced by the following projects.

**Emerging middleware standards:** Distributed real-time and embedded (DRE) systems are increasingly being implemented via standard middleware. CORBA is one of most widely used middleware platforms for DRE systems. To enable the use of CORBA as middleware for building DRE systems the Object Management Group has specified RT-CORBA 1.0 [13] and the RT-CORBA 2.0 JFS [14].

**Meta-programming techniques and reflective middleware:** Meta-programming techniques have been a focus of research for many years. For example, the Common Lisp Object System (CLOS) is an early example of a sophisticated meta-object protocol (MOP) [10]. Meta-programming techniques were used initially in artificial intelligence research, but are now being applied in systems software research, where they are used to make ORB middleware more dynamically configurable, adaptive, and reflective.

An example of this cross-fertilization is dynamic-TAO [11] from the University of Illinois, Urbana Champaign, which illustrates that TAO can be reconfigured at runtime by dynamically linking/unlinking certain components. A related effort at Washington University and UCI [19] is exploring the application of reflective middleware techniques in the context of the CORBA Component Model [1]. Yet another example is the Adapt Project [5, 3] at Lancaster University, which is applying a multi-level reflective middleware model focused on dynamic composition of objects.

## 5. Concluding Remarks

This paper presents a model that formally characterizes properties, competitors, and schedulers in open distributed real-time and embedded (DRE) systems. A key idea that emerges from this formal model is that properties “belong” to competitors. Moreover, a competitor can expose more or fewer properties than a scheduler strictly needs to order access to resources. The process of making a property a first-class entity is fundamental to achieve interoperability among heterogeneous ORB endsystem schedulers.

This paper also outlines how our formal model of properties, competitors, and schedulers is being reified in *Juno*. *Juno* applies *meta-programming techniques* to improve scheduler interoperability in heterogeneous ORB endsystems. Meta-programming techniques are becoming a popular way to enable DRE systems that are adaptable, flexible, configurable, predictable [19] and composable [17].

Our future research on *Juno* focuses on the following two topics:

**1. Theoretical analysis:** We are investigating the theoretical aspects involved in transforming and adapting the properties of competitors. Understanding the effect of a property transformation on a competitor's importance—and how we can relate the equivalence classes created by different scheduling algorithms—is important to detect “invalid” transformations, *i.e.*, transformations that disregard properties fundamental to expressing the key QoS requirements of competitors. The advantage of expressing these theoretical aspects in formal model is that it simplifies the communication between systems and the transformations performed on properties of competitors.

Another theme in our theoretical investigation is how adaptations affect the fulfillment of end-to-end application QoS requests. Our focus is on schedulability analysis in end-to-end DRE systems where each endsystem can potentially have a different scheduling algorithm that requires adaptation of QoS requirements. This investigation will provide us with criteria to determine the schedulability of a given set of competitors automatically in an open DRE environment.

**2. Empirical evaluation:** We are extending TAO to support the Juno meta-programming architecture described in Section 3. These enhancements are part of broader efforts to apply reflective middleware techniques [19] and dynamic scheduling [4] to TAO. In these efforts we are developing a testbed to conduct empirical benchmarks that will quantify the QoS provided by Juno. Our goals are to

1. Identify the critical software patterns and framework components and
2. Measure the impact of our solution on end-to-end DRE performance, predictability, and flexibility.

This dimension of our research will demonstrate how to develop open DRE systems that implement the flexible Juno formalisms and meta-programming architecture presented in this paper without undue loss of QoS.

All the source code, documentation, and test cases for the TAO open-source CORBA ORB can be downloaded from [www.cs.wustl.edu/~schmidt/TAO.html](http://www.cs.wustl.edu/~schmidt/TAO.html).

## References

- [1] BEA Systems, *et al.* *CORBA Component Model Joint Revised Submission*. Object Management Group, OMG Document orbos/99-07-01 edition, July 1999.
- [2] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
- [3] Fábio M. Costa and Gordon S. Blair. A Reflective Architecture for Middleware: Design and Implementation. In *ECOOP'99, Workshop for PhD Students in Object Oriented Systems*, June 1999.
- [4] C. D. Gill, D. L. Levine, and D. C. Schmidt. The Design and Performance of a Real-Time CORBA Scheduling Service. *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 20(2), March 2001.
- [5] Gordon S. Blair and G. Coulson and P. Robin and M. Papatomas. An Architecture for Next Generation Middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 191–206, London, 1998. Springer-Verlag.
- [6] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–199, Atlanta, GA, October 1997. ACM.
- [7] M. Henning and S. Vinoski. *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1999.
- [8] E. D. Jensen. Eliminating the Hard/Soft Real-Time Dichotomy. *Embedded Systems Programming*, 7(10), Oct. 1994.
- [9] E. D. Jensen. Distributed Real-Time Specification for Java. [java.sun.com/aboutJava/communityprocess/jsr/jsr\\_050\\_drt.html](http://java.sun.com/aboutJava/communityprocess/jsr/jsr_050_drt.html), 2000.
- [10] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of The Metaobject Protocol*. The MIT Press, Cambridge, Massachusetts, 1991.
- [11] F. Kon and R. H. Campbell. Supporting Automatic Configuration of Component-Based Distributed Systems. In *Proceedings of the 5<sup>th</sup> Conference on Object-Oriented Technologies and Systems*, pages 175–178, San Diego, CA, May 1999. USENIX.
- [12] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM*, 20(1):46–61, January 1973.
- [13] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.4 edition, Oct. 2000.
- [14] Object Management Group. *Dynamic Scheduling Real-Time CORBA 2.0 Joint Final Submission*, OMG Document orbos/2001-04-01 edition, April 2001.
- [15] Real-time Java Experts Group. *Real-time Java Specification*.
- [16] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, Apr. 1998.
- [17] N. V. Sebastian Gutierrez-Nolasco. A Composable Reflective Communication Framework. In *Proceedings of IFIP/ACM Workshop on Reflective Middleware 2000*, April 2000.
- [18] N. Venkatasubramanian. *An Adaptive Resource Management Architecture for Global Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, 1998.
- [19] N. Wang, D. C. Schmidt, M. Kircher, and K. Parameswaran. Adaptive and Reflective Middleware for QoS-Enabled CCM Applications. *IEEE Distributed Systems Online*, 2(5), July 2001.
- [20] C. Zimmermann, editor. *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, Boca Raton, FL, 1996.