Report Number: WUCS-01-24

2001-08-01

Juno: A Framework for Reconciling Scheduling Disciplines

Angelo Corsaro

Scheduling problems arise each time there is some form of resource contention. The problem addressed by scheduling disciplines is that of ordering the access to contended resources. The ordering is typically based on either (1) properties that are exposed by the entities that compete for the resource (like a deadline), or by (2) external properties (like the arrival order), or (3) a combination of both. In literature there exist many different scheduling algorithms, each of which has certain properties and an associated application domain. All these scheduling disciplines are based on the assumption that all the entities that com-pete... Read complete abstract on page 2.

Juno: A Framework for Reconciling Scheduling Disciplines

Angelo Corsaro

Complete Abstract:

Scheduling problems arise each time there is some form of resource contention. The problem addressed by scheduling disciplines is that of ordering the access to contended resources. The ordering is typically based on either (1) properties that are exposed by the entities that compete for the resource (like a deadline), or by (2) external properties (like the arrival order), or (3) a combination of both. In literature there exist many different scheduling algorithms, each of which has certain properties and an associated application domain. All these scheduling disciplines are based on the assumption that all the entities that com-pete for a resource are provided with the same collection of properties. This assumption makes sense in a closed environment; however it makes interoperability difficult for sys-tems that have different scheduling algorithms and in which competitors migrate from one system to another. This problem is becoming evident in distributed computing environ-ments like Object Request Brokers (ORB), Agent Frameworks, Load Balancing Systems, in which active components, which have usually QoS requirements, migrate through differ-ent endsystems. In such scenarios we cannot assume that all the endsystems will provide the same scheduling disciplines for all the resources that might be subject to scheduling. Even if they do, there might be a lack of a global knowledge that would make interoperability hard. In general, it is desirable that the QoS requirements exposed by any of these active components will be preserved and enforced even in face of presence of non-homogeneity, and migration. This thesis tackles the problem outlined above, by (1) providing a formal framework that can be used to describe any scheduling discipline, (2) providing a set of transformations that can be applied to the migrating entities, to reconcile their QoS requirements with respect to the scheduling discipline of the visited end system, and (3) providing a meta-programming architecture that maps the formalized abstractions to a software architecture that can be used as a reference model for a system that implements the ideas expressed in this thesis.

2001-24

Juno: A Framework for Reconciling Scheduling Disciplines

Authors: Corsaro, A.

Abstract: A thesis presented to the Sever Institute of Washington University for the degree Master of Science August 2001

Type of Report: Other

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

JUNO: A FRAMEWORK FOR RECONCILING SCHEDULING DISCIPLINES

by Angelo Corsaro

ADVISOR: Ron K. Cytron

August, 2001
Saint Louis, Missouri

Scheduling problems arise each time there is some form of resource contention. The problem addressed by scheduling disciplines is that of ordering the access to contended resources. The ordering is typically based on either (1) properties that are exposed by the entities that compete for the resource (like a deadline), or by (2) external properties (like the arrival order), or (3) a combination of both. In literature there exist many different scheduling algorithms, each of which has certain properties and an associated application domain. All these scheduling disciplines are based on the assumption that all the entities that compete for a resource are provided with the same collection of properties. This assumption makes sense in a closed environment; however it makes interoperability difficult for systems that have different scheduling algorithms and in which competitors migrate from one system to another. This problem is becoming evident in distributed computing environments like Object Request Brokers (ORB), Agent Frameworks, Load Balancing Systems,

in which active components, which have usually QoS requirements, migrate through different endsystems. In such scenarios we cannot assume that all the endsystems will provide the same scheduling disciplines for all the resources that might be subject to scheduling. Even if they do, there might be a lack of a global knowledge that would make interoperability hard. In general, it is desirable that the QoS requirements exposed by any of these active components will be preserved and enforced even in face of presence of non-homogeneity, and migration.

This thesis tackles the problem outlined above, by (1) providing a formal framework that can be used to describe any scheduling discipline, (2) providing a set of transformations that can be applied to the migrating entities, to reconcile their QoS requirements with respect to the scheduling discipline of the visited endsystem, and (3) providing a meta-programming architecture that maps the formalized abstractions to a software architecture that can be used as a reference model for a system that implements the ideas expressed in this thesis.

Con Immenso Affetto, a mio Padre Angelo, mia Madre Lina, mio Fratello Alessandro, e la mia dolce compagna Huifen.

Contents

List of Figures

Acknowledgments

In the year and half I've spent at the Washington University, I have had a chance to work, and/or engage in stimulating discussion with many different people. So there are many people that I should thank, and I might forget somebody. I would like to thank Dr. David L. Levine, for giving me the ability of embarking in this adventure in "The New World"; Dr. Doug C. Schmidt for his amazing and effective presence and responsiveness; my advisor Dr. Ron K. Cytron for his support, and time spent brainstorming; Chris D. Gill for the visionary discussion; David Sharp for helping me out in casting abstractions down to real problem; Nanbor Wang (aka Don Vincenzo) for being a great God Father, and for being able to stand me as office mate for one and half year, being always ready to hear about my existential problem and complains; Irfan Pyarali, for the interesting discussion about POAs and concurrency, Krishnakumar Balasubramanian (aka Kitty) the Linux Wiz; Frank Hunleth, my dear XP pair; Yamuna Krishnamurthy, Pradeep Gore, Kirthika Parameswaran (aka Kirthicuccia), and Vishal Kachroo (aka Er Metallico) for all the nights spent in the lab together playing music and programming; Jeff Parsons for the talk about music, and musicians; Balachandran Natarajan for teaching me how to eat the Indian way; Luther Baker (aka Lutero) for his love for Italy, and his help in getting settled down when I first got to the States; Joe Hoffert for the funny talk about Italy, Italian food, and Italian slang.

Finally, I thank DARPA for supporting my research under contract N66001–97–C–8519.

Angelo Corsaro

Washington University in Saint Louis
August 2001

Chapter 1

Introduction

The tremendous widespread of networking infrastructures, in conjunction with the decreasing cost of computing devices, has pushed the deployment of Distributed Software Systems (DSSs) to scenarios that were unthinkable before. Most of the newer application domains are characterized by the fact that:

1. They require the DSS to be Quality of Service (QoS)-aware,
2. They contain active components that migrate through the system's endpoints, and
3. Endpoints cannot usually be assumed to be homogeneous computing platforms:[1] platforms that appear to be homogeneous may have heterogeneous configurations; for example, the platforms may have different scheduling disciplines.

The above scenario implies that these DSSs need to guarantee the enforcement of certain deterministic and/or statistical properties and behaviors in the face of migration and heterogeneity. This imposes new requirements on the middleware used to develop DSSs. QoS-Enabled Middleware currently lacks (1) the ability to cope with heterogeneous scheduling disciplines, and (2) the ability to reason about the ordering of active components that expose different properties. The following sections provide a statement of this problem in the context of the Common Object Request Broker Architecture (CORBA), which enjoys widespread use, particularly among the research and academic communities.

[1]The term *computing platform* includes both the Operating System (OS) and Middleware on which the DSS is developed/deployed.

1.1 Emerging challenges

Distributed Object Computing (DOC) middleware, such as CORBA [25], Component Object Model (COM)+ [23], and Java Remote Method Invocation (RMI) [40], 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 [11]. The maturation of DOC middleware specifications and implementations over the past decade has 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 (RT)–CORBA 1.0 [25], RT–Java [28], [2] and Distributed Real-Time Java (DRTJ) [14], has simplified 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 preconfigured 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 [37] 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) [26] addresses some aspects of the challenges outlined

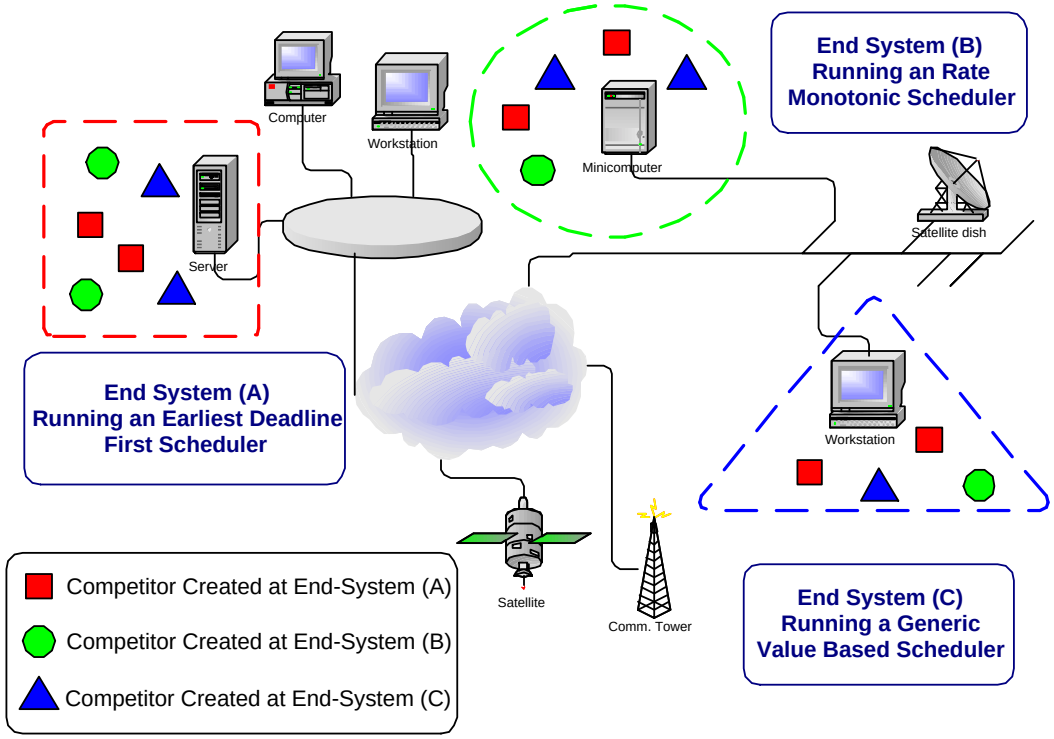

Figure 1.1: DRE Systems with Competitors that Migrate from System to System.

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*[2] 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. The dynamic call graph is a directed graph whose nodes are the endsystems visited by the distributable thread, and whose edges represent the direction in which the distributable threads migrates across endsystems.
- It contains certain scheduling information carried across ORBs embedded in a General Inter-ORB Protocol (GIOP)[3] service context and used by ORBs visited by a

[2]The *locus of execution* of a distributable thread represents the Object Request Brokers (ORBs) visited by the thread while servicing a remote method invocation.

[3]The GIOPis the communication protocol used between ORBs.

distributable thread to ensure that the thread is processed at the appropriate priorities end-to-end.

For example, Figure 1.1 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.[4] 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 endsystem. 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:

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) [19] 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 [13], where every competitor is characterized by a time-dependent function that describes the value associated with

[4] Threads at each endsystem are shown with a different shape, depending on the endsystem on which each originated.

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 thesis to enable interoperability can be used to propagate *and* update scheduling parameters end-to-end.

   The problem outlined above in the context of CORBA holds true for the other middleware platform that could be used to develop DSSs. Even the upcoming Real-Time Java specification does not take into account how the problem of enabling QoS can be applied in presence of heterogeneous scheduling disciplines.

1.2 What do we need?

At this point, we have defined the problem addressed in this thesis—achieving interoperability of scheduling disciplines. To obtain the greatest leverage, a solution to the above problem should be deployed in middleware, where it can be shared by applications and end-systems alike. In line with this idea, this thesis proposes a solution to the problem outlined, by

- Providing a formal framework that can be used to describe any scheduling discipline;
- Providing a set of transformations that can be applied to the migrating entities, to reconcile their QoS requirements with respect to the scheduling disciplines of the visited endsystems;

- Providing a meta-programming architecture [17, 35, 43] that maps the formalized abstractions to a software architecture that can be used as a reference model for a system that implements the ideas expressed in this thesis;

Moreover, this thesis provides documents experience in extending an ORB to support our ideas, and the design of a Real-Time Event Channel that implements the framework described in this thesis.
Figure 1.1 outlines our approach in the context of CORBA. Three endsystems are shown, each configured with a different scheduling discipline. 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 requirement are expressed. We also allow each endsystem to schedule competitors—including those initiated remotely—by adapting the competitors' properties for use by an ORB's local scheduler.

1.3 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 provides an overview of the related work; Chapter 3 defines a formal model for reconciling heterogeneous scheduling disciplines in open distributed real-time systems; Chapter 4 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 Chapter 3; Chapter 5 and Chapter 6 present two case studies that show how the concepts introduced by this thesis can be applied to solve real-world problems; and Chapter 7 presents concluding remarks and outlines possible future research directions.

Chapter 2

Related Work

While the problem addressed in this thesis has not yet been investigated, this chapter surveys extant literature dealing with related problems that has motivated our work.

2.1 Emerging middleware standards

DRE systems are increasingly 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 (OMG) has specified RT–CORBA 1.0 [25] and the RT-CORBA 2.0 JFS [26]. Such standards facilitate the development of portable applications; moreover, research effort can focus on such standards and thereby have more widespread effect.

2.2 Meta-programming techniques and reflective middleware

Meta-programming techniques have been a focus of research for many years. For example, the Common List Object System (CLOS) is an early example of a sophisticated Meta-Object Protocol (MOP) [17]. Meta-programming techniques were used initially in artificial intelligence research [34, 22], but are now being applied in systems software research, where they can make ORB middleware more dynamically configurable, adaptive, and reflective.

An example of this cross-fertilization is dynamicTAO [18] from the University of Illinois, Urbana Champaign, which illustrates that The ADAPTIVE Communication

Environment (ACE) ORB (TAO) can be reconfigured at run-time by dynamically linking/unlinking certain components. A related effort at Washington University and UCI [38] explores the application of reflective middleware techniques in the context of the CORBA Component Model (CCM) [1]. Yet another example is the Adapt Project [9, 6] at Lancaster University, which is applying a multilevel reflective middleware model focused on dynamic composition of objects.

2.3 Scheduling

The two-level scheduling scheme proposed by Deng, Liu, and Sun [41, 42] can be used as an alternative to Juno for centralized systems, though it is not designed to address the heterogeneous adaptation issues addressed in this thesis. We plan to study how two-level scheduling can be used in conjunction with Juno's meta-programming architecture.

2.4 Real-time distributed resource management

The Realize project [15] has designed and implemented a resource manager for CORBA-based, distributed real-time systems. This project has many characteristics that will make it a good candidate for the application of the concepts exposed in this thesis.

Chapter 3

Terminology and Formalism

This Chapter defines the terminology used throughout the thesis and motivates the assumptions that underlie our work. The formalisms presented here are applicable to a range of different systems, from open DRE system to OSs. For concreteness, however, examples are presented in the context of RT–CORBA 1.0 [25] and the RT–CORBA 2.0 Dynamic Scheduling Joint Final Submission (JFS) [26].

3.1 Properties, Competitors, and Schedulers

We model an open DRE system through *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 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 the focus of

this thesis is on dynamic systems, all the schedulers operate on-line [8], rather than off-line [30].

The remainder of this Chapter presents a formal model for properties, competitors, and schedulers. The advantage of creating a formal model is to enable heterogeneous endsystems, for instance ORBs, 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 endsystem's scheduler.

3.1.1 Properties

Definition 3.1.1 *Let* Π *be the Universe of Properties. A generic element of* Π *is denoted by* π *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_π *is the domain of the property.*
2. $d_\pi \in D_\pi$ *is the default value for the property.*

That is, given any property $\pi \in \Pi$*, its associated domain is denoted by* D_π*, and its associated default value by* d_π*.*
Moreover, given a property $\pi \in \Pi$ *a* Tagged Domain *(*T_π*) of* π *is defined as:*

$$T_\pi = \{\pi\} \times D_\pi = \{(\pi, u) : u \in D_\pi\} \tag{3.1}$$

Given any $e \in T_\pi$ *the property of the tagged element is denoted by* $e.\pi$ *and its value by* $e.value$*.*

RT-CORBA[1] $\Rightarrow$ 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.

[1] Henceforth, our use of the term "RT-CORBA" connotes both static *and* dynamic scheduling capabilities.

Definition 3.1.2 *Given a set of n properties,* $n > 0$*:*

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

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

$$CPD = \{E : |E| = n \ \ and \ \ |E \cap T_{\pi_i}| = 1, \ \ 1 \leq i \leq n\} \tag{3.2}$$

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. The provided definition emphasize the identity of single properties, so each scheduling parameter is treated as a collection of properties, rather than as a typed aggregate of properties.

3.1.2 Competitors

Let $\mathcal{C}$ be the Universe of Competitors.[2] We assume that each competitor exposes a set of properties, as shown in Figure 3.1.

Definition 3.1.3 *We define the following function:*

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

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

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 $c.pval$.

[2] In our case, the universe of discourse is those entities that can compete for the use of resources, and are thus subject to scheduling.

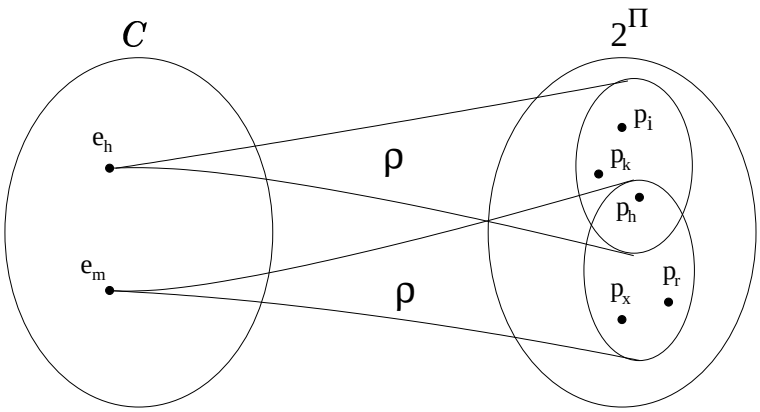

Figure 3.1: Association Between Competitors and Properties

Figure 3.1 shows schematically how the relation ρ works. In this figure, C represents the Universe of Competitors, 2^{Π} 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^{Π}), 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 [10] 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.

3.1.3 Schedulers

Definition 3.1.4 *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 3.1.5 *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*, its characteristic set is indicated with* χs*.*

RT-CORBA $\Rightarrow$ Figure 3.2 shows the characteristic sets the RT-CORBA 2.0 JFS defines for the least laxity first (LLF)[3], EDF, and rate monotonic (RM) scheduling disciplines. If we consider the RT-CORBA 2.0 JFS EDF scheduler, the properties in the scheduler's characteristic set are the *deadline* and the *importance*.[4] 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:

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

[3] 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.

[4] In the canonical EDF definition [19] there is no concept of "importance" but in the RT-CORBA 2.0 JFS there is.

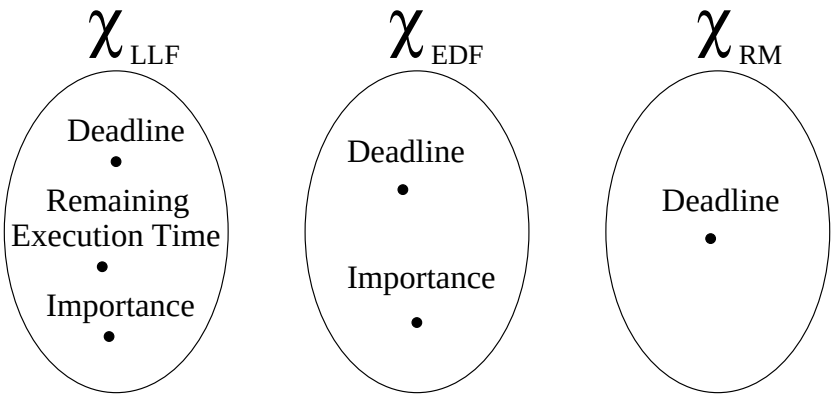

Figure 3.2: Characteristic Set for Least Laxity First (LLF), Earliest Deadline First (EFF), and Rate Monotonic (RM) Scheduling Disciplines

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, any scheduler can be treated 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 ρ. Note that the scheduler partitions the full *Compound Property Domain* of its characteristics into a series of equivalence classes and 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, it is assumed that all schedulers in DRE systems are *well-behaved*, which means that schedulers on different endsystems try to enforce real-time QoS over the properties used to characterize the competitors. Specifically, are not considered pathological cases where schedulers do not work to improve QoS in at least some dimension. For example, a rate monotonic scheduler (RMS) [19] and an EDF scheduler will use different orderings of operations, but they will both work to *improve* the deadline feasibility of operations they schedule.

3.2 Adapters

3.2.1 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 3.2.1 *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:*

$$A_{P_1 \to P_2} : CPD_{P_1} \to CPD_{P_2} \tag{3.3}$$

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 3.3 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 3.3, (endsystem C has a static RM scheduler). As shown in Figure 3.2 these schedulers have different *Characteristic Sets*. As a result, some adaptation will be required when a distributable thread crosses a *scheduling domain*.[5] The claim in this thesis is that the proper type of *Adapter* can handle this adaptation. In addition, Figure 3.3 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.

[5] A scheduling domain is a collection of ORB endsystems using the same scheduling algorithm and properties.

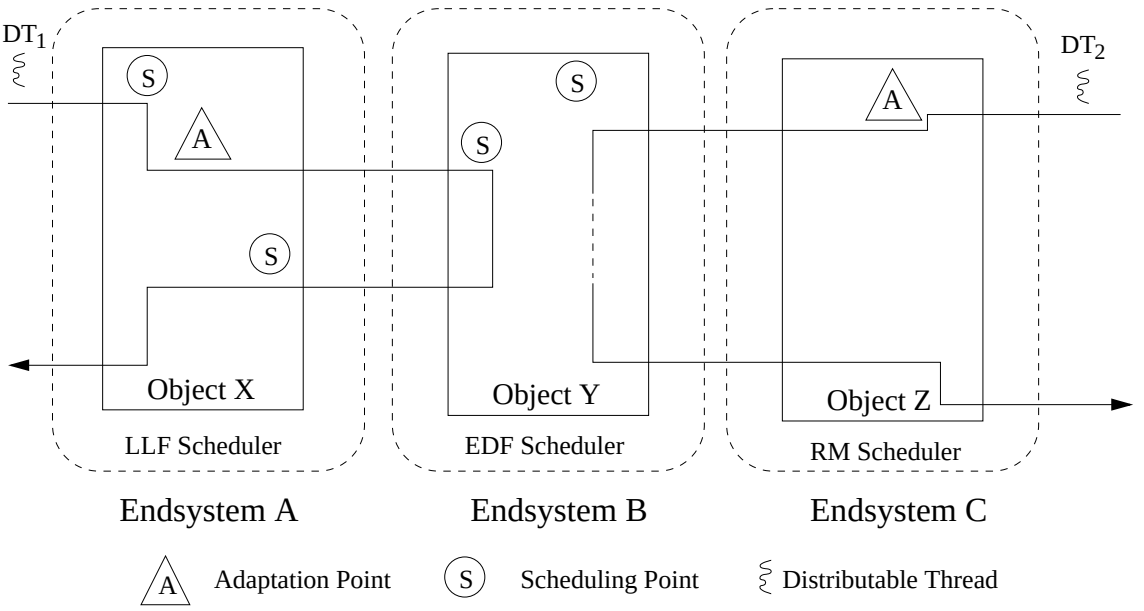

Figure 3.3: A Distributable Thread traversing endsystems that have different Scheduling Disciplines

Figure 3.3 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*.

3.2.2 Reconciling Properties Through Adapters

Now that the terminology and formal model have been defined, we next show how these formalisms can be used to reconcile properties to support interoperability between heterogeneous schedulers. Three relevant cases are shown below; a solution is outlined for each of them.

Case 1: Figure 3.4 shows a scheduler S with a non-empty *Characteristic Set* χ_S, and a 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 χ_S, the following *Restriction Adapter* can be applied:

$$RA : CPD_{\rho(c)} \rightarrow CPD_{\chi_S} \tag{3.4}$$

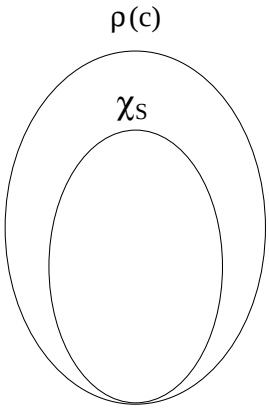

Figure 3.4: The Properties used by a Scheduler are a Subset of Properties Exposed by a Competitor

defined as:

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

Notice that X is a set of *Tagged Properties*; in fact the CPD is a set of sets of *Tagged Properties*. 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 3.3, 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 properties being adapted in a form that can be manipulated efficiently by the scheduler. This form is generally a scheduler-dependent structure that efficiently represents the properties exposed by competitors.

Case 2: Figure 3.5 shows a scheduler S with a *Characteristic Set* χ_S and a competitor c with a non-empty $\rho(c) \subset \chi_S$. To map the properties exposed by the *Competitor* into the ordering of equivalence classes created by the scheduler over χ_S the following *Default Extension Adapter* can be used:

$$DEA : CPD_{\rho(c)} \rightarrow CPD_{\chi_S} \tag{3.5}$$

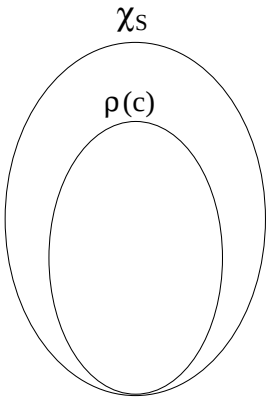

Figure 3.5: The Properties Exposed by the Competitor are a Subset of Properties used by the Scheduler

which is defined as:

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

A variation of the *Default Extension Adapter* is one in which values of the properties exposed by the competitor are used to generate the properties needed by the scheduler, which are not exposed by the competitor. This kind of *Adapter*, hereby called *Extension Adapter* can be defined as the tuple:

$$\langle E, EA \rangle$$

where:

1. $E : CPD_{\rho(c)} \rightarrow CPD_{\chi_S - \rho(c)}$
2. $EA : CPD_{\rho(c)} \rightarrow CPD_{\chi_S}$
   $\forall X \in CPD_{\rho(c)}, \;\; EA(X) = X \cup E(X)$

E is a function that maps the CPD of the properties exposed by a competitor to the remaining property needed by the scheduler i.e. to the CPD of $\chi_S - \rho(c)$. 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 3.3, a *Default Extension Adapter* or *Extension Adapter* could be applied to DT_2 just 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* χ_S and given competitor c with $\rho(c)$, there could be no particular relation between the two set of properties χ_S and $\rho(c)$, as shown in Figure 3.6. In this case a *Generalized Adapter* should

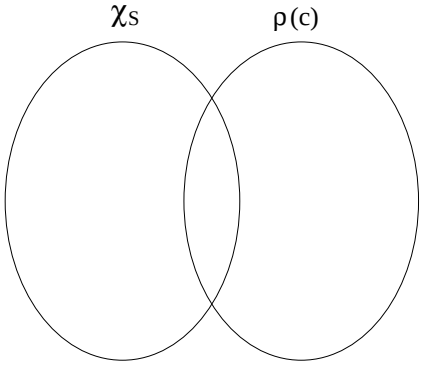

Figure 3.6: No Assumption about the Properties used by the Scheduler and the Properties Exposed by the Competitor

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

$$GA : CPD_{\rho(c)} \rightarrow CPD_{\chi_S} \tag{3.6}$$

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 4.2), *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 appropriate transformation can be generated at run-time by the system or provided by the users or applications.

Chapter 4

Juno A Meta-Programming Architecture for Heterogeneous Middleware Interoperability

This Chapter shows how the formalisms described in Chapter 3 can be used as the conceptual foundation for building DRE middleware that supports interoperability between heterogeneous scheduling disciplines.

We first describe the key concepts of meta-programming and then outline the requirements imposed on a DRE middleware meta-programming framework we developed, called Juno, which implements the formalisms described in [5]. We then present a case study that shows how Juno maps onto the Dynamic Scheduling Real-Time CORBA Joint Final Submission (RT-CORBA 2.0 JFS) [26]. Although we present Juno's architecture in the context of DRE middleware, its applicability extends to other types and levels of middleware and software systems as well.

4.1 Overview of Meta-Programming

The approach taken in this thesis for providing scheduler interoperability is based on *meta-programming* [17, 43, 29]. Meta-programming is a term given to a collection of techniques designed to improve software adaptability by decoupling application behavior from the various cross-cutting aspects [16] and resources used by applications. Meta-programming involves identifying and dissecting programming constructs into the following entities:

- *Base-objects*, which implement certain application-centric functionality; and

- *Meta-objects*, which abstract certain non-functional properties from base-objects—such as persistence, concurrency, scheduling, atomicity, ordering, state, replication, and change notifications—and control various aspects of their behavior at run-time.

Two key concepts in meta-programming are *reification* and *reflection*, which are outlined briefly below:

Reification The process of reifying an aspect of an object consists in making that aspect a *first-class entity*, *i.e.*, something that can be changed and/or controlled via meta-objects. For example, the Java programming language reifies methods and parameters, making them accessible for programmatic introspection. In contrast, methods and parameters in C++ are not first-class entities, *i.e.*, they are not reified in the C++ type system and any introspection must therefore be handled by patterns, such as Reflection [3]. The formal model described in Chapter 3 presents several candidates for reification, such as *properties* and *competitors*.

Reflection Reflection refers to the ability of a system to reason and act upon itself. To make this possible, a reflective system provides a representation of its own behavior that is amenable to inspection (introspection) and adaptation; the representation is causally connected to the underlying behavior it describes. A good reflective interface makes it possible to *open up* a software system implementation, without revealing unnecessary implementation details or compromising portability. Moreover, as described in [34], reflective techniques allow implementation particulars to be exposed in a way that satisfies two important criteria:

1. The access to the implementation is at an appropriately high level of abstraction and
2. The access is *effective*, in the sense that adjustment must actually change the system behavior—the self-representation has to be causally connected to the underlying implementation.

The self-examination property of reflective systems provides them with the ability to adapt their behavior—to cope with changes in their environment. This ability is particularly useful when middleware-based applications are deployed in hostile and/or dynamic environments such as multimedia, group communication, real-time and embedded systems, handheld devices and mobile computing systems. In DRE systems, automatic adaptation must be performed carefully to ensure that distributed systems retain their stability.

In general, the following design constraints must be addressed when applying meta-programming architectures to DRE systems:

Meta-level engineering constraints The manner in which meta-objects reify objects must be designed to minimize meta-level complexity and to ensure run-time efficiency and predictability. In particular, the points at which meta-objects exert control must be designed carefully to avoid incurring significant overhead.

For example, CORBA stubs that provide clients with proxies to target-objects play the role of meta-objects [29]. Stub meta-objects determine how client- and target-objects communicate in the presence of collocation optimizations [39]. In RT-CORBA, stub meta-objects are responsible for ensuring that collocated clients and objects do not incur priority inversion [38].

In the context of this thesis, another example of reification is the scheduler defined in Section 3.1. Reification allows introspection of its characteristic set and control of its behavior. A concrete example of a reified scheduler is the CORBA Scheduling Service [30].

Meta-object protocol engineering constraints Meta-object protocols (MOPs) [43] consist of interfaces defined by meta-objects that provide a way for base-objects to communicate with meta-objects. To avoid synchronization and coherency hazards, middleware developers must carefully design the way in which base-level aspects are reified at the meta-level and reflected back to the base-level after a meta-object has manipulated the reified aspect.

For example, the MOP for the CORBA stub meta-object is implicit, *i.e.*, the client passes control implicitly to the stub as a result of a remote operation invocation. In contrast, a `Meta_Competitor` is a meta-object that provides an explicit MOP that enables its base-objects, *i.e.*, the competitors, to explicitly control the type of adaptation(s) that can be applied to their properties. A competitor may want to disable the use of Restriction Adapters that would discard certain properties crucial for its end-to-end QoS.

4.2 Overview of the Juno Meta-Programming Architecture

To assure scheduler interoperability, a DRE system that implements the formalisms introduced in Chapter 3 must determine how to map properties, competitors, the function $\rho(c)$, schedulers, each scheduler's characteristic set χ_S, and the necessary *Adapters* onto a meta-programming software architecture. Juno's architecture and design are based on the

observation that the function $\rho(c)$ and the characteristic set χ_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 Chapter 3 can be obtained via the Juno meta-programming architecture shown in Figure 4.1. As shown in this figure, competitors and properties are first-class entities, along

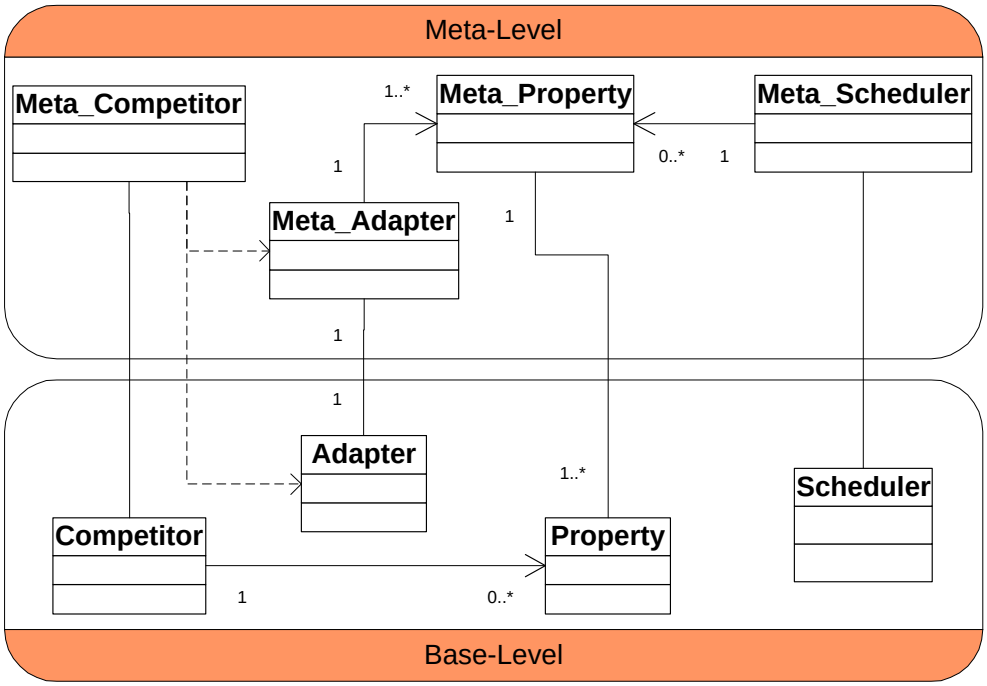

Figure 4.1: UML Class Diagram for Juno's Meta-Programming Architecture

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 χ is exploited by the association between the scheduler meta-object and the property meta-object. The roles of components in Figure 4.1 are summarized below:

Property This class provides an abstraction for the representation of a value of the property domain, as defined in Chapter 3. 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 the endsystem scheduler. It provides an interface for adding and removing competitors and for testing their feasibility. In the context of RT-CORBA, this scheduler would represent the scheduler used by an ORB to schedule distributable threads; a concrete implementation of this class could be an EDF scheduler or an LLF scheduler.

Adapter This class provides an abstract interface for all *Adapter* implementations. As defined in Chapter 3, 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)[1] 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 Chapter 3. As shown in Chapter 3, this association was achieved by using the Tagged Domain of a `Property`. In Juno, this association is achieved by tying each property base object to its meta-object `Meta_Property`.

A `Meta_Property` provides access to the default value of a property; a factory method creates its associated property. This meta-class reconciles property representations that might differ from endsystem to endsystem. For example, a property that represents time could be expressed using different time scales on different systems. One approach would be to require that all time values be specified using the same units; although adequate, this approach is difficult to enforce across separately authored systems. Rather than mandating the units in which time values must be specified, Juno adapts the scales where necessary so that comparisons across systems can make sense.

Meta_Competitor This meta-class manages the transformations required whenever a competitor has properties that do not directly match a scheduler's characteristic set. Juno encapsulates the logic that performs the necessary property reconciliation in the meta-class

[1] A maximize accrued utility (MAU) scheduler associates each competitor with a value-function and the scheduler tries to maximize the value of this function.

`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 a base-object, *i.e.*, an `Adapter`. The term "signature" indicates 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`.

Chapter 5

Dynamic Scheduling for Real-Time CORBA

The previous part of this thesis has described a formalism to enable interoperability of endsystems that use different scheduling disciplines, along with Juno, a meta-programming architecture that represent a reference model for systems that want to implement these concepts. This Chapter shows how the Juno architecture can be applied concretely to solve the interoperability problems that arise in the upcoming Real-Time CORBA 2.0: Dynamic Scheduling Joint Final Submission (RT-CORBA 2.0 JFS) [26]. The context in which Juno is applied is TAO [30], which is a widely-used, CORBA-compliant, open-source ORB designed to support applications with stringent QoS requirements.

5.1 Current Limitations with RT-CORBA 2.0 JFS

The RT-CORBA 2.0 JFS states in Section 5.3 that it does not address interoperability between heterogeneous dynamic schedulers. Moreover, it does not address how to manage client requests that emanate from ORB endsystems that are not scheduled dynamically. These omissions are problematic, because the RT-CORBA 2.0 JFS also requires interoperability with non-real-time CORBA ORB endsystems. As described in Section 5.3, Juno addresses interoperability between dynamically scheduled and non-dynamically scheduled ORBs as a simple, special case of the interoperability problems between dynamically scheduled ORBs.

The RT-CORBA 2.0 JFS also omits explicit capabilities to express scheduling parameters. Section 3 of RT-CORBA 2.0 JFS proposes some reasonable interfaces, which

are shown in Figures 5.4, 5.5, and 5.6. However, these interfaces are not required for RT-CORBA 2.0 implementors. Section 5.2 shows how Juno's architecture can be mapped to that of an RT-CORBA ORB 1.0 (TAO). Section 5.3 then shows how the mapping from one scheduling discipline to another can be automated using Juno.

Based on our discussion thus far, it is clear the following design issues must be resolved:

- *Where should reconciliation occur?* This decision can be determined either by policies or it can be negotiated by ORBs at binding time. Negotiation is necessary if no policies are set to express which model to use. In this case, the two ORBs can exchange meta-properties that describe their schedulers' characteristic sets. Depending on which ORB has an *Adapter* that can handle the potential conversion of properties, one of the two can be chosen. If both ORBs have an *Adapter* then a heuristic can be used to choose between the two.

- *How are* `Adapter`*s retrieved?* Adapters can be configured into an ORB statically (*i.e.*, at build-time) or dynamically (*i.e.*, at run-time using the Component Configurator pattern [31]). It is also possible to retrieve *Adapters* from a distributed registry. In this case, when an ORB does not have an *Adapter* to perform a given conversion, it can retrieve the *Adapter* from the registry. Moreover it is important to notice that some kind of adapter, like *Restriction* and *Extension Adapters*, can be generated on the fly.

- *Which transformations are legal/desirable for a distributable thread?* The type of transformation(s) that can be performed on the properties exposed by a distributable thread can be controlled via policies. For example, it may not be desirable for a distributable thread to have any *Restriction Adapter* applied to its properties. Thus, a policy could be used to express this constraint.

5.2 Juno's ORB Architecture for Interoperable Dynamic Scheduling

Juno's meta-programming architecture described in Chapter 4 can be used as a reference model to realize interoperable RT-CORBA ORBs.[1] As shown in Figure 5.1, a CORBA

[1]Henceforth, the phrase *ORB interoperability* specifically means the interoperability of the ORBs' dynamic schedulers.

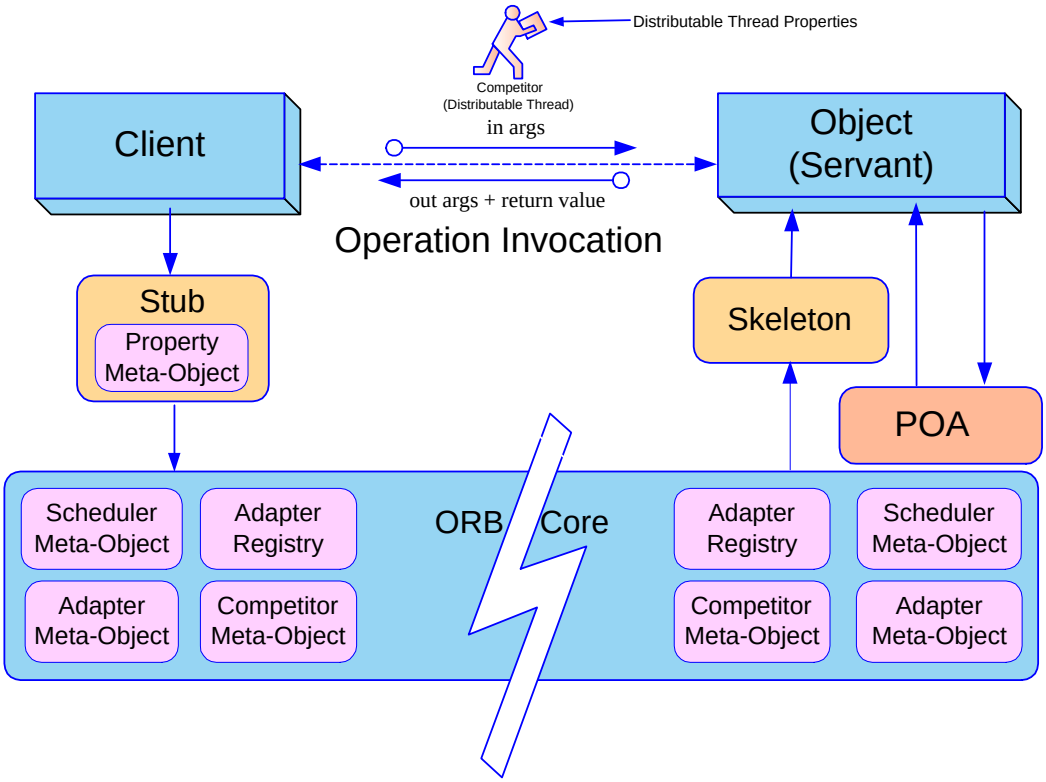

Figure 5.1: An Open and Interoperable RT-CORBA Implementation

ORB can incorporate certain meta-objects present in the meta-layer outlined in Figure 4.1 directly inside the ORB core, whereas other meta-objects can be associated with stubs. In the context of the RT-CORBA 2.0 JFS, a competitor is associated with a *distributable thread*, which can migrate among the following entities:

- A *Home ORB*, which is the ORB where a thread originates.
- *Foreign ORB(s)*, which are any ORBs different from the home ORB that are visited by the distributable thread while performing a remote operation invocation.

Each time a distributable thread transitions from one ORB to another, the new ORB endsystem potentially needs to reconcile properties, such as adapting deadlines and importance to become value functions. The properties of each competitor must be mapped into the *Characteristic Set* of a foreign ORB endsystem's scheduler.

Section 3.2 shows three fundamental cases of property reconciliation. In the context of an ORB, it must be decided where and when reconciliation occurs. As Figure 3.3 suggests, the reconciliation of the properties exposed by a competitor has to occur before the scheduler can perform any scheduling on the competitor. Therefore, as discussed below, it

must happen either right before the distributable thread leaves its home ORB or right after the arrival at the foreign ORB.

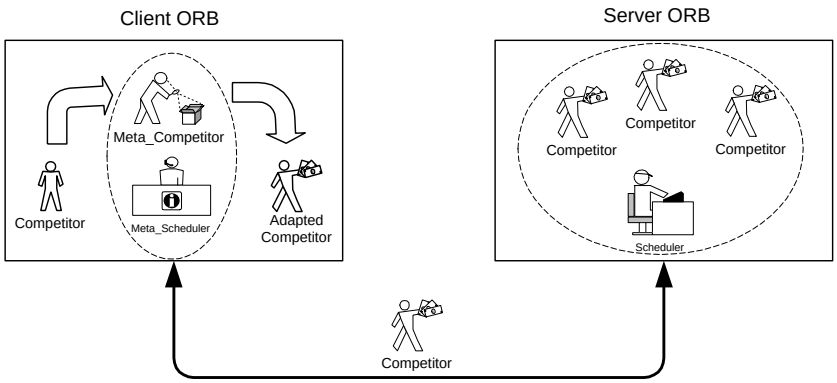

Figure 5.2: Client ORB-based Property Reconciliation

Client-side ORB property reconciliation: In this case, the client ORB should obtain meta-properties that describe the properties used by the server ORB scheduler inside a *tagged component* contained within the object reference of the target object. As shown in Figure 5.1, this information can be retained in the stub, which is the client's proxy for the target object. Whenever a call is made on the remote target object, the stub can use a meta-object protocol (MOP) to communicate the properties used by the foreign ORB endsystem's scheduler to a meta-competitor. The meta-competitor associated with the distributable thread performing this call then reflects the changes needed into the distributable thread. Thus, when the client request arrives at the server ORB, the properties of the distributable thread can be adapted to reconcile with the server ORB scheduler's needs. This scenario is depicted in Figure 5.2.

Server-side ORB property reconciliation: In the case, reconciliation is performed by the server ORB receiving the distributable thread. The meta-scheduler uses a MOP to notify the meta-competitor of the properties the scheduler uses. The meta-competitor then ensures the right *Adapter* is used reconcile the properties. This scenario is depicted in Figure 5.3.

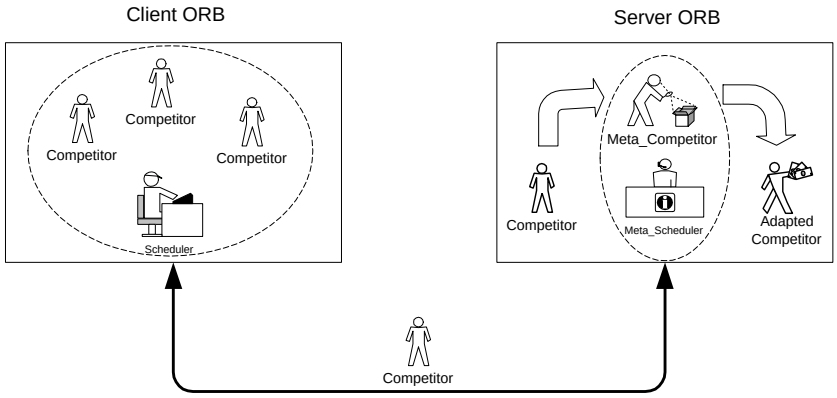

Figure 5.3: Server ORB-based Property Reconciliation

5.3 Adapters for RT-CORBA 2.0 JFS

We next show how to specify an *Adapter* for the sample scheduling disciplines provided in the RT-CORBA 2.0 JFS. The IDL interfaces for these sample scheduling disciplines are outlined in Figure 5.4, 5.5, and 5.6. These IDL definitions are taken from Section 3 of the RT-CORBA 2.0 JFS. Some of these IDL interface are declared as `local`, which makes their instances "locality constrained" CORBA objects. Any attempt to pass a locality constrained reference as an argument to a remote CORBA operation will raise an exception.

Figure 5.4 shows the IDL interface for an earliest deadline first (EDF) scheduler, which uses the deadline of a competitor to determine its execution eligibility (*i.e.*, its scheduling order). In this case, the equivalence classes are characterized by all competitors having the same deadline at a given point in time. Figure 5.5 shows the IDL interface for a least laxity first (LLF) scheduler, which determines the execution eligibility based on *laxity* – defined as the difference between the deadline, the current time, and the estimated remaining computation time. Finally, figure 5.6 shows the IDL interface for a maximize accrued utility (MAU) scheduler, where each competitor is associated with a value function and the scheduler tries to maximize the value of this function. Cases 1 through 4 in Figure 5.7 represent common value functions.

Value functions provide a way of expressing when it is desirable to schedule a task. For example, the cases depicted in Figure 5.7 can be defined as follows:

1. This case represents the situation where there is no value loss or surplus in scheduling the task–*i.e.*, a competitor–at any point in time between its ready time and its deadline. Conversely, after the deadline is passed there is no gain in scheduling the

```
module EDF_Scheduling
{
  local interface Scheduler
    : RTScheduling::Scheduler {};

  struct SchedulingParameter
  {
    TimeBase::TimeT deadline;
    long            importance;
  };

  local interface SchedulingParameterPolicy
    : CORBA::Policy
  {
    attribute SchedulingParameter value;
    static SchedulingParameterPolicy
        create (in SchedulingParameterPolicy value);
  };
};
```

Figure 5.4: Earliest Deadline First Scheduling Parameters Proposed by the RT-CORBA 2.0 JFS

```
module LLF_Scheduling
{
  local interface Scheduler
    : RTScheduling::Scheduler { };

  struct SchedulingParameter
  {
    TimeBase::TimeT deadline;
    TimeBase::TimeT estimated_initial_execution_time;
    long            importance;
  };

  local interface SchedulingParameterPolicy
    : CORBA::Policy
  {
    attribute SchedulingParameter value;
  };
};
```

Figure 5.5: Least Laxity First Scheduling Parameters Proposed by the RT-CORBA 2.0 JFS

```
module MAU_Scheduling
{
  local interface Scheduler
    : RTScheduling::Scheduler {};

  struct SchedulingParameter
  {
    TimeBase::TimeT deadline;
    long            importance;
  };

  local interface SchedulingParameterPolicy
    : CORBA::Policy
  {
    attribute SchedulingParameter value;
  };
};
```

Figure 5.6: Maximize Accrued Utility Scheduling Parameters Proposed by the RT-CORBA 2.0 JFS

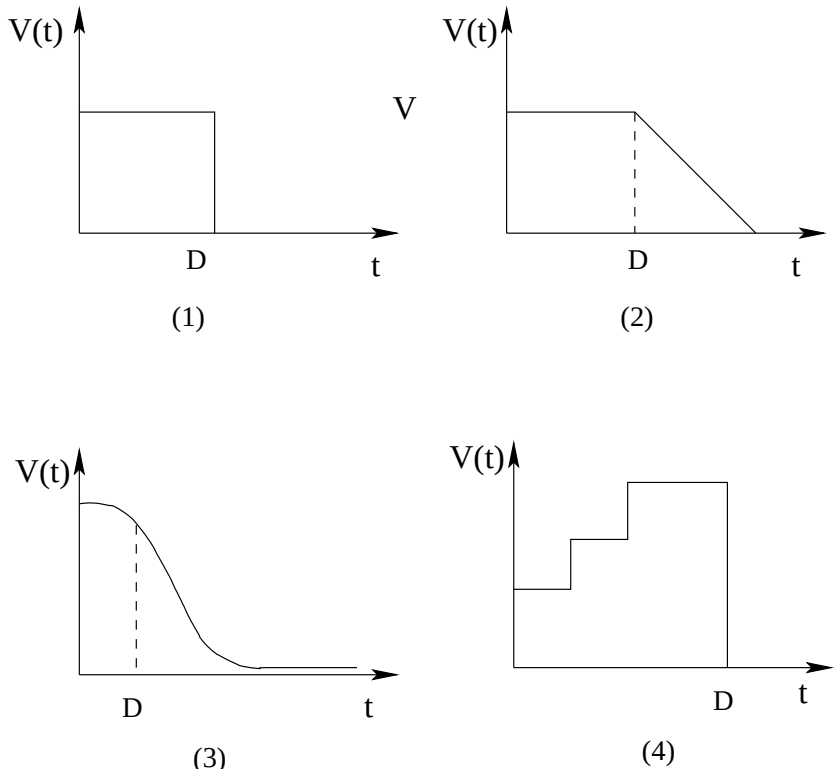

Figure 5.7: Some Value Functions Typically used in Value-based Scheduling

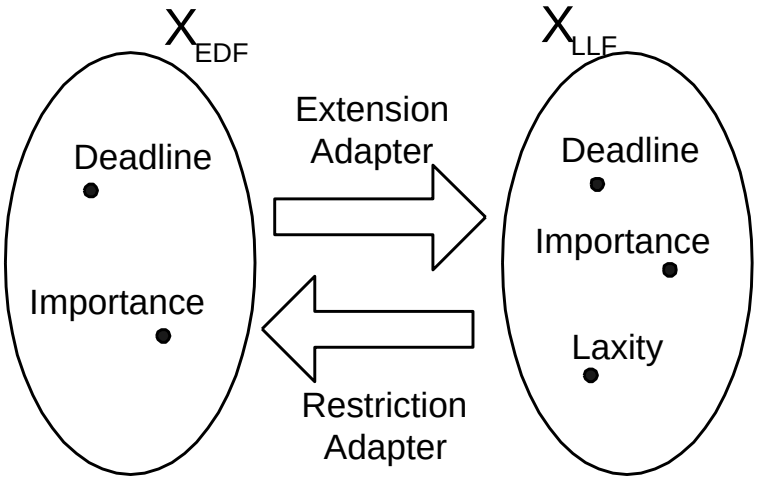

Figure 5.8: EDF-LLF Adaptation of Properties

task. Hard real-time tasks are often specified using this type of function, where the value goes to minus infinity if the task is scheduled after its deadline.

2. The value function depicted in this case shows there is still some value in scheduling a task even after its deadline has been missed, and that this value decreases linearly.

3. This case, like case 2, shows a value function expressing the fact that there is still some value in scheduling a task after its deadline, and value decreases exponentially after its deadline has passed.

4. This case represents the value function of a task in which the benefit of scheduling it increases as time increases, stabilizes at a given point in time, to then drops to zero after its deadline has passed.

The IDL module defined in Figure 5.6 does not define any value functions. The RT-CORBA 2.0 JFS states that each distributable thread is associated with a function that maps the deadline to a value function, but the representation of this function is not specified.

It is interesting to observe the structural similarity between the scheduling properties for the different scheduling disciplines depicted in Figures 5.4, 5.5, and 5.6. It is also interesting to note that the same information, such as deadline and importance, must be provided repeatedly because the scheduling discipline is hard-coded into the competitor– *i.e.*, the distributable thread. We believe this design is overly restrictive since it prematurely binds properties to competitors. Thus, a given scheduler can only use certain properties to enforce the QoS requirements expected by its competitors.

5.4 Dynamic Scheduling Adapters and Use-Cases

we next analyze the different combinations of scheduling disciplines depicted in Figure 5.4, 5.5, and 5.6 (which are taken from Section 3 of the RT-CORBA 2.0 JFS) and show the transformations necessary for achieving interoperability.

Case I: EDF → LLF: Assume that two ORBs exchanging distributable threads have EDF and LLF scheduling disciplines configured, where the LLF ORB is the one that will receive the distributable thread. In this case, an *Extension Adapter* or *Default Extension Adapter* could be used to achieve interoperability, as shown in Figure 5.8.

Figure 5.8 illustrates that the characteristic set of the EDF scheduler is a subset of the characteristic set of the LLF scheduler. Moreover, the two sets differ by just one element. If a *Default Extension Adapter* is used, the value for the estimated initial execution time property would be set by the property's default value, as defined in Section 3.2. Conversely, if an *Extension Adapter* is used, the value for the estimated initial execution property could be provided by the ORB.

Case II: LLF → EDF: This case is the dual of Case I. In this case, we have a distributable thread that is leaving an LLF-scheduled ORB and is moving to an EDF-scheduled ORB. Thus, the transformation should be performed by a *Restriction Adapter*, which simply drops the estimated initial execution time property. In general, it is better to perform the restriction adaptation on the client ORB for the following reason. While an extension adaptation can be performed on the server, we can avoid the extra overhead arising from shipping unused properties across a network.

Case III: MAU →EDF: In this case a distributable thread is going from an ORB that uses a MAU scheduling discipline to an ORB that uses an EDF scheduling discipline. This case is similar to the previous one, in that the characteristic set of a MAU scheduler is a superset of the characteristic set of an EDF scheduler (see Figures 5.4 and 5.6). Thus, a *Restriction Adapter* that ignores the value function will suffice, as shown in Figure 5.9. The same consideration made for the *Restriction Adapter* in Case II applies in this case.

Case IV: EDF →MAU: In this case, the ORB where the distributable thread originates is running an EDF scheduling discipline and the ORB receiving the distributable thread is running a MAU scheduling discipline. We can once again use the *Extension Adapter* shown in Figure 5.9. As shown in Figure 5.4 and Figure 5.6, the MAU scheduler uses an

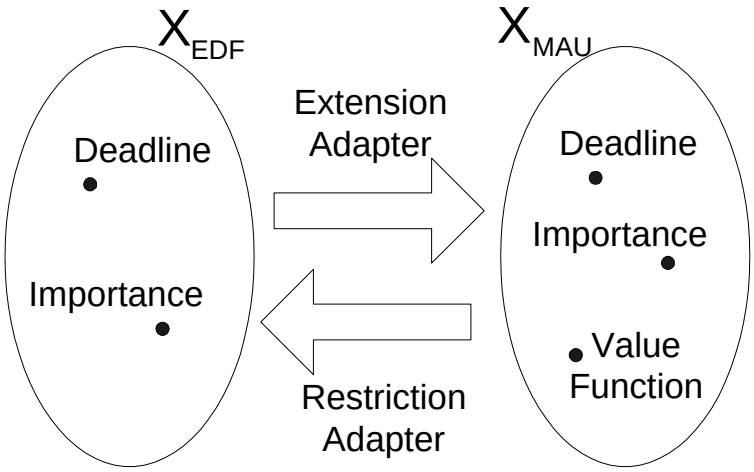

Figure 5.9: EDF-MAU Adaptation of Properties

importance property and a deadline property other than the value function. The value function associated with a competitor could resemble a value function depicted in Figure 5.7, or it could be a generalization of these. In general, the value function might depend on other properties, and the meta-property associated with the value function property can capture this type of information. In the case of a value function property, the property default value is a parameterized function, where the parameters could be other properties. For example, we might assume that the default value for the "property" value function is the parametric function:

$$DefaultValue < D, I > (t) = \begin{cases} I & \text{if } 0 \leq t \leq D \\ 0 & \text{otherwise} \end{cases}$$

where D and I represent the deadline property value and the importance property value, respectively. In this case, we can generate a value function by using an *Extension Adapter* that uses the value associated with the deadline and the importance property.

Case V: LLF→MAU: The transformation described in Case IV can be applied to this case, as shown in Figure 5.10. The difference is that the estimated initial execution time can be removed by using a *Restriction Adapter*.

Case VI: MAU→LLF: This case can be treated similarly to Case II (EDF→LLF). The difference is that a *Restriction Adapter* can be used to remove the value function from the competitors that are leaving the ORB configured with a MAU scheduling discipline (see Figure 5.10). The above six cases have shown how *Adapters* can be created to reconcile the

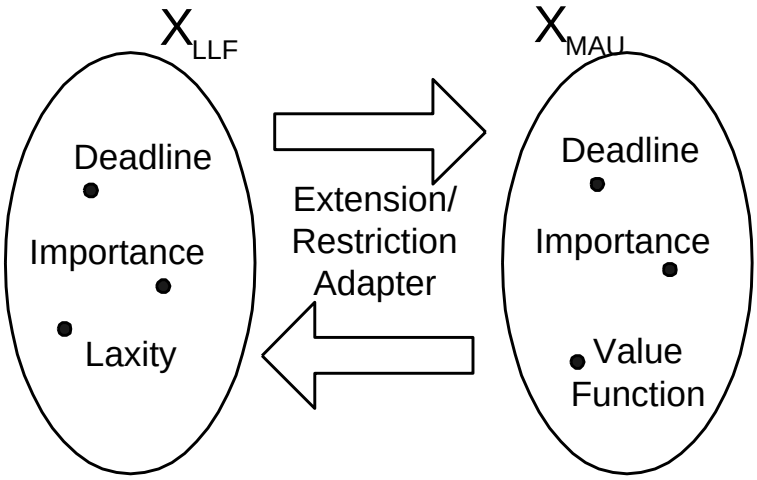

Figure 5.10: LLF-MAU Adaptation of Properties

properties exposed by competitors and used by schedulers. The Table shown below synthesize the adapters that can be used for the different combinations of scheduling disciplines seen so far:

	EDF	**LLF**	**MAU**
EDF	-	EA	EA
LLF	RA	-	EA+RA
MAU	RA	EA+RA	-

RA - Restriction Adapter EA - Extension Adapter

Based on this exercise, we have the following general observations:

- *Restriction Adapter*s trade-off information loss and performance. For example, if a distributable thread traverses different ORBs while serving a remote call, each application of the *Restriction Adapter* can cause information loss, because a restriction could discard some properties that might be needed by another ORB visited later during the remote invocation. This problem only arises when a distributable thread visits more than two ORBs while servicing a request.

- Information loss occurs only when discarding a property that was provided by the user. Conversely, we can always safely drop the property created by an *Adapter* because it can be regenerated “on-demand” by an ORB that needs it.

- At the beginning of this section we mentioned that Juno's meta-programming architecture makes it easier for a dynamically scheduled ORB to interoperate with a regular non-dynamically scheduled ORB. The operation invocation from an object residing on a regular ORB toward an object residing on a dynamically scheduled ORB can be modeled as a distributable thread that exposes no properties. A *Default Extension Adapter* can therefore be used to decorate the competitors with the properties needed by the server ORB scheduler.

- It is important to determine how and when properties that are exposed by competitors should be updated to keep them consistent. For example, consider a distributable thread that has an end-to-end deadline of D and which services a remote operation invocation that traverses multiple ORBs (the number of which is not known *a priory*). At each ORB traversed, the deadline should be updated to avoid the case where each ORB schedules the distributable thread within exactly D, but with the side-effect that the end-to-end time required to perform the call will be actually $n \times D$, where n is the number of ORBs visited by the thread. The RT-CORBA 2.0 JFS does not specify how to handle this situation. For cases where Juno is used to achieve interoperability, however, the meta-competitor provides a location where these properties can be updated consistently.

Chapter 6

Real-Time Event Notification Service

An *Event Notification Service* (ENS) provides a way of delivering events generated from *Suppliers* to *Consumers*. An ENS can be thought as a Mediator [7] between suppliers and consumers; in fact, it encapsulates the event notification process and promotes loose coupling by keeping supplier and consumers from referring to each other explicitly.

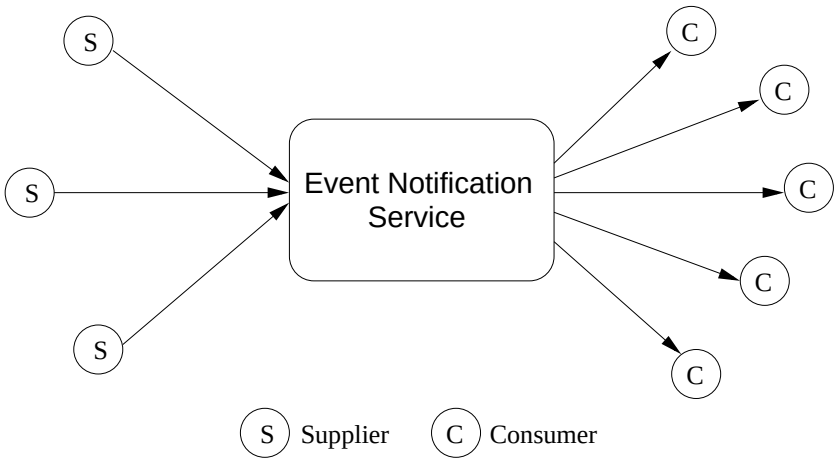

Figure 6.1: An hypothetic Event Notification Service.

In the context of CORBA, the OMG has specified two services, specifically the CORBA Event Service [27] and the CORBA Notification Service [24], that provide event-notification functionality. In many respect, the CORBA Notification Service can be thought as an extension of the CORBA Event Service. In a traditional computing environment, the purpose of an Event Notification Service is to deliver events from suppliers to consumers;

however in DRE systems it is expected to do more. For instance, an ENS should provide event notification with a given QoS. For example, consider a standard two-level process-control system, as shown in Figure 6.2, in which we have:

1. Sensors that make periodic measurement on some of the system's process variables,
2. Base controllers that produce output for the actuators, based on sensor-provided values and control logic,
3. Higher-level controllers that monitor the status of the global process and the outcome of the base controllers to recompute the base controllers' *set-points*, and
4. Actuators that transform the output of the base controllers into some physical action.

There is a producer/consumer [12] relationship between the sensors and the base controllers, between the base controllers and the high-level controller, and between the controllers and actuators. Because of the inherent producer/consumer nature of the problem, the use of an ENS greatly simplifies implementation of this system. If we consider the fact that most current instances of this kind of system tend to have controllers, sensors, and actuators connected through a *fieldbus network*—real-time communication networks becoming popular in automation and process-control environments—this means that the control software needs to be a DSS. To implement such a DSS, CORBA and a CORBA-based ENS could be used as a middleware layer. Note that an ordinary ENS is unsuitable for developing this kind of software; an ENS can deliver events, but it has no understanding of QoS. On the other hand, in a real-time application the correctness depends not only on the fact that an event is received but also on the adherence to timeliness specifications for the received evens. This implies that an *Real-Time Event Notification Service* (RTENS) is required to develop a producer/consumer solution for a real-time system. In hierarchical control systems, it often happens that subscribers belonging to different levels have different *natural way* of expressing the QoS required for their subscriptions. The example depicted in Figure 6.3 represents the high-level structure of an aircraft's auto-pilot control system [33]. This type of system fits the description of a hierarchical control system given above and shown in Figure 6.2. In Figure 6.3, the base controllers are simply control loops that maintain the *roll*, *pitch*, and *yaw* of an aircraft at the desired values, by acting on groups of actuators. The auto-pilot software, based on the state of the plane, provides the *plane trim* to the base controllers. In this system, both the base controllers and the auto-pilot software are consumers of the data provided by the sensors. However, while the

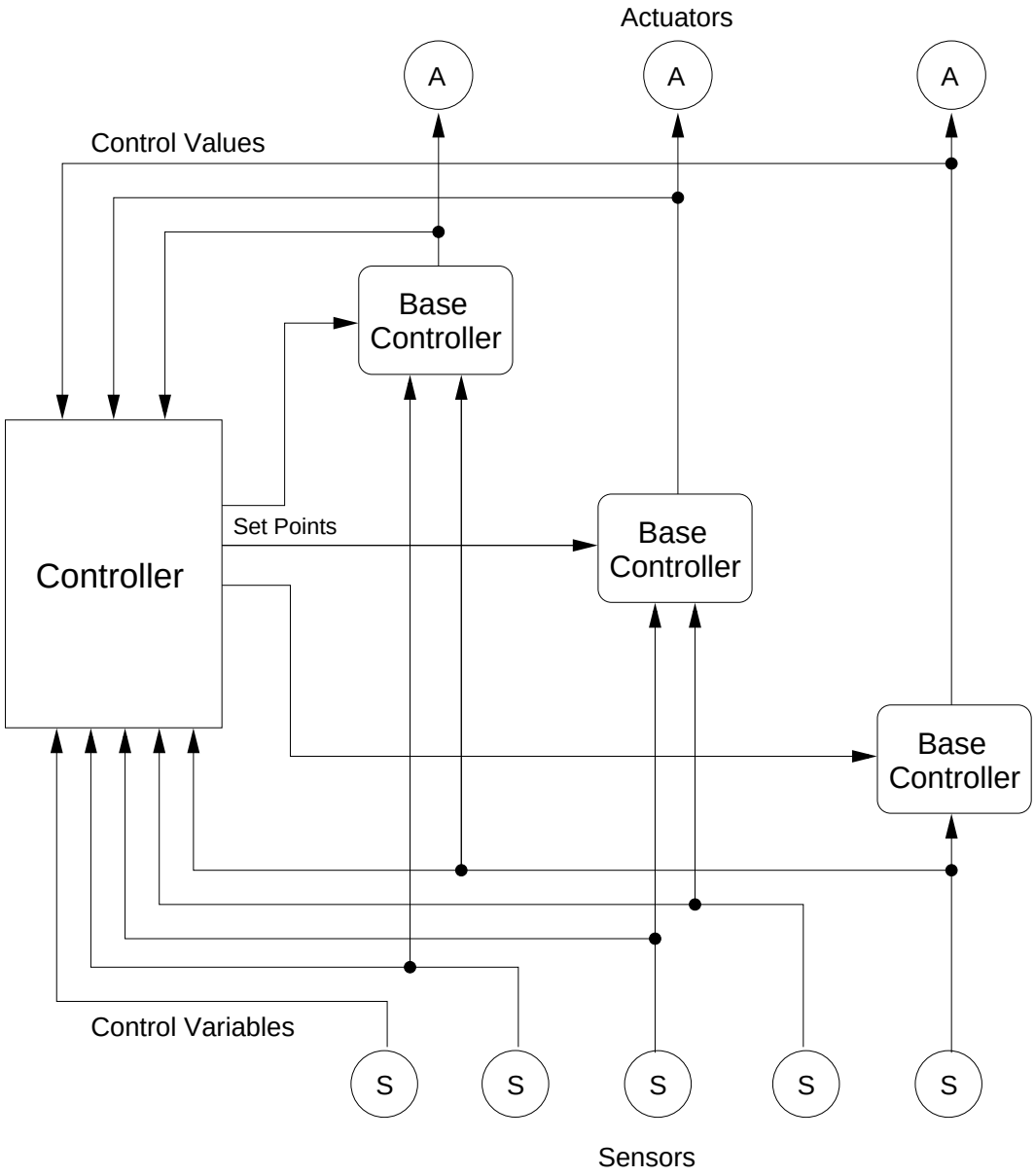

Figure 6.2: A two level control system.

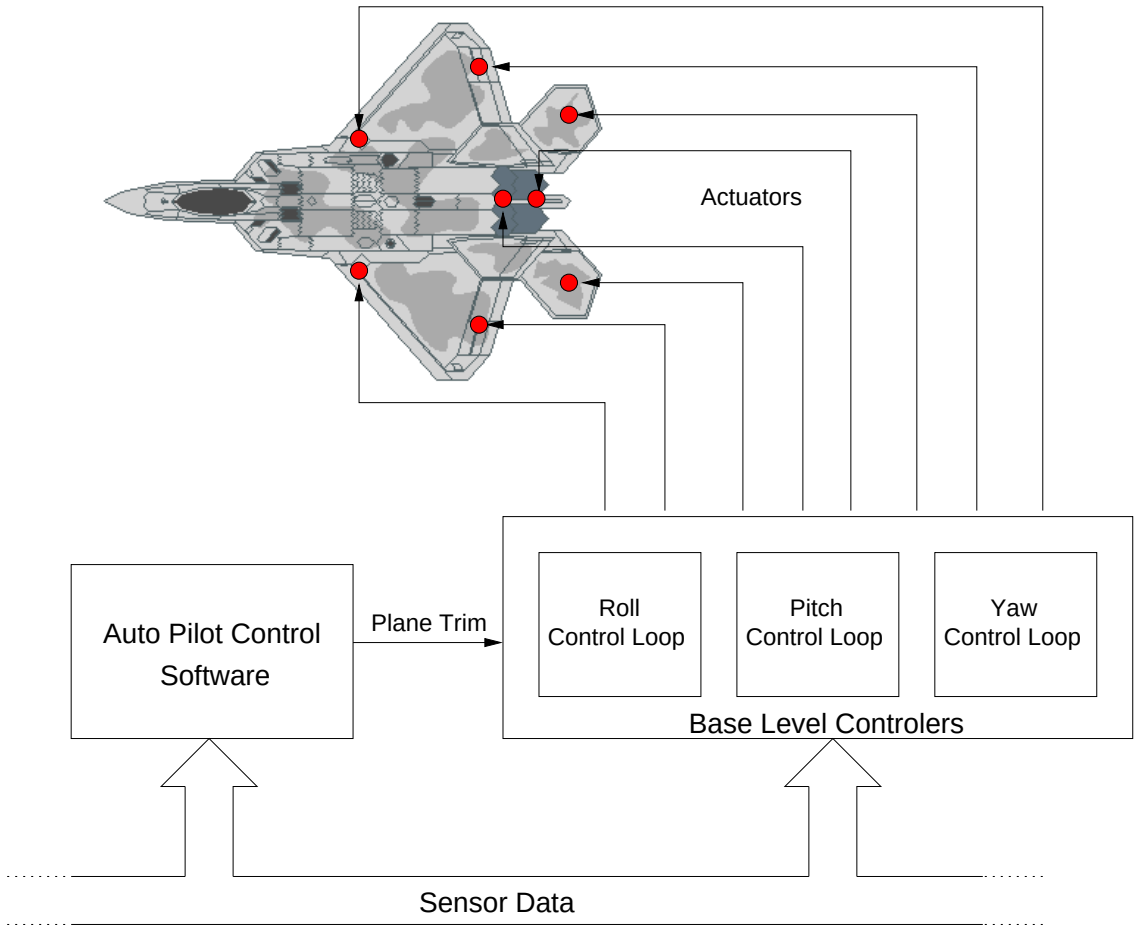

Figure 6.3: Flight Auto-Pilot control system.

base controllers have firm QoS requirements that can be easily expressed using deadlines, the auto-pilot is usually treated as a *soft* real-time task [33]. The QoS requirements with respect to the sensor outputs are usually expressed by value functions. This difference in specification of QoS requirements raises the problem of comparing competitors that expose different properties; as described thus far, this problem can be solved by Juno.

The rest of this chapter will provide a description of the design and implementation of a RTENS that implements the Juno framework.

6.1 Design Forces

In designing any real-time system, attention must be particularly given to the following concerns:

1. Avoiding the potential for priority inversion through proper real-time design techniques,
2. Bounding priority inversions that cannot be avoided,
3. Using data structures, algorithms, and techniques that provide reasonably bounded-time execution where necessary.

An RTENS can be roughly divided into the stages depicted in Figure 6.4, where the first stage takes care of event reception, the second takes care of event processing (examples of actions performed by this stage are dependence analysis and scheduling), and the third stage takes care of event dispatching. In designing and implementing each of these stages, the concerns articulated above must be taken into consideration. We next expand on the modules described above, outlining their architectural and design considerations.

6.1.1 Event Reception Module

Priority inversion can occur in the RTENS during event reception if a less critical or eligible event can be accepted before one that is more critical or eligible. For example, consider a set of periodic event-suppliers. A form of priority inversion occurs if the events are accepted from the network in decreasing order of the events' period (assuming implicitly that the events are all real-time and that their period equals their deadline). Figure 6.5 shows an example of the situation outlined above. Suppose that the first seven events in Figure 6.5 are generated with period $10T$ where T is any dimensional time-constant, while the last

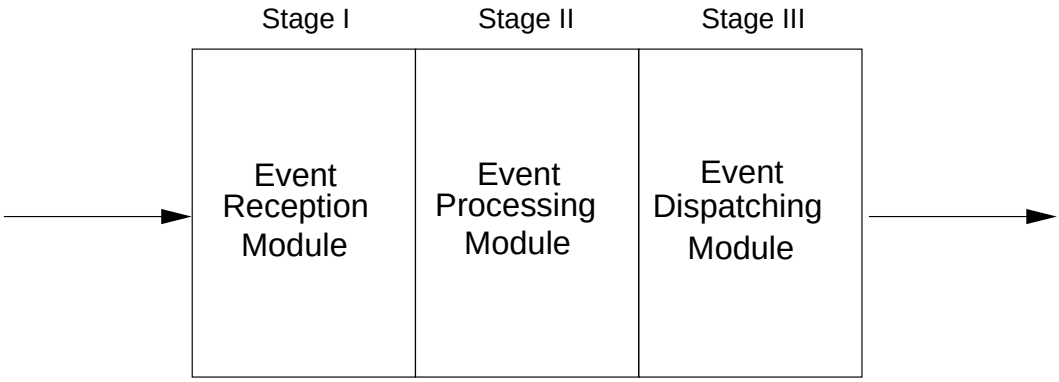

Figure 6.4: Stages of a RTENS.

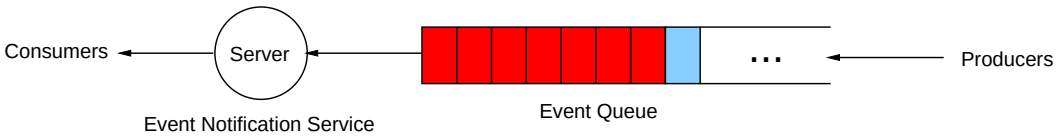

Figure 6.5: Priority Inversion at the receive side of the RTENS.

event is generated with period $5T$. Suppose that the processing and delivery of each event to its consumers takes exactly T time, and that the RTENS can serve only one event at a time. In this case, the last event in the queue would be delivered after $7T$ time units, which would cause the deadline for the last event to be missed.

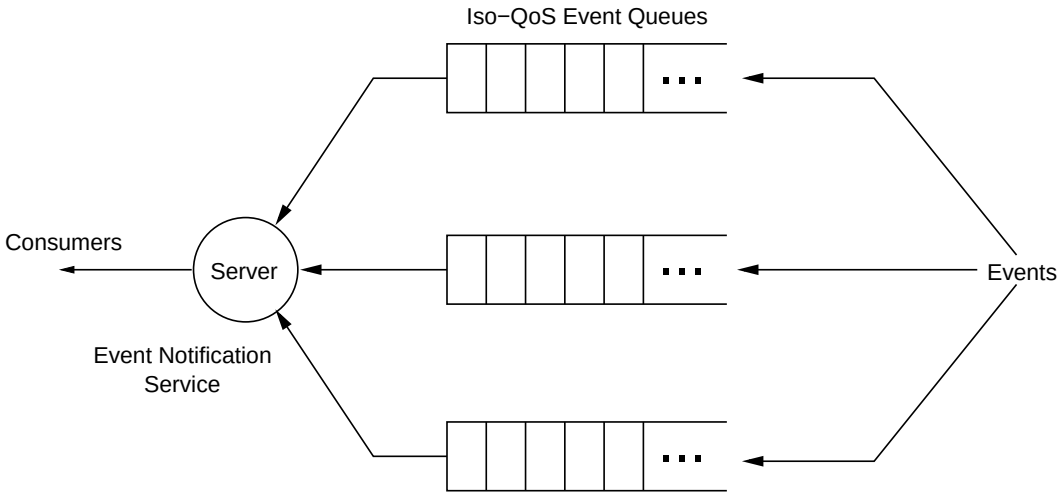

Figure 6.6: Iso-QoS Queue design can reduce the priority inversion on the receive side of the RTENS.

As illustrated in Figure 6.6, priority inversion can be reduced at the receive-end of the RTENS by installing multiple receive queues, each of which is associated with an iso-QoS group of events. In this scheme, events are read from the currently non-empty queue containing the most eligible events first. Each iso-QoS queue receives events that have the same QoS, or the QoS belonging to the same QoS group. Where possible, a QoS group can limit the number of receive-queues by representing a set of events that do not have exactly the same QoS but can be processed adequately by a single queue. This form of grouping can be very effective in limiting the use of system resources. In presence of QoS grouping, some priority inversion is introduced, and it becomes a design issue to trade-off the cost of priority inversion with the use of resources to avoid such inversion. It is worth noticing that and iso-QoS queue represent a class of equivalence as defined in chapter 3, moreover the grouping of an iso-QoS queue can be seen as the partitioning of the space in coarser classes of equivalence.

6.1.2 Event Processing Module

The Event Processing Module typically determines

- How many consumers are registered for the current event,
- Whether some of these consumers require different QoS than others, and
- Whether event correlation has to be performed.

In this module, care should be taken in the way an event's related information is accessed. Ideally, such access is performed in constant-time. Attention should also be paid to the amount of copying/cloning performed in this module, because this can be a major source of overhead. Other design issues peculiar to this module concern how consumers are bound to events of different QoS. New events could be created internally for each such binding, but this is not very efficient. Moreover, the event produced by this stage are subject to scheduling in the dispatch module, and more events imply longer queues and concomitantly longer time to scan those queues.

Another issue present in this module is related to priority inversion. If there are multiple threads of control performing event processing, a reasonable scenario binds each thread with an iso-QoS queue of the reception module. Each thread must access a shared data structure, and so appropriate concurrency control must protect such access. In this scenario, the thread priority-inversion could occur as follows: a high-priority thread waits for a resource already allocated exclusively to a lower-priority thread, while a medium-priority thread starves the lower-priority thread with respect to the CPU [21]. To remedy this kind of priority inversion, it is important that the underlying system provide a mechanism for combating priority-inversion, like, priority-ceiling or priority-inheritance [4, 20], so that the thread can make progress toward releasing the resource needed by the high-priority thread. Moreover, the delay experienced by the high-priority thread must be reasonably bounded.

6.1.3 Event Dispatching Module

The event dispatching module is comprised of two sub-modules: one implements the scheduling disciplines of the RTENS while another is responsible for dispatching events to the consumers. The major issue that arises in designing this module is that the dispatching architecture should be matched to the scheduling discipline. For example if the RTENS uses a Rate Monotonic [4, 20] scheduling discipline, then the dispatcher can be designed

by having a separate queue for events that belong to the same rate group, as shown in Figure 6.7.

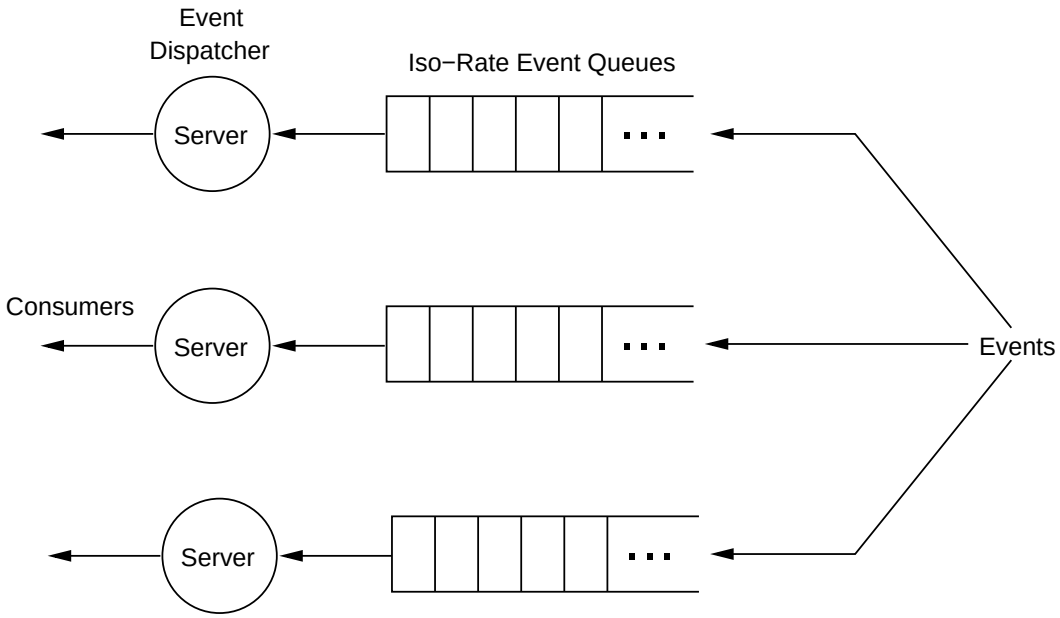

Figure 6.7: Iso-Rate Queue design can exploit the characteristic of Rate Monotonic scheduling policy.

On the other hand if the RTENS uses an EDF scheduling discipline, then it would be better to keep only one queue, as shown in Figure 6.8. In fact, maintaining multiple out-queues for an EDF scheduler is wasteful, because events would have to be moved between queues as their deadline gets closer.

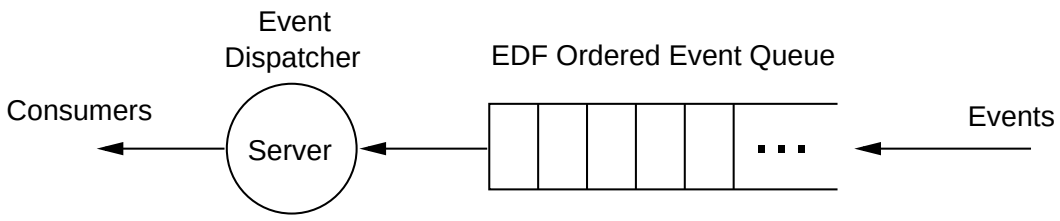

Figure 6.8: Earliest Deadline ordered dispatching Event Queue.

Based on these examples, it should be clear that care should be taken in making the dispatching module easy to configure. In particular based on the scheduling discipline being used the right dispatching scheme should be used.

So far we have concentrated on the part of the RTENS that takes care of receiving events from producers and delivering them to consumers. We have not yet considered the administration aspect of the RTENS. Associated with any RTENS is a control layer which usually has the following responsibilities:

1. Managing the registration of suppliers' provided events,
2. Managing the consumers' subscriptions to events,
3. Performing acceptance tests and feasibility analysis if the RTENS is required to operate in a dynamic environment.

The administration of the RTENS should not interfere with the event-notification functionality, meaning that there should be no resource contention between administration and dispatching of events. In fact, while the load of the events received can be limited by the acceptance test performed by the administration module, it is not possible to restrict the load hitting the administration module. This leads to the idea that the administration and event-notification modules should be in a separately deployable unit. This would let the administration module reside on one machine, and the event notification module be deployed on another machine. Moreover while the event-notification module needs to run on a Real-Time Operating System (RTOS), this is not strictly necessary for the administration module.

Before presenting the design of the RTENS, it is worth emphasizing another set of issues that arises in designing an RTENS: how are QoS concerns specified and in which format; what are the legal combinations of QoS attributes; and where is QoS specified (*i.e.,* consumer, supplier or both).

We next present the design of an RTENS that tries to resolve all the forces presented in this Section, and which uses Juno to enable:

1. QoS specifications independent of scheduling disciplines, and
2. Interoperability of different scheduling disciplines.

6.2 RTENS Architecture

The general architecture of the RTENS proposed in this thesis is depicted in Figure 6.9; Appendix A contains the Interface Definition Language (IDL) that defines the interfaces of the RTENS. The architecture of the RTENS depicted in Figure 6.9 should be regarded as

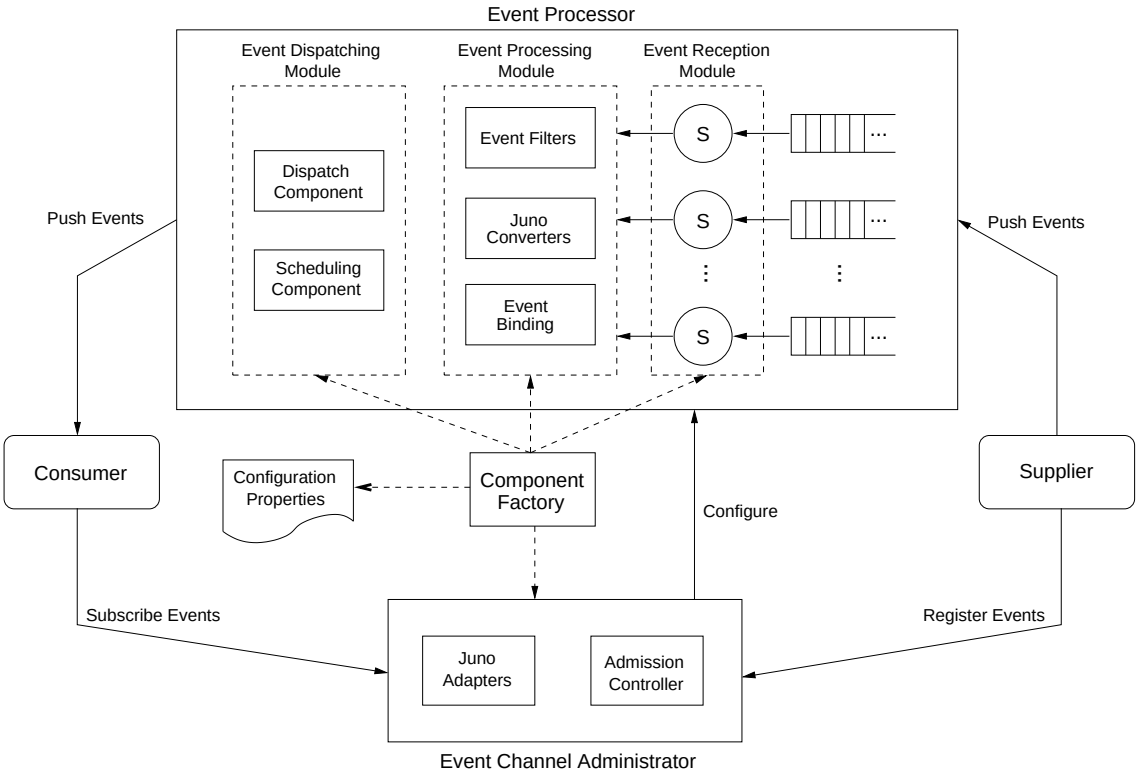

Figure 6.9: RTENS Architecture.

a high-level description of the pieces of an RTENS. Figure 6.9 shows that the RTENS is comprised of two separately deployable modules—the *Event Processor* (EP) and the *Event Channel Administrator* (ECA). The EP's main responsibility is that of delivering events produced by suppliers to consumers, while the ECA's main responsibility is that of managing the registration of events from suppliers, the subscription of events from consumers, the configuration of the EP's resources, the adaptation of QoS, and the admission of new supplier and consumer based on the resource available to the EP. The *Component Factory* (CF) depicted in Figure 6.9 takes care of creating the various components that constitute

the RTENS. A component's creation can be customized by using properties specified in a configuration file, and/or specifying properties at run-time. This allows great flexibility in the configuration and customization of the RTENS. We next provide a detailed explanation of the EP and ECA internals.

6.2.1 Event Processor

Event Reception Module

The event reception module is based on the iso-QoS queues described in Section 6.1.1. The implementation of the iso-QoS queues associates a `PushConsumer` (see Appendix A for IDL) with each iso-QoS queue. A network endpoint is associated with each queue, and the event queuing actually happens in the OS. Each `PushConsumer` associated with an iso-QoS queue runs on a Portable Object Adapter (POA) whose QoS is compatible with that of the queue being served. The required QoS is passed at POA-creation time in the form of a policy containing Juno properties. This properties represent the flow specification of the events that will be pushed to the queue.

`PushConsumer`s are created under request of the ECA. A new `PushConsumer` needs to be created each time a new event is registered whose QoS does not belong to any of the iso-QoS queues already present in the Event Reception Module. `PushConsumer`s are created with a proper `EventHandler`. The event handler to be associated with a `PushConsumer` can be controlled via a property file, or at runtime through a meta-object protocol. `EventHandler`s are always built using the `ComponentFactory`. Each time an event is received by a `PushConsumer`, it lets the associated handler take care of it.

Event Processing Module

The event-processing module maintains constant-access-time data structures for retrieving all the information needed to process an event. Constant-time access is achieved by associating with each event a pair of integers (S_{ID}, E_{ID}). S_{ID} represents the unique supplier ID. Suppliers ID are generated by the ECA the first time a supplier registers an event. This IDs are guaranteed to be sequential and unique. E_{ID} represents an event ID that is unique and sequential within the event generated by a supplier. The pair (S_{ID}, E_{ID}) is used to access a dynamic pool of constant-size arrays in constant time. In this constant-time-access table are stored the binding of the events—which are the consumers subscribed for the event, and the QoS of the subscriptions; also, the table could store the filter that need to be applied to the event.

The event-processing module contains also the Juno *Converters*. Juno Converters are scheduler-specific components that convert Juno properties into data structures that can be efficiently manipulated by the scheduler. The conversion happens each time a new event binding is registered with the EP. In this case the ECA provides the information of the binding, which consists of the tuple $(Event, QoS, Consumer)$, where the QoS element is provided in the form of a Juno property set. When registering a new binding, the EP converts the Juno property set into a scheduler-specific implementation using the converters.

Event Dispatching Module

The structure of the event-dispatching module depends greatly on the scheduling disciplines used by the RTENS. In fact, as described in Section 6.1.3, different scheduling disciplines can take advantage of different dispatching schemes. Once the scheduling policy is chosen, the appropriate dispatching components are created via the `ComponentFactory`. This module has no awareness of the existence of Juno—which is it does not know about adapters or properties—it works with structures that are highly scheduler dependent so to maximize the performance.

6.2.2 Event Channel Administrator

The ECA is the place were Juno's adaptation takes place. The ECA uses Juno *Adapters* to adapt the way an event's QoS is specified to the scheduler's characteristic set of the event processor. The details of this adaptation are explained next.

Registration of Events

Events are registered with the ECA by the suppliers, by providing a specification of the generation pattern of the event by using a Juno property set. Upon reception of such a request, the ECA performs the following steps:

1. The properties that describes the event flow spec are adapted to the properties that are used by the EP Reception Module scheduler. This adaptation is performed using the techniques presented in Chapter 5;

2. Feasibility analysis is performed to check whether the EP has enough resources to accommodate the new event(s);

3. If the feasibility test succeeds, then a check is performed to see whether there is an iso-QoS queue to which the event could be associated. If so, the `PushConsumer` associated with the iso-QoS queue is returned to the supplier; otherwise, the EP is asked to create a new iso-QoS queue, and its associated `PushConsumer` is returned.

4. If this is the first time for the supplier to register any event, then a supplier ID is generated.

The ECA keeps a map of the events generated by each registered supplier. This allows the consumer to query the ECA to retrieve a list of events generated that satisfy certain criteria.

Event Subscription

Consumers subscribe for event notification with the ECA by providing a specification of the QoS with which they wish to receive the event; this is done by using a Juno property set. Upon reception of such a request the ECA performs the following steps:

1. The properties that describe the event subscription are adapted to the properties that are used by the EP Dispatching Module scheduler. This adaptation is performed using the techniques presented in Chapter 5;

2. Feasibility analysis is performed to check whether the EP has enough resources to accommodate the new event(s) subscription;

3. If the feasibility test succeeds, then a binding with the proper QoS is created between the consumer and the event(s) and this binding is communicated to the EP.

6.2.3 Suppliers and Consumers

Suppliers

Each Supplier is provided with a `SupplierAdminProxy`, which lets the supplier register events by using names (i.e. string). The `SupplierAdminProxy` takes care of all the details of the generation of the unique event ID, and also provides a factory method to create the `PushConsumerProxy` through which a consumer can push events. The `PushConsumerProxy` shields the supplier code from the complex details of the RTENS implementation. One of the most important actions performed by the `PushConsumer-Proxy` is that of performing transparently a demultiplexing of the events pushed by the

supplier. As previously described, different events might need to be pushed on different PushConsumers. The PushConsumerProxy provides a constant-time event-dispatch table that associates the events with their target PushConsumer. Other responsibilities performed by a PushConsumerProxy are traffic shaping. Algorithms like leaky bucket [36] could be applied to enforce the flow specification provided when registering the event. The type of PushConsumerProxy created can be specified statically through a property file, or dynamically via a meta-object protocol.

Consumers

Consumers are provided with the ability to register event handlers to execute their *command* when a specific event is notified.

6.3 Hello World Application

In this section, a "Hello World" application will be developed by using the RTENS that has been designed and implemented as part of this thesis.

6.3.1 Hello World Event Supplier

The following listing shows how the Hello World Event Supplier (HWES) code should be written using the API of the RTENS proposed by this thesis. The code on line 19-20 creates a SupplierAdminProxy for the HWES, by passing the name associated with the newly created event supplier. Lines 22-44 create the Juno properties that specify the flow-spec of the Hello World Event (HWE), and put those into a property set. The time scale is specified by the Juno environment (which in this case is msec), but can be changed on a object by object basis. Lines 47-48 show how to register the HWE, while lines 50-53 create the actual HWE. Finally lines 57-60 push an HWE with the period specified in the event flow-spec.

```
1   package edu.wustl.doc.rtevents.demos.helloworld;
2
3   import edu.wustl.doc.rtevents.util.*;
4   import edu.wustl.doc.rtevents.supplier.*;
5   import edu.wustl.doc.rtevents.rtEventChannelAdmin.*;
6   import edu.wustl.doc.rtevents.rtEventComm.*;
7
8   import org.omg.*;
```

```
9   import org.omg.CORBA.*;
10
11  public class Supplier {
12
13      public static void main(String args[])
14          throws Exception
15      {
16        ORBSystem.createORBSingleton(args, null);
17
18        // Create a SupplierAdminProxy for the
19        // Hello World Event Source.
20        SupplierAdminProxy adminProxy =
21           new SupplierAdminProxy("HelloWorldSupplier");
22
23        // Creates the Event Properties
24        juno.MetaNumericValue.NumericValue tv =
25          juno.MetaDimensionalNumericValue.createNumericValue(
26             juno.Juno.ValueSet.Time,
27             new Integer(500));
28        juno.MetaProperty.Property period =
29           juno.MetaProperty.createProperty(
30              juno.Juno.PropertyType.Period,
31              tv);
32
33          juno.MetaNumericValue.NumericValue size =
34              juno.MetaDimensionalNumericValue.createNumericValue(
35                 juno.Juno.ValueSet.Storage,
36                 new Integer(100));
37          juno.MetaProperty.Property storage =
38              juno.MetaProperty.createProperty(
39                 juno.Juno.PropertyType.StorageCost,
40                 size);
41
42          juno.PropertySet properties = new juno.PropertySet();
43
44          properties.addProperty(period);
45          properties.addProperty(storage);
46
47           // Register the event.
48          EventHeader header =
49             adminProxy.registerEvent("HelloWorldEvent",
50                                      properties);
```

```
51
52          Any any = ORBSystem.instance().orb().create_any();
53          any.insert_string("Hello World!");
54          PushConsumerProxy consumerProxy =
55             adminProxy.pushConsumerProxy();
56
57          Event e = new Event(header, any);
58
59          // Push Hello World Events for ever and ever and ever ...
60          // and ever...
61          while(true) {
62            consumerProxy.push(e);
63            Thread.sleep(period.getValue().longValue());
64          }
65      }
66  }
```

6.3.2 Hello World Event Consumer

The following listing shows how the Hello World Event Consumer (HWEC) code should be written using the API of the RTENS proposed by this thesis. First of all, notice (line 11) that the class `Consumer` implements the `PushConsumerOperations` interface. Lines 16-20 implement the code that is executed whenever an event is pushed on this consumer. The action performed in this case is simply that of writing a log message on the standard output containing the contents of the message. Line 37 creates an instance of the `ConsumerAdminProxy` which will be used to subscribe for the HWE. Lines 40-52 create the Juno properties that specify the QoS of the HWE subscription , and put those into a property set. The time scale is specified by the Juno environment (which in this case is msec), but can be changed on a object by object basis. Lines 53-63 simply create a `PushConsumer` that will be registered as a target for the HWE. Finally line 69 subscribes the consumer for the HWE.

```
1   package edu.wustl.doc.rtevents.demos.helloworld;
2
3   import edu.wustl.doc.rtevents.util.*;
4   import edu.wustl.doc.rtevents.consumer.*;
5   import edu.wustl.doc.rtevents.rtEventChannelAdmin.*;
6   import edu.wustl.doc.rtevents.rtEventComm.*;
7
8   import org.omg.*;
```

```
9   import org.omg.CORBA.*;
10
11  public class Consumer implements PushConsumerOperations {
12
13      public void push_vec(Event[] data) {
14
15      }
16      public void push(Event data) {
17        JLogger.instance().logMessage(
18          "Received Event: ["+data.header.supplierID+
19           ", "+data.header.eventID+"]");
20      }
21      public void disconnect_push_consumer() { }
22
23      public static void main(String args[]) throws Exception {
24
25          if (args.length < 2) {
26              System.err.println(
27                 "You need to pass the <SupplierName>\
28                  and the <EventName>");
29              System.exit(-1);
30          }
31
32          String supplierName = args[0];
33          String eventName = args[1];
34
35          ORBSystem.createORBSingleton(args, null);
36
37          ConsumerAdminProxy adminProxy = new ConsumerAdminProxy();
38          // Creates the Event Properties
39
40          juno.MetaNumericValue.NumericValue tv =
41              juno.MetaDimensionalNumericValue.createNumericValue(
42                 juno.Juno.ValueSet.Time,
43                 new Integer(100));
44
45           juno.MetaProperty.Property period =
46              juno.MetaProperty.createProperty(
47                juno.Juno.PropertyType.Deadline,
48                tv);
49
50          juno.PropertySet properties = new juno.PropertySet();
```

```
51
52          properties.addProperty(period);
53          PushConsumer pushConsumer = null;
54          PushConsumerPOATie tie =
55             new PushConsumerPOATie(new Consumer());
56          try {
57              org.omg.CORBA.Object object =
58                  ORBSystem.instance().createObject(tie);
59
60              pushConsumer = PushConsumerHelper.narrow(object);
61          }
62          catch (Exception e) {
63              e.printStackTrace();
64          }
65
66          adminProxy.subscribe(supplierName,
67                               eventName,
68                               properties,
69                               pushConsumer);
70
71          ORBSystem.instance().runLoop();
72      }
73  }
```

Chapter 7

Concluding Remarks

This thesis 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 endsystem schedulers.

This thesis also outlines how the formal model of properties, competitors, and schedulers is reified in Juno. Juno applies *meta-programming techniques* to improve scheduler interoperability in heterogeneous endsystems. Meta-programming techniques are becoming a popular way to enable DRE systems that are adaptable, flexible, configurable, predictable [38] and composable [32].

This thesis has shown how the concept exposed in the first part of the thesis can be applied to the Real-Time CORBA 2.0: Dynamic Scheduling Joint Final Submission (RT-CORBA 2.0 JFS) [26]. A problem present in the RT-CORBA 2.0 JFS–and to a certain extent in Real-Time Java [28]–is that properties of competitors are embedded into structures that are scheduler-specific. In particular, the RT-CORBA 2.0 JFS does not define a way to inspect the properties used by a given scheduler to order the competitors, nor does it define a way to inspect the properties associated with each competitor. Thus, we illustrate how the formalisms defined in Chapter 3 can be mapped onto RT-CORBA to address interoperability problems that are not resolved by the RT-CORBA 2.0 JFS. This thesis has also provided insight on how the design of RTENS can be performed, along with an implementation of a RTENS that implements the Juno reference architecture for scheduling disciplines interoperability.

Future research on Juno might focus on the following topics:

1. Theoretical analysis: 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 the 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 criteria to determine the schedulability of a given set of competitors automatically in an open DRE environment.

2. Empirical evaluation: Extending TAO to support the Juno meta-programming architecture described in Chapter 4. These enhancements would be part of broader efforts to apply reflective middleware techniques [38] and dynamic scheduling [8] to TAO. In these efforts it is essential the development of testbeds to conduct empirical benchmarks that will quantify the QoS provided by Juno. The main goals should be

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

All the source code for Juno and for Juno's Real-Time Event Notification Service can be downloaded from `http://tao.doc.wustl.edu/~corsaro/distrcomp.html`, while 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`.

Appendix A

Real-Time Event Notification Service IDL

This Appendix contains the IDL for the Real-Time Event Notification Service.

A.1 Event Communication IDL

```
module edu { module wustl { module doc { module rtevents {

module rtEventComm {

  // -- Event Structure --
  typedef long SupplierID;
  typedef long EventID;

  struct EventHeader {
    EventID eventID;
    SupplierID supplierID;
  };

  typedef sequence<EventHeader> EventHeaderList;


  struct Event {
    EventHeader header;
    any data;
  };
  typedef sequence<Event> EventSet;
```

```
      interface PushConsumer {
        oneway void push_vec(in EventSet data);

        oneway void push(in Event data);

        void disconnect_push_consumer ();
      };

      typedef sequence<PushConsumer> PushConsumerList;

      interface PushSupplier {
        void disconnect_push_supplier ();
      };

      typedef sequence<PushSupplier> PushSupplierList;

    };};};};};
```

A.2 Event Service Administration IDL

```
1   module edu { module wustl { module doc { module rtevents {

    module rtEventChannelAdmin {

5     // -- Exceptions Declaration --
      exception AlreadyConnected {};

      exception UnknownEvent {};

10    exception UnknownSupplier {};

      // Exception thrown when a supplier/consumer is trying to
      // register a request for resource would lead the
      // EventChannelCore to an unfeasible point.
15    exception UnfeasibleScheduleException {};

      // Raised when two event with the same name, from the same
      // source are registered.
      exception EventAlreadyRegistered {};

20
```

```
// -- Property Set --

// It would have been much better to use valuetypes, but
// JacORB does not support those now :-(
// So instead of passing Juno Properties I need to embed
// those into Octet Sequences.
typedef sequence<octet> PropertySet;
typedef sequence<PropertySet> PropertySetList;

typedef sequence<octet> MetaPropertySet;
typedef sequence<MetaPropertySet> MetaPropertySetList;

// -- EventDescriptor Declaration --

// This structure describes an event completely, by providing
// its name, supplier's name and properties. This structure is
// exposed to the user of the EventChannel to registrate event,
// or to registrate the interest in events.
struct EventDescriptor {
  string eventName;
  string supplierName;
};

typedef sequence<EventDescriptor> EventDescriptorList;

//-- EventSpec Declaration --

struct EventSpec {
  EventDescriptor descriptor;
  PropertySet     properties;

  // Reserved field, used internally by the EC to communicate
  // the ID that have to be associated with the event being
  // registered. With each event are associated 2 numeric ID,
  // and the couple is guaranteed to be unique. One of the two
  // number is associated with the event source, and is unique
  // for all the sources within an event channel, while the
  // other number is associated with the events generated by a
  // supplier, and it is unique within the event generated by a
  // supplier. Moreover the number are all adjacent and start
  // from 1.
  edu::wustl::doc::rtevents::rtEventComm::EventHeader header;
```

```
};

typedef sequence<EventSpec> EventSpecList;

//-- EventSubscription Declaration --

struct EventSubscription {
  EventDescriptor descriptor;
  PropertySet     properties;

  edu::wustl::doc::rtevents::rtEventComm::PushConsumer
  consumer;
};

typedef sequence<EventSubscription> EventSubscriptionList;

////////////////////////////////////////////////////////
//            Admin Interfaces Section
////////////////////////////////////////////////////////

// -- ConsumerAdmin Declaration --
interface ConsumerAdmin {
  edu::wustl::doc::rtevents::rtEventComm::EventHeader
    subsribe_event(in EventSubscription subscription)
    raises(UnfeasibleScheduleException,
           UnknownSupplier,
           UnknownEvent);

  edu::wustl::doc::rtevents::rtEventComm::EventHeader
    subsribe_events(in EventSubscriptionList subscriptions)
    raises(UnfeasibleScheduleException,
           UnknownSupplier,
           UnknownEvent);

  void unsubscribe_event(in EventDescriptor descriptor)
    raises(UnknownEvent, UnknownSupplier);

  void unsubscribe_events(in EventDescriptorList descriptors)
    raises(UnknownEvent, UnknownSupplier);
};

// -- SupplierAdmin Declaration --
```

```
105
      // Interface used by the   supplier to register
      // the  event  they  supply
      interface SupplierAdmin {

110     // Register an event and returns the PushConsumer
        // that should be used to push the event just
        // registered
        edu::wustl::doc::rtevents::rtEventComm::PushConsumer
          register_event(inout EventSpec event_spec)
115       raises(UnfeasibleScheduleException,
                 EventAlreadyRegistered);

        // Register an array of events and returns the
        // PushConsumers that should be used to push the
120     // events just registered
        edu::wustl::doc::rtevents::rtEventComm::PushConsumerList
          register_events(inout EventSpecList event_specs)
          raises(UnfeasibleScheduleException,
                 EventAlreadyRegistered);
125
        // Updates the QoS associated with an event and
        // returns the PushConsumer that should be used
        // to push the event
        edu::wustl::doc::rtevents::rtEventComm::PushConsumer
130       update_event(inout EventSpec event_spec)
          raises(UnfeasibleScheduleException);

        // Update the QoS associated with an event's array and
        // returns the PushConsumer that should be used to
135     // push the event
        edu::wustl::doc::rtevents::rtEventComm::PushConsumerList
          update_events(inout EventSpecList event_specs)
          raises(UnfeasibleScheduleException);

140     // Unregister an event from the EventChannel
        void unregister_event(
          in edu::wustl::doc::rtevents::rtEventComm::EventHeader
          header) raises(UnknownEvent);

145     // Unregister an event list from the EventChannel
        void unregister_events(
```

```
      in edu::wustl::doc::rtevents::rtEventComm::EventHeaderList
      headerList) raises(UnknownEvent);

};


// -- EventChannel Declaration --

interface EventChannel {
  ConsumerAdmin for_consumers();
  SupplierAdmin for_suppliers();
  void destroy();
};

// -- Event Processing --

// This structure represent a binding between an event,
// its consumer and the properties that characterize
// the binding.
struct EventBinding {
  edu::wustl::doc::rtevents::rtEventComm::EventHeader header;
  PropertySet properties;
  edu::wustl::doc::rtevents::rtEventComm::PushConsumer
    eventTarget;
};

typedef sequence<EventBinding> EventBindingList;

// This interface implements the processing of events
// within the Event Channel. It should take care of
// receiving/scheduling/dispatching the events.
interface EventProcessor {

  // JacORB does not currently support valuetype, so I
  // need to implement those in the event channel by
  // streaming serialized Object inside obtect sequences.
  // This breaks the interoperability with languages
  // other than Java, but for the time being this is fine.

  // -- Periodic Event Suppliers --
  edu::wustl::doc::rtevents::rtEventComm::PushConsumer
    addPeriodicPushConsumer(in PropertySet properties);
```

```
190     edu::wustl::doc::rtevents::rtEventComm::PushConsumerList
          addPeriodicPushConsumers(in PropertySetList properties);

        // -- Sporadic Event Suppliers --
        edu::wustl::doc::rtevents::rtEventComm::PushConsumer
195       addSporadicPushConsumer(in PropertySet properties);

        edu::wustl::doc::rtevents::rtEventComm::PushConsumerList
          addSporadicPushConsumers(in PropertySetList properties);

200     // -- Non Real-Time Event Suppliers --
        edu::wustl::doc::rtevents::rtEventComm::PushConsumer
          addNonRealTimePushConsumer(in PropertySet properties);

        edu::wustl::doc::rtevents::rtEventComm::PushConsumerList
205       addNonRealTimePushConsumers(in PropertySetList properties);


        void addEventConsumer(in EventBinding binding);

210     void addEventConsumers(in EventBindingList bindings);

        // Retrieves the scheduler characteristic set of the Event
        // Delivery Module.
        MetaPropertySet eventDeliveryModuleSchedulerChi();
215
        // Retrieves the scheduler characteristic set of the Event
        // Receipt Module.
        MetaPropertySet eventReceiptModuleSchedulerChi();

220   };

    };

    };}; }; };
```

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] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns.* Wiley and Sons, 1996.

[4] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems.* Kluwer Academic Publishers, Norwell, Massachusetts, 1997.

[5] Angelo Corsaro, Douglas C. Schmidt, Ron K. Cytron, and Chris Gill. Formalizing Meta-Programming Techniques to Reconcile Heterogeneous Scheduling Disciplines in Open Distributed Real-Time Systems. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications.*, Rome, Italy, September 2001. OMG. to appear.

[6] 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.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[8] Christopher D. Gill, David L. Levine, and Douglas 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.

[9] Gordon S. Blair and G. Coulson and P. Robin and M. Papathomas. 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.

[10] Timothy H. Harrison, David L. Levine, and Douglas 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.

[11] Michi Henning and Steve Vinoski. *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1999.

[12] Kevin Jeffay. The Real-Time Producer/Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems. *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing : states of the art and practice, pp. 796 - 804*, 1993.

[13] E. Douglas Jensen. Eliminating the Hard/Soft Real-Time Dichotomy. *Embedded Systems Programming*, 7(10), October 1994.

[14] E. Douglas Jensen. Distributed Real-Time Specification for Java. java.sun.com/aboutJava/communityprocess/ jsr/jsr_050_drt.html, 2000.

[15] V. Kalogeraki, P.M. Melliar-Smith, and L.E. Moser. Soft Real-Time Resource Management in CORBA Distributed Systems. In *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, San Francisco, CA, December 1997. IEEE.

[16] Gregor Kiczales. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[17] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of The Metaobject Protocol*. The MIT Press, Cambridge, Massachusetts, 1991.

[18] Fabio Kon and Roy H. Campbell. Supporting Automatic Configuration of Component-Based Distributed Systems. In *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, pages 175–178, San Diego, CA, May 1999. USENIX.

[19] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM*, 20(1):46–61, January 1973.

[20] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River, New Jersey, 2000.

[21] D. Locke, L. Sha, R. Rajkumar, J. Lehoczky, and B. Burns. Priority inversion and its control: An experimental investigation. *Proceedings of the second international workshop on Real-time Ada issues, pp. 39 - 42*, 1988.

[22] Pattie Moes. Computational Reflection, 1987.

[23] J. P. Morgenthal. Microsoft COM+ Will Challenge Application Server Market. www.microsoft.com/com/wpaper/complus-appserv.asp, 1999.

[24] Object Management Group. *Notification Service Specification,* OMG Document telecom/99-07-01 edition, July 1999.

[25] Object Management Group. *The Common Object Request Broker: Architecture and Specification,* 2.4 edition, October 2000.

[26] Object Management Group. *Dynamic Scheduling Real-Time CORBA 2.0 Joint Final Submission,* OMG Document orbos/2001-06-09 edition, April 2001.

[27] Object Management Group. *Event Service Specification Version 1.1,* OMG Document formal/01-03-01 edition, March 2001.

[28] Real-time Java Experts Group. *Real-time Java Specification.*

[29] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming: Resolving Complexity Using ACE and Patterns.* Addison-Wesley Longman, Reading, Massachusetts, 2001.

[30] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications,* 21(4):294–324, April 1998.

[31] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2.* Wiley & Sons, New York, NY, 2000.

[32] Nalini Venkatasubramanian Sebastian Gutierrez-Nolasco. A Composable Reflective Communication Framework. In *Proceedings of IFIP/ACM Workshop on Reflective Middleware 2000,* April 2000.

[33] David Sharp. Personal Communication, 2001. Boeing St. Louis.

[34] Brian C. Smith. Reflection and Semantics in Lisp. In *Procedings 11th Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, Salt Lake City, Utah, January 1984.

[35] R. Sosic. The many faces of introspection, 1992.

[36] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Upper Saddle River, New Jersey, 3rd edition, 1996.

[37] US Navy Program Executive Office (Cruise Missiles and Joint Unmanned Aerial Vehicles). ”Unmanned air vehicle makes successful shipboard landing”. www.mediacen.navy.mil/pubs/allhands/mar00/pg6g.htm, 2000.

[38] Nanbor Wang, Douglas C. Schmidt, Michael Kircher, and Kirthika Parameswaran. Adaptive and Reflective Middleware for QoS-Enabled CCM Applications. *IEEE Distributed Systems Online*, 2(5), July 2001.

[39] Nanbor Wang, Douglas C. Schmidt, and Steve Vinoski. Collocation Optimizations for CORBA. *C++ Report*, 11(10):47–52, November/December 1999.

[40] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. *USENIX Computing Systems*, 9(4), November/December 1996.

[41] J. Sun Z. Deng, J. W.-S. Liu. A Scheme for Scheduling Hard Real-Time Applications in Open System Environment. In *Proceeding of 9th Euromicro Workshop on Real-Time Systems, pp. 191-199*, June 1997.

[42] J. W.-S. Liu Z. Deng. Scheduling Real-Time Applications in a Open System Environment. In *Proceeding of the 18th IEEE Real-Time Systems Symposium (RTSS97)*, June 1997.

[43] Chris Zimmermann, editor. *Advances in Object-Oriented Metalevel Architectures and Reflection.* CRC Press, Boca Raton, FL, 1996.

Vita

Angelo Corsaro

Date of Birth July 18, 1974

Place of Birth Bergamo, Italy

Degrees Laurea Magna Cum Laude, Computer Engineering, Jul 1999

Professional Societies
Association for Computing Machinery
IEEE Computer Society
IEEE Communication Society
Instrumentation, Systems and Automation Society (ISA)

Publications

A. Corsaro, D. Schmidt, C. Gill, R. Cytron (2001) Formalizing Meta Programming Techniques to Reconcile Heterogeneous Scheduling Disciplines in Open Distributed Real-Time Systems, *Proceeding of the 3rd International Symposium on DistributedObjects and Applications.*

M. Kiercher, P. Jain, A. Corsaro, D. Levine (2001). Distributed eXtreme Programming, *Proceeding of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*. **10**(4): 66–71.

S. Cavalieri, A. Corsaro, S. Monforte, G. Scapellato (2001). Multicycle Polling Scheduling Algorithms for Fieldbus Networks, *IEEE Journal of Realtime Systems (to appear).*

S. Cavalieri, A. Corsaro, O. Mirabella, G. Scapellato (1998). Scheduling Periodic Information Flow in Fieldbus and Multi-Fieldbus Environment, *Proceeding of the International Conference on Automation 1998 , Milano Italy*

August 2001