Applying Model-Integrated Computing to Provision Middleware and Application Quality of Service

Nanbor Wang
Christopher D. Gill
{nanbor,cdgill}@cs.wustl.edu
Dept. of Computer Science
Washington University
One Brookings Drive
St. Louis, MO 63130, USA

Douglas C. Schmidt
schmidt@uci.edu
Dept. of Electrical and Computer Engineering
University of California
616E Engineering Tower
Irvine, CA 92697, USA

Aniruddha Gokhale
Balachandran Natarajan
{a.gokhale,b.natarajan}@vanderbilt.edu
Institute for Software Integrated Systems
Vanderbilt University
P.O. Box 36, Peabody
Nashville, TN 37203, USA

Craig Rodrigues, Joseph P. Loyall, Richard E. Schantz and Richard Shapiro
{crodrigu,jloyall,schantz,rshapiro}@bbn.com
BBN Technologies
10 Moulton Street
Cambridge, MA 02138, USA

Abstract

Commercial of-the-shelf (COTS) distribution middleware is gaining acceptance in the distributed real-time and embedded (DRE) community as (1) the cost and time required to develop and verify DRE applications precludes developers from implementing DRE applications from scratch and (2) implementations of standard COTS middleware specifications, such as CORBA, mature. Although standard COTS specifications define the interfaces and policies to provision DRE application resources end-to-end, they do not yet provide sufficient abstractions to separate quality of service (QoS) policy configurations and adaptations from application functionality. DRE application developers must therefore configure QoS policies and program adaptation mechanisms in an ad hoc way. This tight-coupling tends to scatter the code that ensures end-to-end QoS throughout many parts of DRE applications, making it hard to configure, validate, modify, and evolve complex DRE applications consistently.

This paper provides three contributions to the study of the development of QoS-enabled DRE applications. First, we illustrate how standard component-based middleware can be enhanced to flexibly compose static QoS provisioning policies with application logic. Second, we describe how standard component-based middleware can be integrated with adaptive middleware capabilities to flexibly compose dynamic QoS provisioning and adaptation into DRE applications. Third, we illustrate how the Model-Integrated Computing paradigm can be applied to simplify the development of DRE applications and to help generate and validate static and dynamic QoS provisioning for both middleware and applications. Our qualitative and quantitative results show that (1) static and dynamic QoS provisioning improves the performance and helps ensure the end-to-end QoS of DRE systems and (2) decoupling the QoS provisioning logic from the application logic simplifies the maintenance and and evolution of DRE systems.

Keywords:
QoS Provisioning, QoS Adaptation, Middleware, Model-Integrated Computing, Model Driven Architectures, CORBA Component Model

1 Introduction

1.1 Emerging Trends

Commercial-off-the-shelf (COTS) distribution middleware technologies, such as the OMG's CORBA, Sun's EJB/J2EE, and Microsoft's COM+/SOAP/.NET, have matured considerably in recent years. They are increasingly used to reduce the time and effort required to develop applications in a broad range of domains. Historically, these middleware technologies have been applied to *enterprise applications* [1], which are a large class of applications that perform important business functions, such as planning enterprise resource usage, automating key business functions, and managing supply chains and customer relationships. Examples of enterprise applications include airline reservation systems, bank asset management systems, and just-in-time inventory control systems.

More recently, middleware has been applied to distributed real-time and embedded (DRE) applications with stringent quality of service (QoS) requirements for predictability, latency, efficiency, scalability, dependability, and security. There are many types of DRE applications, but they have one thing in common: *the right answer delivered too late becomes the wrong answer.* Examples of DRE applications include *industrial process control systems*, such as hot rolling mill control systems that process molten steel in real-time, and *avionics systems*, such as mission management computers that help aircrafts navigate through their route legs. DRE applications are an increasingly important domain since over 99% of all microprocessors are now used for embedded systems [2] to control physical, chemical, or biological processes and devices in real-time.

Regardless of the domain in which middleware is applied, it helps expedite the application development process by shielding programmers from many accidental and inherent complexities, such as platform and language heterogeneity, resource location, and fault tolerance. *Component middleware* is a maturing class of middleware that enables reusable services to be composed, configured, and installed to create applications rapidly and robustly. In particular, component middleware offers application developers the following reusable capabilities:

- *Connector mechanisms between components*, such as remote method invocations and message passing
- *Horizontal infrastructure services*, such as request brokers, and
- *Vertical models of domain concepts*, such as common semantics for higher-level reusable component services ranging from transaction support to multi-level security.

Examples of COTS component middleware include the CORBA Component Model (CCM) [3], Java 2 Enterprise Edition (J2EE) [4], and the Component Object Model (COM) [5], which use different APIs, different protocols, and different component models.

As the use of middleware becomes more pervasive, DRE applications are increasingly combined to form distributed systems that are joined together by the Internet and intranets. These systems can further be combined with other distributed systems to create "systems of systems." Examples of these large-scale systems of systems include:

- *Just-in-time manufacturing inventory control systems* that schedule the delivery of supplies to improve efficiency and
- *Military command and control systems* that gather and assimilate information from various devices (such as unmanned arial vehicles and wearable computers), present and analyze the information, and coordinate the deployment of available forces and weaponry.

Figure 1 illustrates how the combination of individual manufacturing information systems is fundamental to achieve the

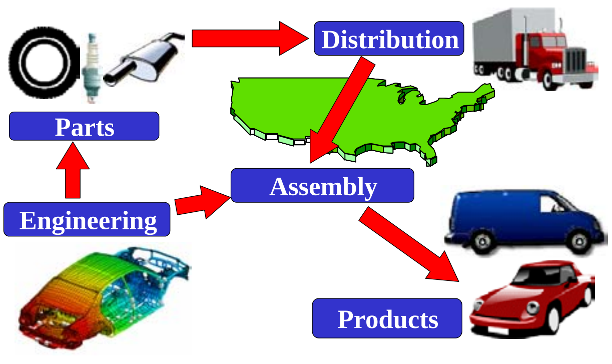


Figure 1: **Characteristics of Manufacturing System of Systems**

efficiencies of modern "just-in-time" manufacturing supply chains. Information from engineering systems is used to design parts, assemblies, and complete products. Parts manufacturing suppliers must keep pace with (1) engineering requirements upstream in the supply chain and (2) distribution constraints and assembly requirements downstream. Distribution must be managed precisely to avoid parts shortages while keeping local inventories low. Assembly factories must achieve high throughput, while making sure the output matches product demand at the sales and service end of the supply chain. Throughout this process, information gathered at each stage of the supply chain must be integrated seamlessly into the control processes of other stages in the chain.

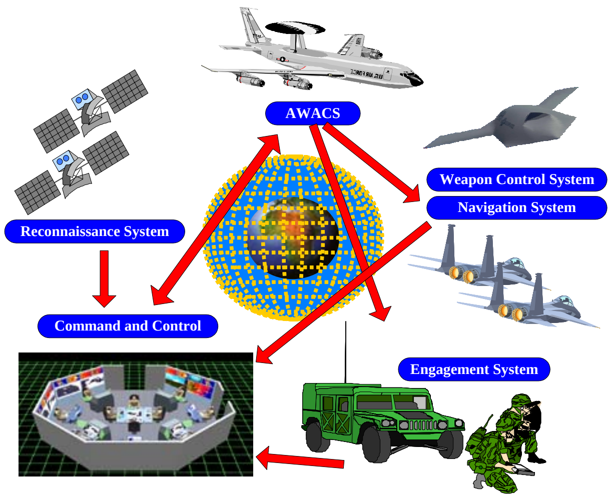


Figure 2: **Characteristics of Military System of Systems**

Figure 2 shows how diverse military information systems are being integrated to counter threats that are increasingly (1) geographically dispersed, (2) elusive in time and space, and (3) threaten asymmetric attacks against infrastructure, people, and places. Intelligence and positioning information from ground troops, surveillance aircraft, and reconnaissance satellites must be fused at command-and-control centers to provide a unified and detailed picture to command personnel. Decisions, such as retasking fighter aircraft or redeploying mobile infantry units, must be communicated rapidly to the involved warfighters. Moreover, supporting information, such as coordinates and imagery, must be assembled and analyzed on-the-fly and sent along with retasking orders.

1.2 Unresolved Challenges

The following key technical challenges arise when developing and deploying large-scale DRE systems outlined above:

1. Satisfying multiple quality of service (QoS) requirements in real-time. An increasing number of DRE applications, such as controllers for surface-mount component pick-and-place machines [6] or total ship computing environments [7], have stringent QoS requirements that must be satisfied simultaneously in real-time. Examples of these QoS requirements include processing resources allocation and network latency, jitter, and bandwidth. To ensure DRE applications can achieve their QoS requirements, various types of *QoS provisioning* must be performed to allocate and manage system computing and communication resources end-to-end. QoS provisioning can be performed in the following ways:

- *Statically*, where the amount of resources required to support a particular degree of QoS is pre-configured into an application. Examples of static QoS provisioning include task prioritization and communication bandwidth reservation. Section 3.1.1 describes the range of QoS resources that can be provisioned statically.
- *Dynamically*, where the amount of resources required are determined and adjusted based on the runtime system status. Examples of dynamic QoS provisioning include runtime reallocations to handle bursty CPU load, primary and second storage, and network traffic demands. Section 3.2.1 describes the range of QoS resources that can be provisioned dynamically.

QoS provisioning in large-scale DRE systems cross-cuts multiple system layers and requires end-to-end enforcement. Existing component middleware technologies, such as CCM, J2EE, and .NET, were designed largely for applications with conventional business-oriented QoS requirements, such as data persistence, encryption, and transactional support. They therefore do not enforce the stringent QoS requirements of DRE applications effectively. What is needed is a *QoS-enabled component middleware* that preserves existing support for heterogeneity in standard component middleware, yet also provides multiple dimensions of QoS provisioning and enforcement to meet the end-to-end QoS requirements of DRE applications.

2. Accidental complexities in integrating software systems. To reduce lifecycle costs and time-to-market, application developers today often assemble and deploy DRE applications by manually selecting the right set of compatible COTS and custom components, which can be a daunting task. The problem is further exacerbated by the existence of myriad strategies for configuring and deploying the underlying component middleware to leverage special hardware and software features. Application developers spend non-trivial amounts of time debugging problems associated with the selection of incompatible strategies and components. What is needed is an integrated set of software development processes and tools that can (1) select and validate a suitable configuration of middleware components and (2) generate optimized configurations automatically.

1.3 A Promising Solution

A promising way to address the DRE software development and integration challenges in Section 1.2 is to combine *Model-Integrated Computing* (MIC) technologies [8] with QoS-enabled component middleware. Model-Integrated Computing is an emerging paradigm for expressing application functionality and QoS requirements at higher levels of abstraction than is possible using third-generation programming languages, such as Visual Basic, Java, C++, or C#. In the context of DRE applications, Model-Integrated Computing tools can be applied to

1. **Analyze** different—but interdependent—characteristics of system behavior, such as scalability, predictability, safety, and security. Tool-specific model interpreters translate the information specified by models into the input format expected by analysis tools. These tools can check whether the requested behavior and properties are feasible given the specified application and resource constraints.
2. **Synthesize** platform-specific code that is customized for particular component middleware and DRE application properties, such as end-to-end timing deadlines, recovery strategies to handle various runtime failures in real-time, and authentication and authorization strategies modeled at a higher level of abstraction.

Combining Model-Integrated Computing and QoS-enabled component middleware effectively is essential to resolve the static and dynamic QoS provisioning challenges of complex DRE systems described in Section 1.2. This paper provides the following three contributions to the successful integration of Model-Integrated Computing and QoS-enabled component middleware that is essential to address these challenges:

- We illustrate how enhancements to standard component middleware can simplify the development of DRE applications by composing QoS provisioning policies statically with applications. Our discussion focuses on a QoS-enabled enhancement of the standard CORBA Component Model (CCM) [3] called the *Component-Integrated ACE ORB* (CIAO), which is being developed at Washington University, St. Louis.

- We describe how dynamic QoS provisioning and adaptation can be addressed using middleware capabilities called *Qoskets*, which are enhancements of the Quality Objects (QuO) [9] middleware developed by BBN Technologies. Our discussion focuses on how Qoskets can be combined with CIAO to compose adaptive QoS assurance into DRE applications dynamically. In particular, Qoskets manage modular QoS *aspects*, which can be combined with CIAO and woven to create an integrated QoS-enabled component model.

- We discuss how QoS-enabled component middleware enables Model-Integrated Computing tools to rapidly develop, generate, assemble, and deploy flexible DRE applications, yet can be tailored readily to meet the needs of multiple simultaneous QoS requirements. Our discussion focuses on the *Component Synthesis with Model-Integrated Computing* (CoSMIC) tools being developed by the Institute for Software Integrated Systems (ISIS) at Vanderbilt University.

1.4 Paper Organization

The remainder of this paper is organized as follows: Section 2 describes how component middleware addresses key limitations of object-oriented middleware, as well as how conventional component middleware fails to support DRE application development effectively; Section 3 illustrates how CIAO component middleware and BBN's QuO middleware framework expand the capability of conventional component middleware to facilitate static and dynamic QoS provisioning and enforcement for DRE applications; Section 4 explains how Model-Integrated Computing and QoS-enabled component middleware can be combined to resolve key challenges associated with DRE application integration and how we are applying these technologies to synthesize component-based applications from high-level models in the CIAO and CoSMIC projects; Section 5 presents empirical results that show how QoS provisioning helps a surveillance and reconnaissance application meet its multimedia mission requirements in a tactical environment; Section 6 compares our work on CIAO, Qoskets, and CoSMIC with related research; and Section 7 presents concluding remarks.

2 Component Middleware: A Powerful Approach to Building DRE Applications

This section motivates the need for component middleware and then presents an overview of component middleware. It also discusses why conventional component middleware fails to support key QoS provisioning needs of DRE applications.

2.1 Overview of Middleware Capabilities

Middleware is reusable software that resides between applications and underlying operating systems, network protocol stacks, and hardware [10]. Middleware's primary role is to bridge the gap between application programs and the lower-level hardware and software infrastructure, to coordinate how parts of applications are connected and how they interoperate. When implemented properly, middleware can help to:

- Shield application developers from low-level, tedious, and error-prone platform details, such as socket-level network programming.
- Simplify the development of distributed applications by providing a consistent set of capabilities that are closer to application design-level abstractions than to the underlying computing and communication mechanisms.
- Provide higher-level abstraction interfaces for managing system resources, such as instantiation and management of interface implementations and provisioning of QoS resources.
- Amortize software lifecycle costs by leveraging previous development expertise and capturing implementations of key patterns in reusable frameworks, rather than rebuilding them manually for each use.
- Provide a wide array of developer-oriented services, such as transactional logging and security, that have proven necessary to operate effectively in a distributed environment.
- Ease the integration of software artifacts developed by multiple technology suppliers.

Various technologies, such as OSF's Distributed Computing Environment (DCE) [11], IBM's MQ Series [12], and CORBA [13], have emerged over the past two decades to alleviate complexities associated with developing software for enterprise applications. Their successes have added the middleware paradigm to the familiar operating system, programming language, networking, and database offerings used by previous generations of software developers. By decoupling application-specific functionality and logic from the accidental complexities inherent in the infrastructure, middleware enables application developers to concentrate on programming application-specific functionality, rather than wrestling repeatedly with lower-level infrastructure challenges.

2.2 Limitations with Object-oriented Middleware

The Object Management Architecture (OMA) in the CORBA 2.x specification [14] defines an object-oriented middleware standard for building portable distributed applications. The CORBA 2.x specification focuses on *interfaces*, which are contracts between clients and servers that define how clients *view* and *access* object services provided by a server. Objects can either be collocated or distributed throughout a network.

Although the CORBA object model has certain virtues, such as location and implementation language transparency, it also has the following limitations [15]:

Lack of functional boundaries. The CORBA 2.x object model treats all interfaces as client/server contracts. This object model does not, however, provide sufficient mechanisms to prevent tight coupling among collaborating object implementations. For example, object implementations that depend on other objects need to discover and connect to those objects explicitly. To build complex distributed applications, therefore, application developers need to program the connections among interdependent services, which can yield brittle and non-reusable implementations.

Lack of generic component servers. CORBA 2.x does not specify a generic *component server* framework to perform common "bookkeeping" work, including initializing the broker and its QoS policies, providing common services such as an event service, and managing the runtime environment of each component. Although CORBA 2.x standardized the interactions between object implementations and object request brokers (ORBs), server developers are still responsible for determining how object implementations are installed in an ORB and the interaction between the ORB and object implementations. The lack of a generic component server standard has yielded tightly coupled, *ad-hoc* server implementations, which increase the complexity of software upgrades and reduce the reusability and flexibility of CORBA-based applications.

2.3 Promising Solution: Component Middleware

In recent years, *component middleware* [16] has emerged to address the limitations with object-oriented middleware outlined above. Component middleware addresses these issues by (1) creating a virtual boundary around application components that interact with each others only through well-defined interfaces and (2) then composing and executing components in generic component servers. The OMG's CCM addresses the limitations with object-oriented middleware described above. It has many similarities to other component middleware frameworks, *e.g.*, EJB and COM+. Sidebar 1 explains the reasons why we base our work on CCM.

Sidebar 1: Motivation for Using the CCM

We base our work on the CCM since CORBA is the only standards-based COTS middleware that has made a substantial progress in satisfying the QoS requirements of DRE systems. For instance, the OMG has adopted the following DRE-related specifications in recent several years:

- **Minimum CORBA**, which removes non-essential features from the full OMG CORBA specification to reduce footprint so that CORBA can be used in memory-constrained embedded systems.
- **Real-time CORBA**, which includes features that allow applications to reserve and manage network, CPU, and memory resources predictably end-to-end.
- **CORBA Messaging**, which exports additional QoS policies, such as asynchronous invocations, timeouts, request priorities, and queueing disciplines, to DRE applications.
- **Fault-tolerant CORBA**, which uses entity redundancy of objects to support replication, fault detection, and failure recovery.

These QoS specification and enforcement capabilities are essential to support DRE systems. Moreover, multiple interoperable and robust implementations of these CORBA capabilities and services are now available. Many of these CORBA implementations are freely-available in open-source format, which is conducive to research and whitebox evaluation. For these reasons, our work focuses on the CCM as the basis for QoS-enabled component models to support DRE systems.

Figure 3 shows an overview of the runtime architecture of the CCM model. *Components* are implementation entities that export a set of interfaces to clients. Components can also express their intent to collaborate with other components by defining *ports*, which consist of the following types of interfaces:

- **Facets**, which define an interface that accepts method invocations from other components synchronously,
- **Receptacles**, which indicate dependencies on synchronous method interfaces provided by other components, and
- **Event sources/sinks**, which indicate a willingness to exchange messages with other components asynchronously.

A *container* provides the runtime environment for a component. It contains various pre-defined hooks that provide strategies, such as persistence, event notification, transaction, and security, to the component it manages. Each container manages one type of component and is responsible for initializing this component and connecting it to other components

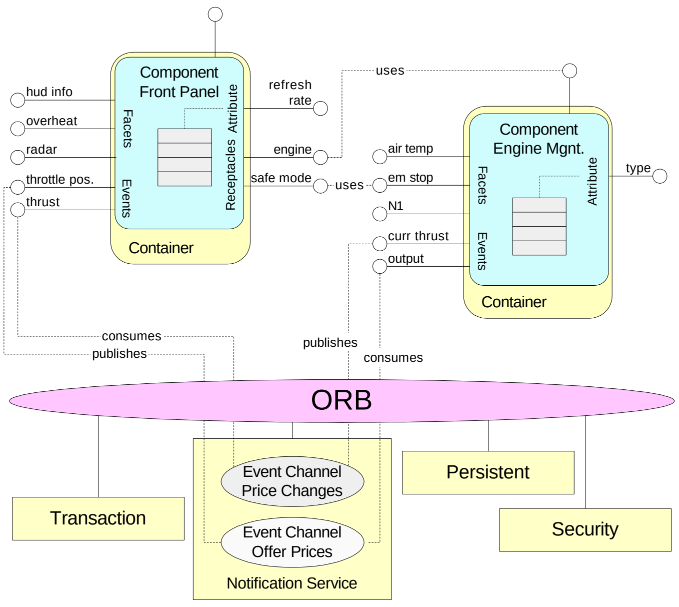


Figure 3: **Overview of the CCM Run-time Architecture**

and ORB services. Developer-specified metadata is used to instruct the CCM deployment mechanism how to create these containers.

In addition to the building blocks outlined above, the CCM also standardizes component implementation, packaging, and deployment. The CCM Component Implementation Framework (CIF) helps generate the component implementation skeletons and persistent state management automatically using the Component Implementation Definition Language (CIDL). The CCM also extends the Open Software Description (OSD) [17], which is a vocabulary of XML defined by W3C to specify component packaging and assembly descriptors. OSD is used by the CCM deployment mechanisms to configure the component connections and containers declaratively.

The tools and mechanisms defined by the CCM collaborate together to address the limits described in Section 2.2. The CCM programming paradigm separates many common concerns of composing and provisioning reusable software components to build an application. This separation of concerns enables programmers to concentrate on the work at hand and separate the role of developers in the application development process. The CCM differentiates the following roles:

- **Component designers**, who define the component features by defining the component interfaces
- **Component implementors,** who develop component implementations
- **Component packagers**, who package component implementations with their default properties
- **Component assemblers**, who select component implementations and compose them into applications
- **System deployers**, who deploy component assemblies into component servers

Although the CCM specification has recently been finalized by the OMG, it still has not been fully incorporated into the core CORBA specification.[1] A number of CCM implementations are available based on the current draft [3], including *OpenCCM* by the Universite des Sciences et Technologies de Lille, France, *K2 Containers* by iCMG, *MicoCCM* by FPX, and *CIAO* by the DOC groups at Washington University in St. Louis. The architectural patterns used in CCM are also used in other popular component middleware technologies, such as J2EE [18] and .NET.

2.4 Limitations with Component Middleware for DRE Systems

Large-scale DRE applications require seamless integration of many hardware and software systems. Figure 4 shows a representative air traffic control system that collects and processes real-time flight status from multiple regional radars across an entire country. Based on the real-time flight data, the system then reschedules flights, issues air traffic control commands to airplanes in flight, notifies airports, and updates the displays in an airport's flight bulletin boards.

The types of systems shown in Figure 4 require complicated application provisioning where developers must connect numerous distributed or collocated subsystems together and define the functionality of each subsystem. Component middleware can reduce the software development effort for these types of systems by enabling application development through composition. Conventional component middleware frameworks, however, are designed with business applications in mind and do not yet support QoS provisioning for DRE applications. Developers are therefore forced to configure and control these mechanisms imperatively in their component implementations.

Although it is possible for component developers to take advantage of certain features in middleware or OS to implement QoS-enabled components by embedding certain QoS provisioning code in component implementations, most features are simply not possible to implement within component implementations. In particular, the following limitations restrict the effectiveness of conventional component models:

- QoS provisioning must be done end-to-end, *i.e.*, it needs to be applied to all interacting components. Implementing QoS provisioning logic internally to a component greatly hampers its reusability.

[1]The latest CORBA 3.0 specification [13] released by the OMG includes only changes in IDL definition and Interface Repository changes from the Component specification.

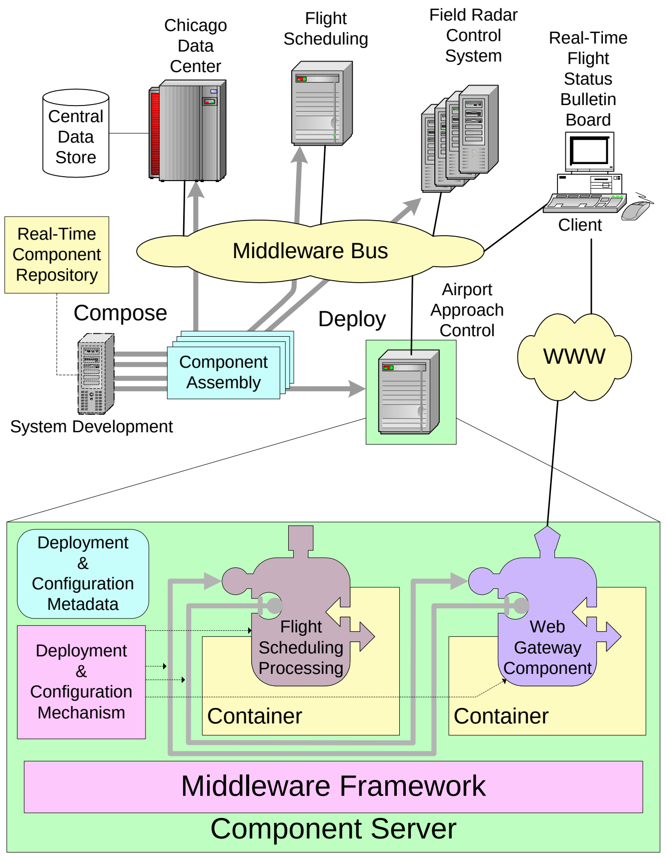


Figure 4: **Integrating DRE Applications with Component Middleware**

- Certain resources, such as thread pools in Real-time CORBA, can only be provisioned within an execution unit, *i.e.*, a component server. Since component developers often have no *a priori* idea of which other components a component implementation will collaborate, the component implementation is not the right level at which to perform QoS provisioning.
- Certain QoS assurance mechanisms, such as configuration of non-multiplexed connections between components, affect component interconnections. Since a reusable component implementation may not know how it will be composed with other components, it is not generally possible for component implementations to perform these types of QoS provisioning in isolation.
- Many QoS provisioning policies and mechanisms require the installation of customized ORB modules to work correctly. Some of these policies and mechanisms, such as high throughput and low latency, however, may be inherently incompatible. It is hard for QoS provisioning mechanisms implemented within components to foresee these incompatibility without knowing the end-to-end QoS requirements *a priori*.

In general, forcing QoS provisioning functionality into component implementations prematurely commits each implementation to a QoS provisioning scenario in a system's lifecycle. This tight coupling defeats one of the key benefits of component models: *separating component functionality from system management*. By creating dependencies between application components and the underlying component framework, component implementations become hard to reuse, particularly in DRE applications with stringent QoS requirements.

3 QoS Provisioning and Enforcement with CIAO and QuO Qoskets

In traditional DRE systems, code for provisioning and enforcing QoS properties is often spread throughout the software and tangled with the application logic. This tangling makes the DRE applications hard to maintain or to extend with new QoS mechanisms and behaviors. As discussed in Section 2.4, a key challenge in QoS provisioning is to decouple the reusable, multi-purpose, off-the-shelf, resource management aspects of the middleware from aspects that need customization and tailoring to the specific preferences of the application.

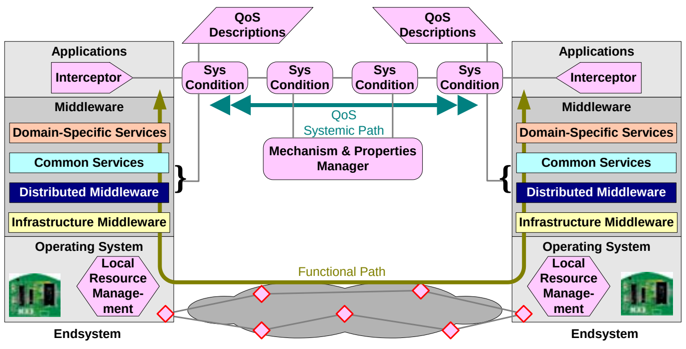


Figure 5: **Decoupling the Functional Path from the Systemic QoS Path**

Based on our experience developing dozens of research and production DRE systems over the past two decades, we have found that it is most effective to separate the programming of QoS concerns along the two dimensions shown in Figure 5 and discussed below:

Functional paths, which are flows of information between client and remote server applications. Distributed middleware is responsible to ensure that this information is exchanged efficiently, predictably, scalably, dependably, and securely between remote nodes. The information itself is largely application-specific and determined by the functionality being provided (hence the term "functional path").

QoS systemic paths, which are responsible for determining how well the functional interactions behave end-to-end with respect to key DRE QoS properties, such as

1. When, how, and what resources are committed to client/server interactions at multiple levels of distributed systems,
2. The proper application and system behavior if available resources are less than expected, and
3. The failure detection and recovery strategies necessary to meet end-to-end dependability requirements.

In next-generation DRE systems, the middleware – rather than operating systems or networks alone – will be responsible for separating QoS systemic properties from functional application properties and coordinating the QoS of various DRE system and application resources end-to-end. The architecture shown in Figure 5 enables these properties and resources to change independently, *e.g.*, over different distributed system configurations for the same application.

The architecture in Figure 5 assumes that QoS systemic paths will be provisioned by a different set of specialists (such as systems engineers, administrators, operators, and possibly automated computing agents) and tools than those customarily responsible for programming functional paths in DRE systems. In conventional component middleware, such as CCM that we described in Section 2.3, there are multiple software development roles, such as component designers, assemblers, and packagers. QoS-enabled component middleware identifies yet another development role called *qosketeer* [9] that is responsible for performing QoS provisioning, such as preallocating CPU resources, reserving network bandwidth/connections, and monitoring/enforcing the proper use of system resources at runtime.

This section describes middleware technologies based on the architecture in Figure 5 that we have developed to

1. Statically provision QoS resources end-to-end to meet key requirements. Some DRE systems, such as avionics mission computing applications, require strict allocation of critical resources via static QoS provisioning.
2. Monitor and manage the QoS of the end-to-end functional application interactions.
3. Enable the adaptive and reflective decision-making needed to dynamically provision QoS resources robustly and enforce the QoS requirements of applications in the face of rapidly changing mission requirements and environmental conditions.

3.1 Static Qos Provisioning and Enforcement via QoS-enabled Component Middleware and CIAO

3.1.1 Overview of Static QoS Provisioning

Static QoS provisioning refers to pre-determining the resources needed to satisfy certain QoS requirements and allocating the resources of a system before or during start-up time. Certain applications use static QoS provisioning because they require tightly bounded predictability for certain functionality in the systems. For example, in avionic control systems, the commands for control surfaces should be assured access to resources *e.g.*, through planned scheduling of those operations or assigning them the highest priority. In contrast, the handling of secondary functions such as flight path calculation, can be delayed without significant impact on the overall system functioning. On other occasions, static QoS provisioning may be used for its simplicity, *e.g.*, a video streaming application for the unmanned air vehicle (UAV) described in Section 5 may choose to simply reserve a fixed network bandwidth for the audio and video streams.

To address the limitations of existing middleware outlined in Section 2.4, it is necessary to make QoS provisioning policies an integral part of component middleware to decouple QoS provisioning policies from component functionality. This separation of concerns relieves component developers from tangling the code to manage QoS resources with the component implementation. It simplifies QoS provisioning that cross-cut multiple interacting components to better ensure end-to-end QoS behavior. Specifically,

- To perform QoS provisioning end-to-end throughout a component middleware system robustly, the static QoS provisioning specifications should be decoupled from component implementations and specified instead in component composition metadata. This separation of concerns helps improve component reusability by preventing a premature commitment to specific QoS provisioning parameters.
- To provision QoS resources that need to be allocated in a component server, component assembly metadata need to be extended to allow allocation and configuration for these resources global to a component server, and be able to associate them with component instances that share these resources.
- Component assembly metadata must also be extended to provision QoS resources for component interconnections.
- To ensure a component server is configured with the mechanisms needed to support the provisioned QoS requirements, component assembly metadata need to be extended to include middleware modules that can configure component servers.

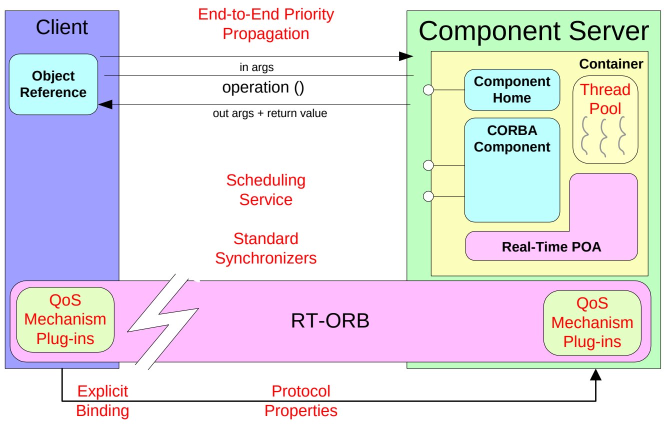


Figure 6: **Examples of Static QoS Provisioning**

Figure 6 illustrates the types of static QoS provisioning that are necessary in large-scale DRE applications:

1. **CPU resources**, which need to be allocated to various competing tasks in a system to make sure these tasks finish on time,
2. **Communication resources**, which the middleware uses to pass messages around to "connect" distributed system together, and
3. **Distributed middleware configurations**, which are middleware plug-ins that a middleware framework uses to realize QoS assurance.

3.1.2 Static QoS Provisioning with CIAO

Figure 7 shows the key elements of the Component-Integrated ACE ORB (CIAO), which is a QoS-enabled implementation of CCM developed at Washington University, St. Louis by extending the TAO ORB [19]. TAO is an open-source, high-performance, highly configurable Real-time CORBA ORB that implements key patterns [20] to meet the demanding QoS requirements of distributed systems. CIAO enhances TAO to simplify the development of DRE applications by enabling developers to statically provision QoS policies end-to-end declaratively when assembling a system.

To support the role of the qosketeer, CIAO makes the following extensions to the CCM to support static QoS provisioning:

Component assembly. A component assembly describes how components are composed into a system. We extend the notion of component assembly to include server-level QoS provisioning and implementations for required QoS supporting mechanisms. We also extend the assembly descriptor format to allow QoS provisioning at the component-connection level.

Client configuration aggregates. We define client-side configuration specifications to configure the client-side ORB

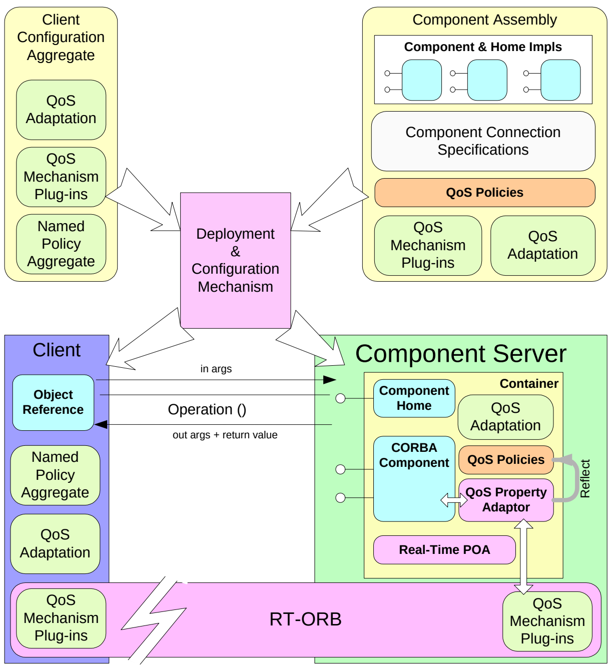


Figure 7: **Key Elements in CIAO**

for support of various QoS provisioning policies. Clients can then associate with named QoS provisioning policies defined in an aggregate, interact with servers, and provide end-to-end QoS assurance. Client configuration aggregates can be installed into a client ORB transparently in CIAO.

QoS-aware containers. They provide the centralized interface for managing provisioned component QoS policies and interacting with QoS assurance mechanisms required by the QoS policies.

QoS adaptations. CIAO also supports installation of meta-programming hooks which can be used to perform dynamic QoS provisioning.

To support these capabilities, CIAO extends the CCM packaging and deployment framework so that system developers can specify the necessary features in component assembly descriptors as various policies. These capabilities enable CIAO to statically provision the types of QoS resources outlined in Section 3.1.1 as follows:

1. **CPU resources** – These policies specify how to allocate CPU resources when running certain tasks, *e.g.*, priority model of a component instance;
2. **Communication resources** – These policies specify ways to reserve and allocate communication resources for component connections, *e.g.*, an assembly can demand a private connection between two critical components in the system, and reserve bandwidth for the connection using the RSVP protocol;

3. **Distributed middleware configuration** – These policies specify the required software modules that control the QoS mechanisms for:
   - **ORB configurations:** The ORB needs to know how to support the functionality required to enable higher level policies, *e.g.*, installing and configuring customized communication protocol.
   - **Meta-programming mechanisms:** Software modules, such as those developed with the QuO Qosket middleware framework, which implement dynamic QoS provisioning and adaptation can be installed statically at system composition time via meta-programming mechanisms, such as smart proxies and interceptors [21].

System developers can use CIAO to decouple QoS provisioning functionality from component implementation and compose these static QoS provisioning requirements into a system at some later point of the development cycle.

3.2 Dynamic QoS Provisioning and Enforcement via QuO Adaptive Middleware and Qoskets

3.2.1 Overview of Dynamic QoS Provisioning

Dynamic QoS provisioning involves the allocation and management of resources at run-time to satisfy certain application QoS requirements. Certain events, such as fluctuations in resource availability or changes in QoS requirements, can trigger reevaluation and reallocation of resources. The following middleware capabilities are needed to support dynamic QoS provisioning:

- To detect changes in available resources, middleware must *monitor* the system status to determine if reallocations are required. For example, network bandwidth is usually shared by multiple applications on modern computers. It is important that middleware for bandwidth-sensitive applications, such as video conferencing, to notice changes in available bandwidth.
- When available resources change, an application may need to *adapt* to the change by adjusting the resources required. For instance, a video conferencing application may choose to lower the resolution temporarily when there is less available network bandwidth to support the original resolution, and switch back to user requested resolution when sufficient bandwidth is available.

As described in Section 1, conventional middleware tries to isolate applications functionality behavioral aspects, such as operation invocations, by abstracting these behavioral aspects under the interface interaction semantic. Although there are ways to implement dynamic QoS provisioning functionality

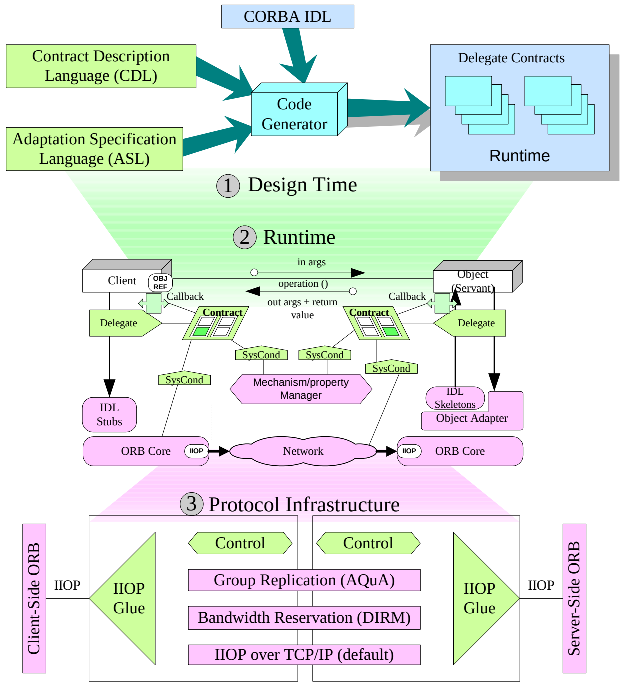


Figure 8: **Examples of Dynamic QoS Provisioning**

in existing applications which use conventional middleware, naïve approaches can yield non-portable code that depends on specific OS features, tangled implementations that are tightly coupled with the application software, and other problems that make it hard to adapt the application to changing requirements. It is therefore essential to separate the functionality of dynamic QoS provisioning from both middleware and application functionality.

Figure 8 illustrates the kinds of dynamic QoS provisioning abstractions and mechanisms that are necessary in large-scale DRE applications:

1. A design time formalism to specify the level of service desired by a client, the level of service an object expects to provide, operating regions indicating possible measured QoS, and actions to take when the level of QoS changes.
2. A runtime mechanism to adapt application behavior based upon the current state of QoS in the system.
3. A set of interfaces to resources and mechanisms in the protocol infrastructure that need to be measured and controlled dynamically.

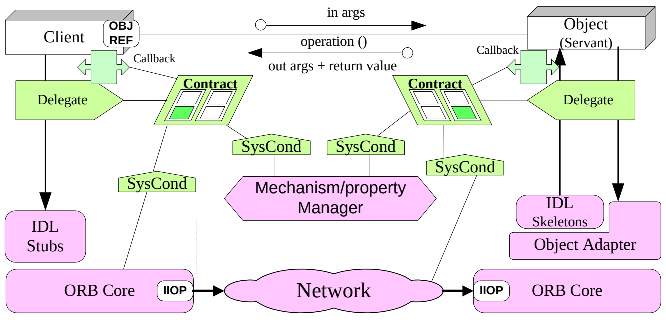


Figure 9: **Elements in the QuO Architecture**

3.2.2 Overview of QuO

Quality Objects (QuO) [9] is an adaptive middleware framework developed by BBN Technologies that allows the DRE developer to use aspect-oriented software development [22] techniques to separate the concerns of QoS programming from application logic in DRE applications. The QuO framework allows DRE developers to specify (1) their QoS requirements, (2) the system elements that must be monitored and controlled to measure and provide QoS, and (3) the behavior for adapting to QoS variations that occur at runtime.

Figure 9 illustrates how the elements in QuO support the following dynamic QoS provisioning needs:

- **Contracts** specify the level of service desired by a client, the level of service an object expects to provide, operating regions indicating possible measured QoS, and actions to take when the level of QoS changes.
- **Delegates** act as local proxies for remote objects. Each delegate provides an interface similar to that of the remote object stub, but adds locally adaptive behavior based upon the current state of QoS in the system, as measured by the contract.
- **System condition objects** provide interfaces to resources, mechanisms, and ORBs in the system that need to be measured and controlled by QuO contracts.

QuO applications can also use resource or property managers that manage given QoS resources, such as CPU or bandwidth, or properties, such as availability or security, for a set of QoS-enabled server objects on behalf of the QuO clients using those server objects. In some cases, managed properties require mechanisms at lower levels in the protocol stack, such as replication or access control. To support this, QuO includes a gateway mechanism [23], which enables special-purpose transport protocols and adaptation below the ORB.

For more information about the QuO adaptive middleware, see [9, 23, 24, 25].

3.2.3 Qoskets: QuO Support for Reusing Systemic Behavior

One goal of QuO is to separate the role of the systemic QoS programmer from that of an application programmer. A complementary goal of this separation of programming roles is that systemic behaviors can be encapsulated into reusable units that are not only developed separately from the applications that use them, but that can be reused by selecting, customizing, and binding them to an application program. To support this goal, we have defined *Qoskets* as a unit of encapsulation and reuse of systemic behavior in QuO applications. A Qosket is each of the following, simultaneously:

- **A collection of cross-cutting implementations**, *i.e.*, a Qosket is a set of QoS specifications and implementations that are woven throughout a distributed application and its constituent components to monitor and control QoS and systemic adaptation.
- **A packaging of behavior and policy**, *i.e.*, a Qosket generally encapsulates elements of an adaptive QoS behavior and a policy for using that behavior, in the form of contracts, measurements and code to provide adaptive behavior
- **A unit of behavior reuse**, largely focused on a single property, *i.e.*, a Qosket can be used in multiple applications, or in multiple ways within a single application, but typically deals with a single attribute (*e.g.*, performance, dependability, security)

Qoskets are a first step towards individual behavior packaging and reuse, as well as a significant step toward the more desirable (and much more complex) ability to compose behaviors within an application context. They are a means toward the larger goal of flexible design tradeoffs at runtime among properties such as real time performance, dependability and security, varying with current operating conditions. Qoskets are used to bundle in one place all of the specifications and objects for controlling systemic behavior, independent of the application in which the behavior might end up being used, and whether or not the behavior will be used in-band or out-of-band.

Qoskets encapsulate the following systemic QoS aspects:

- **Adaptation policies**, as expressed in QuO contracts
- **Measurement and control**, as defined by system condition objects and callback objects
- **Adaptive behaviors**, as defined by ASL specifications, partially specified as templates until they are specialized to a functional interface, and by contract transitions and states.
- **QoS implementation**, as defined by Qosket methods.

A Qosket is a collection of the interfaces, contracts, system condition objects, callback objects, unspecialized adaptive behavior, and implementation code associated with a reusable piece of systemic behavior. A Qosket is specified by defining the following:

- The contracts, system condition objects, and callback objects it encapsulates;
- ASL template code, defining partial specifications of adaptive behavior.
- Implementation code for instantiating the Qosket's encapsulated objects, for initializing the Qosket, and for implementing the Qosket's defined systemic measurement, control, and adaptation;
- The interfaces that the Qosket exposes.

A Qosket can be instantiated and used in either or both of two ways:

1. Used by the application through an adapter to provide out-of-band adaptation and QoS management.
2. Used in conjunction with a QuO delegate, *i.e.*, specialized to a particular functional interface to provide in-band adaptation and QoS control.

The general structure of Qoskets, objects they encapsulate, and interfaces they expose are illustrated in Figure 10. The

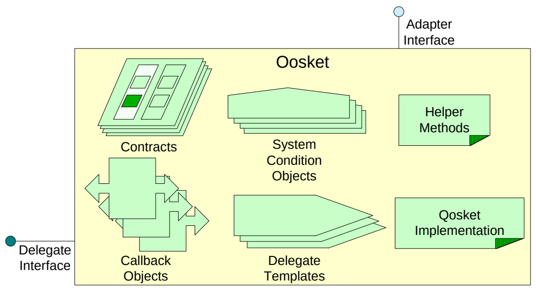


Figure 10: **Qoskets Encapsulate QuO Objects into Reusable Behaviors**

two interfaces that qoskets expose correspond to these two use cases:

- **The adapter interface**, which is an application programmer interface. This interface provides access to QoS measurement, control, and adaptation features in the Qosket (such as the system condition objects, contracts, and so forth) so that they can be used anywhere in an application.
- **The delegate interface**, which is an interface to the in-band method adaptation code. In-band adaptive behaviors of delegates are conveniently specified in the QuO ASL language. The adaptation strategies of the delegate are conveniently encapsulated, and woven into the application using code generation techniques.

3.3 Integrated QoS provisioning via CIAO and Qoskets

As discussed in Section 3.2.3, Qoskets provide abstractions for dynamic QoS provisioning and adaptive behaviors. However, the current implementation of Qoskets in QuO requires application developers to modify their application code manually to "plug in" the behavior into existing applications. Instead of retrofitting DRE applications to use Qosket specific interfaces, it would be more desirable to use existing and emerging COTS component technologies and standards to encapsulate QoS management.

Conversely, although CIAO allows system developers to compose static QoS provisioning, adaptation behaviors, and middleware support for QoS resources allocating and managing mechanisms into DRE applications transparently as depicted in Section 3.1.2, CIAO does not provide an abstraction to model, define, and specify dynamic QoS provisioning. We can take advantage of CIAO's capability to transparently configure Qoskets into component servers and provide an integrated QoS provisioning solution, which enables the composition of both static and dynamic QoS provisioning into DRE applications.

The static QoS provisioning mechanisms of CIAO enables the composition of qoskets into applications as part of component assemblies. As shown in Figure 11, CIAO installs a

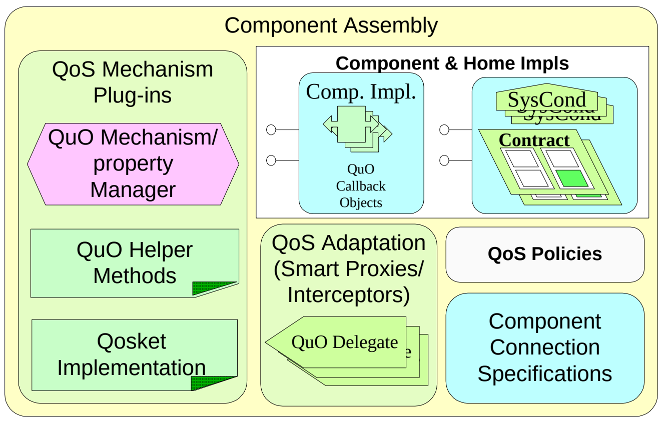


Figure 11: **Composing a qosket using CIAO**

qosket using using the following mechanisms:

- QuO delegates can be implemented as smart proxies or portable interceptors [21] and injected into component servers using assembly descriptors and the client-side configuration aggregates described in Section 3.1.2;
- Likewise, developers can specify Qosket specific ORB configuration and assemble QoS mechanisms into the component server or client ORB;
- Out-of-band provisioning and adaptation modules, such as contracts, system conditions, and callback objects can

be implemented as CCM components and be assembled into component servers.

While the use of CIAO to compose Qoskets into component assemblies simplifies retrofitting, a significant problem remains open: *component cross-cutting*. Qoskets are adept at separating concerns between systemic QOS properties and application logic, as well as implementing limited cross-cutting between a single client/object pair. Neither Qoskets nor CIAO currently provides the ability to cross-cut application components, however. Many QOS-related adaptations will need to modify the behavior of several components at once, possibly in a distributed way. Some form of dynamic aspect-oriented programming might be used to handle this, but this research is ongoing.

4 Model-Integrated Synthesis of QoS-enabled Component Middleware: A Powerful Approach to Resolving DRE Application QoS Provisioning Challenges

Section 1.2 outlined the key challenges associated with developing DRE applications with multidimensional QoS requirements. Sections 2 and 3 addressed some of these challenges by describing middleware mechanisms for provisioning and enforcing DRE application QoS requirements. These mechanisms do not, however, resolve the challenge of choosing, configuring, and assembling the appropriate set of semantically compatible QoS-enabled DRE middleware components, such as CIAO and Qoskets, tailored to the application's QoS requirements. Moreover, the mechanisms described earlier do not resolve the challenge posed by obsolescence of infrastructure technologies and its impact on long-term DRE system lifecycle costs.

This section explains how Model-Integrated Computing technologies can help to address the unresolved challenges outlined above. We first present an overview of Model-Integrated Computing and then show how it can be integrated with the QoS-enabled component middleware discussed in Section 3. Finally, we describe how we are developing a MIC toolsuite called CoSMIC (Component Synthesis using MIC) to address the aforementioned DRE application challenges.

4.1 Overview of Model-Integrated Computing

Model-Integrated Computing (MIC) [8] is a development paradigm that applies domain-specific modeling languages systematically to engineer computing systems ranging from small-scale real-time embedded systems to large-scale distributed enterprise applications. MIC provides rich, domain-specific modeling environments, including model analysis and model-based program synthesis tools [26]. In the MIC paradigm, application developers model an integrated, end-to-end view of the entire application, including the interdependencies of its components. Rather than focusing on a single, custom application, MIC models capture the essence of a class of applications. MIC also allows the modeling languages and environments themselves to be modeled by so-called *meta-models* [27], which help to synthesize domain-specific modeling languages that can capture the nuances of domains they are designed to model.

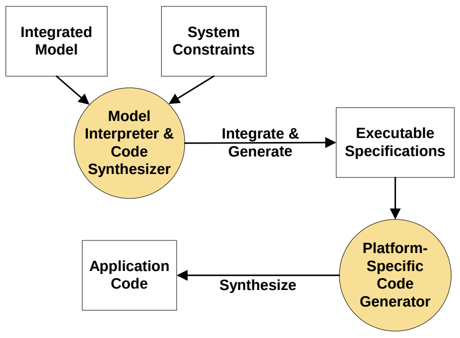
When implemented properly, MIC technologies help to:

- Free application developers from dependencies on particular software APIs, which ensures that the models can be used for a long time, even as existing software APIs become obsolete and replaced by newer ones.
- Provide correctness proofs for various algorithms by analyzing the models automatically and offering refinements to satisfy various constraints.
- Synthesize code that is highly dependable and robust since the tools can be built using provably correct technologies.
- Rapidly prototype new concepts and applications that can be modeled quickly using this paradigm, compared to the effort required to prototype them manually.
- Save companies and projects significant amounts of time and effort in design and maintenance, thereby also reducing application time-to-market.

Early computer-aided software engineering (CASE) technologies have evolved into sophisticated tools, such as *objectiF* and *in-Step* from MicroTool and *Paradigm Plus*, *VISION*, and *COOL* from Computer Associates. This class of products has evolved over the past two decades to alleviate complexities associated with developing software for enterprise applications. Their successes have added the Model-Integrated Computing paradigm to the familiar programming languages and language processing tool offerings used by previous generations of software developers. Popular examples of MIC tools being used today include the Generic Modeling Environment (GME) [26] and Ptolemy [28] (which are used primarily in the real-time and embedded domain) and UML/XML tools based on the OMG Model Driven Architecture (MDA) [29] (used primarily in the enterprise application domain thus far).

As shown in Figure 12, MIC uses a set of tools to

- Analyze the interdependent features of the system captured in a model and
- Determine the feasibility of supporting different non-functional system aspects, such as QoS requirements, in the context of the specified constraints.



Integrated Model
System Constraints
Model Interpreter & Code Synthesizer
Integrate & Generate
Executable Specifications
Platform-Specific Code Generator
Synthesize
Application Code

Figure 12: **The Model-Integrated Computing Process**

Another set of tools then translates models into executable specifications that capture the platform behavior, constraints, and interactions with the environment. These executable specifications can in turn be used to synthesize application software.

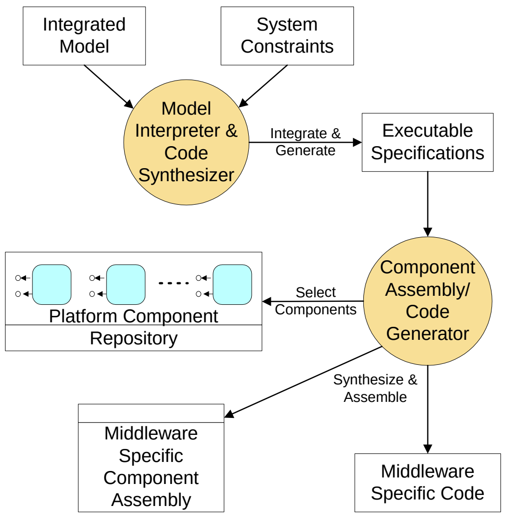
4.2 Combining Model-Integrated Computing and QoS-enabled Component Middleware

MIC and component middleware have evolved independently from different perspectives. Although each of these two paradigms have been successful independently, each also has its limitations, as discussed below:

Complexity due to heterogeneity. Conventional component middleware is developed using separate tools and interfaces written and optimized manually for each middleware technology, such as CORBA, J2EE, and .NET, and for each target deployment, such as various OS, network, and hardware configurations. Developing, assembling, validating, and evolving *all* this middleware manually is costly, time-consuming, tedious, and error-prone, particularly for runtime platform variations and complex application use-cases. This problem is exacerbated as more middleware, target platforms, and complex applications continue to emerge.

Lack of sophisticated modeling tools. Previous efforts at model-based development and code synthesis attempted by CASE tools generally failed to deliver on their potential for the following reasons [30]:

- They attempted to generate entire applications, including the infrastructure and the application logic, which often led to inefficient, bloated code that was hard to optimize, validate, evolve, or integrate with legacy code.
- Due to the lack of sophisticated domain-specific languages and associated modeling tools, it was hard to achieve *round-trip engineering*, *i.e.*, moving back and forth seamlessly between model representations and the synthesized code.



Integrated Model
System Constraints
Model Interpreter & Code Synthesizer
Integrate & Generate
Executable Specifications
Component Assembly/ Code Generator
Select Components
Platform Component Repository
Synthesize & Assemble
Middleware Specific Component Assembly
Middleware Specific Code

Figure 13: **Integrating Model-Integrated Computing and Component Middleware**

- Since CASE tools and modeling languages dealt primarily with a restricted set of platforms (such as mainframes) and legacy programming languages (such as COBOL) they did not adapt well to the distributed computing paradigm that arose from advances in PC and Internet technology and newer object-oriented programming languages, such as Java, C++, and C#.

The limitations with Model-Integrated Computing and component middleware outlined above can largely be overcome by integrating them as follows:

- Combining MIC with component middleware helps to overcome problems with earlier-generation CASE tools since it does not require the modeling tools to generate all the code. Instead, large portions of applications can be *composed* from reusable, prevalidated middleware components, as shown in Figure 13.
- Combining MIC and component middleware helps address environments where control logic and procedures change at rapid pace, by synthesizing and assembling newer extended components that implement the new procedures and processes.
- Combining component middleware with MIC helps to make middleware more flexible and robust by automating the configuration of many QoS-critical aspects, such as concurrency, distribution, resource reservation, security, and dependability. Moreover, MIC-synthesized code can help bridge the interoperability and portability problems between different middleware for which standard solutions do not yet exist.

- Combining component middleware with MIC helps to model the interfaces among various components in terms of standard middleware, rather than language-specific features or proprietary APIs.
- Changes to the underlying middleware or language mapping for one or many of the components modeled can be handled easily as long as they interoperate with other components. Interfacing with other components can be modeled as constraints that can be validated by model checkers.

Figure 14 illustrates seven points at which Model-Integrated Computing can be integrated into component middleware architectures, such as the integrated CIAO and Qoskets middleware suite, and applied to DRE applications. We describe each of these seven integration points below:

1. Configuring and deploying application services end-to-end. As discussed earlier in the explanation of Figure 4, developing complex DRE applications requires application developers to handle a variety of configuration and deployment challenges, such as

- Locating the appropriate existing services
- Partitioning and distributing application processes among component servers using the same middleware technologies and
- Provisioning the QoS required for each service that comprises an application end-to-end.

It is a daunting task to identify and deploy all these capabilities into an efficient, correct, and scalable end-to-end application configuration. For example, to maintain correctness and efficiency, services may change or migrate when the DRE application requirements change. Careful analysis is therefore required to partition collaborating services on distributed nodes so the information can be processed efficiently, dependably, and securely.

Integrating MIC and component middleware to deploy DRE application services end-to-end can help developers configure the right set of services into the right part of an application in the right way. MIC analysis tools can help determine the appropriate partitioning of functionality that should be deployed into various component servers throughout a network. For example, tools like *Matlab*, *Simulink*, *TimeWiz*, and *RapidRMA* allow DRE application developers to model and visualize their end-to-end application and QoS requirements. In particular, the *Simulink* tool allows DRE application developers to model, analyze, simulate, verify, and rapidly prototype applications.

2. Composing components into component servers. Integrating MIC with component middleware provides capabilities that help application developers to compose components into application servers by

- Selecting a set of suitable, semantically compatible components from reuse repositories.

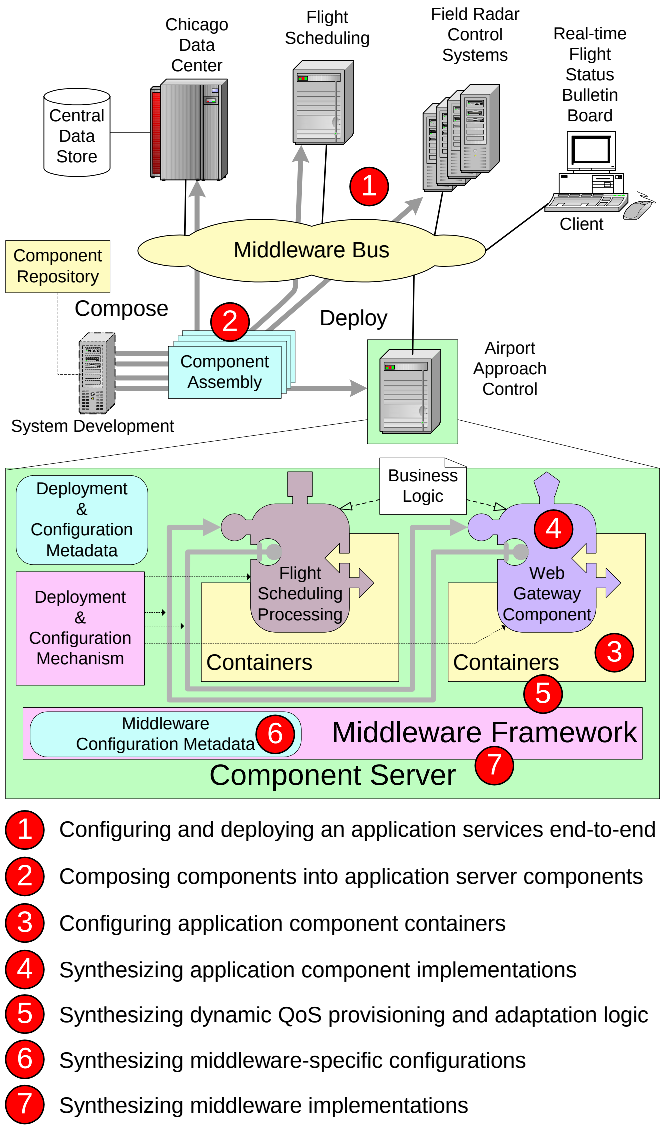

Chicago Data Center
Flight Scheduling
Field Radar Control Systems
Real-time Flight Status Bulletin Board
Central Data Store
Client
1
Middleware Bus
Component Repository
Compose
2
Deploy
Component Assembly
Airport Approach Control
System Development
Deployment & Configuration Metadata
Business Logic
4
Flight Scheduling Processing
Web Gateway Component
Deployment & Configuration Mechanism
Containers
Containers
3
5
Middleware Configuration Metadata
6
Middleware Framework
Component Server
7
1 Configuring and deploying an application services end-to-end
2 Composing components into application server components
3 Configuring application component containers
4 Synthesizing application component implementations
5 Synthesizing dynamic QoS provisioning and adaptation logic
6 Synthesizing middleware-specific configurations
7 Synthesizing middleware implementations

Figure 14: **Integrating Model-Integrated Computing with Component Middleware**

- Specifying the functionality required by new components to isolate the details of DRE systems that (1) operate in environments where DRE processes change periodically and/or (2) interface with third-party software associated with external systems.
- Determining the interconnections and interactions between components in metadata.
- Packaging the selected components and metadata into an assembly that can be deployed into the component server.

MIC tools, such as *Matlab* and *Simulink*, provide visual tools for composing DRE component servers.

3. Configuring application component containers. Application components use containers to interact with the component servers in which they are configured. As discussed in Section 2.3, containers manage many policies that distributed applications can use to fine-tune underlying component middleware behavior, such as its priority model, required service priority level, security, and other quality of service properties. Since DRE applications consist of many interacting components, their containers must be configured with consistent and compatible QoS policies.

Due to the number of policies and the intricate interactions among them, it is tedious and error-prone for a DRE application developer to *manually* specify and maintain component policies and semantic compatibility with policies of other components. MIC tools can help automate the validation and configuration of these container policies by allowing system designers to specify the required system properties as a set of models. Other MIC tools can then analyze the models and generate the necessary policies and ensure their consistency.

The Embedded Systems Modeling Language (ESML) [31] developed as part of the DARPA MoBIES program uses MIC technology to model the behavior of, and interactions between, avionics components. Moreover, the ESML model generators synthesize fault management and thread policies in component containers.

4. Synthesizing application component implementations. Developing complex DRE applications today involves programming new components that add application-specific functionality. Likewise, new components must be programmed to interact with external systems and sensors, such as a machine vision module controller, that are not internal to the application. Since these components involve substantial knowledge of application domain concepts, such as mechanical designs, manufacturing process, workflow planning, and hardware characteristics, it would be ideal if they could be developed in conjunction with mechanical engineers or domain experts, rather than programmed manually in isolation by software developers.

The shift toward high-level design languages and modeling tools is creating an opportunity for increased automation in generating and integrating application components. The goal is to bridge the gap between specification and implementation via sophisticated aspect weavers [22] and generator tools [26] that can synthesize platform-specific code customized for specific application properties, such as resilience to equipment failure, prioritized scheduling, and bounded worst-case execution under overload conditions.

The Adaptive Quality Modeling Environment (AQME) [32] developed as part of the DARPA PCES program uses MIC technology to provide a model-driven approach for QoS adaptation in DRE applications. In particular, AQME is used in conjunction with QuO/Qoskets to provide adaptive QoS policies for an unmanned aerial vehicle (UAV) real-time video distribution application. Section 5 describes this UAV application in more detail.

5. Synthesizing dynamic QoS provisioning and adaptation logic. Based on the overall system model and constraints, MIC tools may decide to plug in existing dynamic QoS provisioning and adaptation modules, using appropriate parameters. When none is readily available, the MIC tools can assist in creating the new behavior by synthesizing the logic using the Qosket QDL languages. The generated dynamic QoS behavior can then be used in system simulation dynamically to verify its validity. It can then be composed into the system as described above.

The AQME [32] modeling language mentioned at the end of integration point 4 above models the QuO/Qosket middleware by modeling system conditions and service objects. Moreover, interactions between the sender and receiver of the UAV video streaming applications, as well as parameters that instrument the middleware and application components, are modeled. We are building upon this work in current research exploring the modeling of higher level adaptation strategies and the constituent pieces comprising them. The dynamic QoS management strategies developed in AQME can be simulated extensively in the *Simulink* tool, before being applied to a real system.

In addition to applying modeling to the specification of higher level adaptation strategies, we are simultaneously applying AQME modeling techniques to solve more complex QoS resource management problems in the UAV application. For example, we are enhancing the UAV application by using AQME to model a QoS resource management strategy for the CPU reservation capability offered by the Timesys Linux real-time kernel [33]. A Timesys Linux CPU reservation guarantees that a thread will have a certain amount of CPU time, irrespective of the priorities of other threads in the system. We hope to show that the CPU reservation strategy developed using AQME modeling techniques will be more robust (*i.e.*, offer end-to-end QoS without priority inversion problems) than a strategy developed using ad hoc coding techniques and that AQME modeling can be used to model more comprehensive, cross-cutting adaptation strategies than are reasonable using hand-coding methods.

6. Synthesizing middleware-specific configurations. The infrastructure middleware technologies used by component middleware provide a wide range of policies and options to configure and tune their behavior. For example, CORBA ORBs often provide the following options and tuning parameters:

- Various types of transports and protocols
- Various levels of fault tolerance
- Middleware initialization options

- Efficiency of (de)marshaling event parameters
- Efficiency of demultiplexing incoming method calls
- Threading models and thread priority settings and
- Buffer sizes, flow control, and buffer overflow handling

Certain combinations of the options provided by the middleware may be semantically incompatible when used to achieve multiple QoS properties.

For example, a component middleware implementation could offer a range of security levels to the application. In the lowest security level, the middleware exchanges all the messages over an unsecure channel. The highest security level, in contrast, encrypts and decrypts messages exchanged through the channel using a set of dynamic keys. The same middleware could also provide an option to use zero-copy optimizations to minimize latency. A modeling tool could automatically detect the incompatibility of trying to compose the zero-copy optimization with the highest security level (which makes another copy of the data during encryption and decryption).

Advanced meta-programming techniques, such as adaptive and reflective middleware [34, 35, 36, 37] and aspect-oriented programming [22], are being developed to configure middleware options so they can be tailored for particular DRE application use cases.

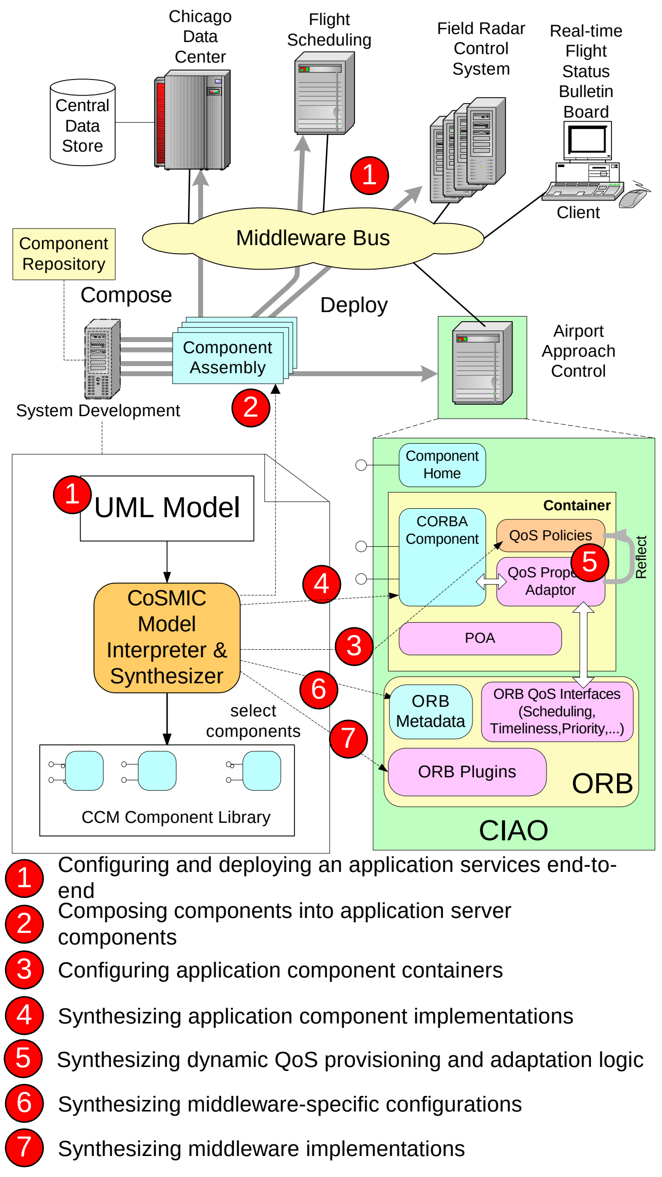
7. Synthesizing middleware implementations. Model-Integrated Computing can also be integrated with component middleware by using MIC tools to generate custom middleware implementations. This integration is a more aggressive use of modeling and synthesis than integration point 5 described above since it affects middleware *implementations*, rather than their configurations. Application integrators could use these capabilities to generate highly customized implementations of component middleware so that

- It only includes the features actually needed for a particular application and
- It is carefully fine-tuned to the characteristics of particular programming languages, operating systems, and networks.

The customizable middleware architectural framework *Quarterware* [38] is an example of this type of integration. Quarterware abstracts basic middleware functionality and allows application-specific specializations and extensions. The framework can generate core facilities of CORBA, RMI, and MPI. The framework-generated code is optimized for performance, which the authors demonstrate is comparable—and often better—than many commercially available middleware implementations.

4.3 Overview of CoSMIC

The Component Synthesis with Model-Integrated Computing (CoSMIC) project is a MIC toolset being developed by the

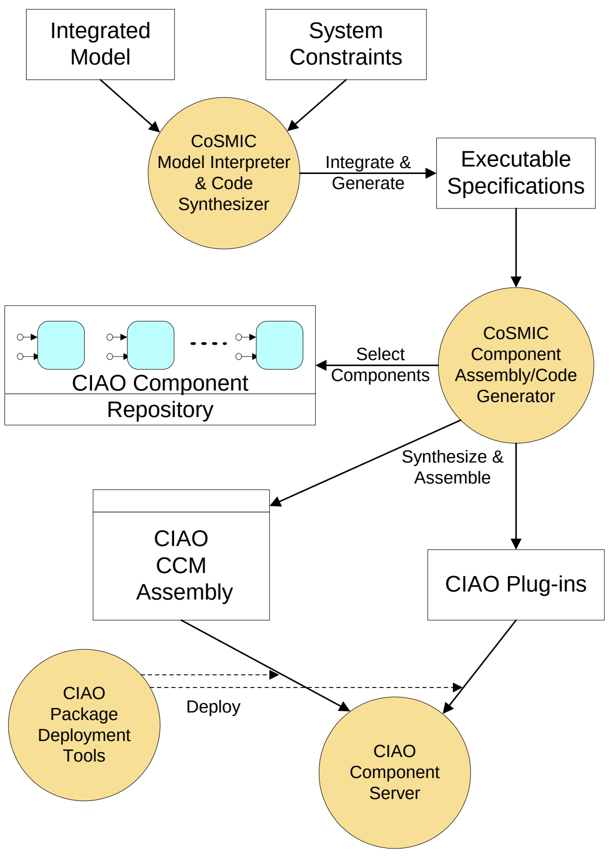


Chicago Data Center
Flight Scheduling
Field Radar Control System
Real-time Flight Status Bulletin Board
Central Data Store
Client
1
Middleware Bus
Component Repository
Compose
Deploy
Component Assembly
Airport Approach Control
System Development
2
Component Home
Container
1
UML Model
CORBA Component
QoS Policies
Reflect
QoS Property Adaptor
5
4
CoSMIC Model Interpreter & Synthesizer
3
POA
6
ORB Metadata
ORB QoS Interfaces (Scheduling, Timeliness,Priority,...)
select components
7
ORB Plugins
ORB
CCM Component Library
CIAO
1 Configuring and deploying an application services end-to-end
2 Composing components into application server components
3 Configuring application component containers
4 Synthesizing application component implementations
5 Synthesizing dynamic QoS provisioning and adaptation logic
6 Synthesizing middleware-specific configurations
7 Synthesizing middleware implementations

Figure 15: **Synthesizing CCM Middleware from MIC Tools**

Institute for Software Integrated Systems (ISIS) at Vanderbilt University to (1) *model and analyze* distributed real-time and embedded application functionality and QoS requirements and (2) *synthesize* CCM-specific deployment metadata required to deliver end-to-end QoS. Figure 15 illustrates the key elements in CoSMIC process.

The CoSMIC toolsuite provides modeling of DRE systems, their QoS requirements, and QoS adaptation policies used for DRE application QoS management. The component behavior, their interactions, and QoS requirements are modeled using a language similar to ESML [31]. Whereas ESML enables modeling a proprietary avionics component middleware,



Integrated Model
System Constraints
CoSMIC Model Interpreter & Code Synthesizer
Integrate & Generate
Executable Specifications
CIAO Component Repository
Select Components
CoSMIC Component Assembly/Code Generator
Synthesize & Assemble
CIAO CCM Assembly
CIAO Plug-ins
CIAO Package Deployment Tools
Deploy
CIAO Component Server

Figure 16: **Interaction betwwen CoSMIC and CIAO**

CoSMIC enables modeling the standards-based CCM components. Moreover, CoSMIC provides modeling languages to model the adaptive QoS behavior supported by QuO/Qoskets.

The CoSMIC project is developing synthesis tools targeted at the CIAO component middleware described in Section 3.1.2. CIAO abstracts component QoS requirements into metadata that can be specified in a component assembly after a component has been implemented. Decoupling QoS requirements from component implementations greatly simplifies the conversion and validation of an application model with multiple QoS requirements into CCM deployment of DRE applications.

The remainder of this section describes how we are combining the CoSMIC design tools and procedures with the CIAO component middleware platform to address key challenges faced by the developers of DRE applications. Figure 16 illustrates the interfaction between CoSMIC and CIAO.

Challenge 1: Satisfying Multiple Quality of Service (QoS) Requirements Simultaneously

Problem. DRE applications demand stringent QoS support from their middleware. For example, DRE applications such as controller for high-speed surface mount component pick-and-place machines require real-time predictability and performance guarantees. Due to (1) the complexity of these QoS requirements, (2) the heterogeneity of the environments in which they are deployed, and (3) the existing legacy systems and data, it is infeasible to develop a single-vendor, end-to-end solution that can address all these challenges. Instead, integrating highly configurable, flexible, and optimized COTS components from several different providers based on standard component middleware enables developers to assemble and deploy these systems rapidly and robustly. Ensuring application QoS requirements end-to-end, however, can be complicated.

Solution. A benefit of MIC is its ability to employ complex modeling tools that can check for certain properties of the implementation, *e.g.*, check the correctness of an algorithm or ensure that a series of constraints are enforced.

There does not yet exist standards-based MIC technology that adequately addresses a broad spectrum of DRE application QoS issues. In particular, the integration of static and dynamic Qos provisioning mechanisms, such as priority propagation, resource allocations, dependability, predictability, and adaptation that are crucial to DRE applications are not yet addressed.

The tools we are developing in CoSMIC are therefore designed to model and analyze both the application functionality and its end-to-end QoS requirements. With CIAO's support for QoS-enabled, reusable CCM components, it is possible to

- Model the QoS requirements of applications using UML
- Associate the model with different static and dynamic QoS profiles
- Simulate and analyze dynamic behaviors and
- Synthesize the QoS-enabled application functionality in component assemblies.

Figure 15 illustrates how CoSMIC can be used to synthesize and assemble QoS-enabled, CCM middleware for DRE applications. This synthesis uses the following iterative process to assemble and deploy QoS-enabled distributed applications:

1. **Model the overall application** using CoSMIC visual modeling tools and specify the application's QoS requirements as constraints. This step defines and partitions the functionality and QoS requirements demanded by each application module based on the overall model of the application, as described by integration point 1 of Figure 15
2. **Compose component servers** using CoSMIC component server composition tools to combine component assemblies by mixing and matching existing off-the-shelf

components and partitioning or defining the functionality of new components, as needed, as shown in Point 2 of Figure 15. The metadata in a component assembly also contain QoS requirements for each components that the composition tools derived from the model.

3. **Model and synthesize components**—If new component implementations are needed from the previous step, each can be modeled by using CoSMIC's modeling tool. CoSMIC's component implementation synthesizer will generate the actual implementations based on the models, as indicated by integration point 4 of Figure 15.
4. **Validate and simulate applications** via's CoSMIC tools that check whether an application composition implements its model definitions correctly.
5. **Deploy the resulting system for testing and tuning** via tools that fine-tune CIAO's QoS requirements for assemblies. Later iterations of this process can use these adjustments as feedback to improve the overall system model.

Challenge 2: Addressing Accidental Complexities in Integrating Software Systems

Problem. QoS-enabled component middleware, such as CIAO, provides libraries of reusable, configurable components that can be used to assemble and deploy QoS-aware DRE applications. However, a naive approach to assemble and configure these components can yield components with incompatible, non-interoperable QoS requirements, thereby increasing accidental complexities. Manual assembling components and configuring their QoS requirements are tedious and error-prone, which adversely affects application lifecycle costs and time-to-market. Moreover, to ensure these requirements are met end-to-end across a DRE application, component servers often explicitly require complex policies and customized middleware plugins. Manually specifying and configuring these policies makes the development process even more vexing.

Solution. The iterative process described in the solution for Challenge 2 above helps DRE application developers manage the accidental complexity of assembling components by providing rich semantics in models and automatically propagating these semantics into assemblies through metadata. There is, however, a need to ensure that the component servers and the underlying middleware are configured properly to satisfy the QoS requirements demanded by the installed components.

The CCM specification does not yet address how to associate component QoS requirements with a component deployment. Our CCM implementation (CIAO) therefore supports the configuration of certain component QoS properties via the component deployment metadata shown by integration point 2 of Figure 15. Since we providing component QoS management services through containers in our CCM implementation[39], the synthesizing tools will also generate container configurations in a component assembly, as depicted in Point 3 of Figure 15.

To support QoS requirements that were not foreseen by the component middleware implementation, CoSMIC can also synthesize middleware modules that CIAO uses to customize its behavior to support non-native QoS supports required by other systems. CIAO's deployment framework then uses these customized modules to configure component servers before deploying the components, as shown by integration point 6 of Figure 15. The automation of semantic propagation described here ensures that all component servers consisting an integrated DRE application perform their work as specified in the overall model, without undue programmer intervention.

5 Adaptive Video Streaming: An Unmanned Air Vehicle (UAV) Prototype

5.1 Overview

This section presents a case study of an unmanned aerial vehicle (UAV) real-time video distribution prototype, in which static and dynamic QoS provisioning must be applied to ensure an MPEG video flow can meet its mission QoS requirements, such as timeliness, jitter, and image resolution. We discuss behaviors that adapt to restrictions in processing power and network bandwidth, *i.e.*, reduction of the video flow volume by dropping frames and bandwidth reservation to ensure a desired level of network bandwidth. We have developed this prototype UAV application by applying the component middleware technologies discussed in earlier sections, as well as the TAO Audio/Video (A/V) Streaming Service [40]. This resulting application establishes and adaptively controls video transmission from a live camera via a distribution process to viewers on computer displays.

Figure 17 illustrates the architecture of the prototype. The

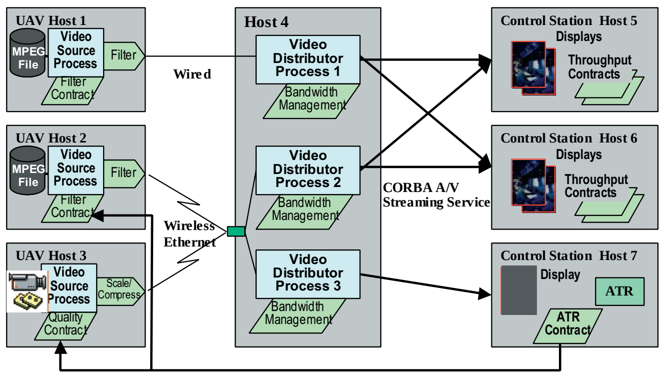


Figure 17: **Architecture of the UAV Prototype**

prototype uses a three-stage pipeline, with simulated UAVs or live UAV surrogates (such as airships with mounted cameras) sending video to processes (distributors) that distribute the video to the proper control stations. The UAVs in our prototypes are simulated by processes that are capturing video from live camera feeds and by processes that read MPEG video from a file. In addition, the prototype uses both wired and wireless Ethernet connections to simulate the data links from the UAVs to the distributor host. The wireless links from the second and third UAV surrogates contend for the same wireless Ethernet connection and provide a forum for experimenting with wireless video adaptation strategies. The wired Ethernet connection provides a higher bandwidth connection simulating current and emerging higher-capacity wireless transports.

The video distributor processes send the video streams to control stations on a land- or ship-based network. The control stations include video display processes and other video processing applications, such as automatic target recognition (ATR), each with their own mission requirements.

5.2 Benefits of QoS Provisioning

The current UAV application can manage QoS by engaging application adaptive behavior, such as frame dropping and setting network bandwidth reservations (RSVP). We are also adding network QoS management capability by prioritizing data streams at the network level (Diffserv). To test the effectiveness of the middleware controlled adaptation in the UAV application we performed the following experiment, which consists of following three runs:

1. A control run, with no adaptation
2. A second run, where adaptation is implemented by frame dropping and
3. A third run, which utilized both frame dropped and RSVP bandwidth management.

All runs were configured with (1) the sender and distributor on the same Pentium III 933 MHz processor and 512 MB of RAM and (2) the receiver on a separate laptop, with a Pentium II 200 MHz processor and 144 MB of RAM, all running Linux, with a 10 Mbps link between them. We started the video running and after 60 seconds we applied a load to the network link for 60 seconds, then removed the load. After another 180 seconds (300 seconds in all) the experiment terminated. The data collector recorded each MPEG I frame (2 per second, 600 in all) that was sent from the sender, and each that was received at the receiver, and the time elapsed from send to receive.

The results of each test run are shown in Figure 18 and described below:

Test run 1, which as the control run without any adaptation, lost 119 of the 121 I frames sent while the system was under load, *i.e.*, only 481 of the 600 I frames sent made it through. The average delay of the frames that made it through was 56.58 ms, with a median delay of 55 ms, a minimum delay of 38 ms, and a maximum delay of 121 ms. The average delay of the frames sent through when the system was not under load was 56.33 ms, with a median of 55 ms, a minimum of 38 ms, and a maximum of 67 ms. 1.65 percent of the I frames sent made it through when the system was under load (2 out of 121), with 98.35% of the I frames being lost by the UDP transport. The average delay of the two that did make it through under load was 115.5 ms.

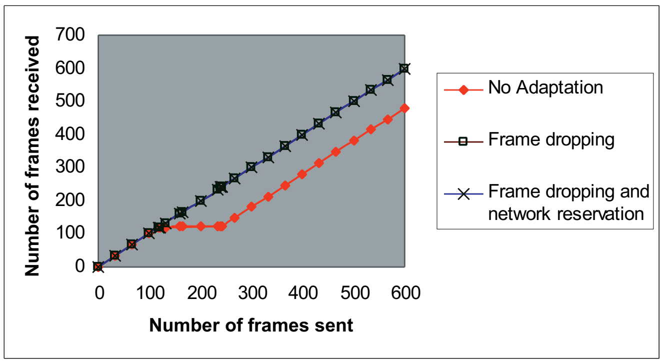


Figure 18: **QuO adaptation ensured successful delivery of all video under load**

Test run 2, with frame dropping as its only adaptation, got all 600 of its I frames through despite the load on the system. The average delay of all the frames was 70.01 ms, with a median delay of 57 ms, a minimum delay of 50 ms, and a maximum delay of 143 ms. The average delay of the frames sent through when the system was not under load was 57.01 ms, with a median of 55 ms, a minimum of 50 ms, and a maximum of 68 ms. 100 percent of the I frames sent made it through when the system was under load (120 out of 120), with 0% of the I frames being lost by the UDP transport. The average delay of the frames delivered while the system was under load was 122.15 ms.

Test run 3, with adaptation using both frame dropping and network reservation (RSVP), also got all 600 of its I frames through despite the load on the system. The average delay of all the frames was 64.2 ms, with a median delay of 59 ms, a minimum delay of 52 ms, and a maximum delay of 106 ms. The average delay of the frames sent through when the system was not under load was 58.1 ms, with a median of 56 ms, a minimum of 52 ms, and a maximum of 71 ms. 100 percent of the I frames sent made it through when the system was under load (120 out of 120), with 0% of the I frames being lost by the UDP transport. The average delay of the frames delivered while the system was under load was significantly lower than the other two runs, 88.5 ms.

Figure 18 illustrates the improvements afforded by the adaptation under load. The test runs that included QuO adaptation were able to recover from the load imposed on the system to keep the video flowing and not lose any important (*i.e.*, I) frames. In contrast, the test runs where the video stream did not have adaptive control lost nearly all the video sent during the time when load was imposed on the system. The combination of frame filtering and resource reservation significantly reduced the latency of the video delivery.

6 Related Work

This section first reviews previous research on Model-Integrated Computing and describes how modeling tools are being used to model and provision QoS requirements. It then reviews other research on QoS provisioning mechanisms using a taxonomy depicted in Figure 19. One dimension depicted in

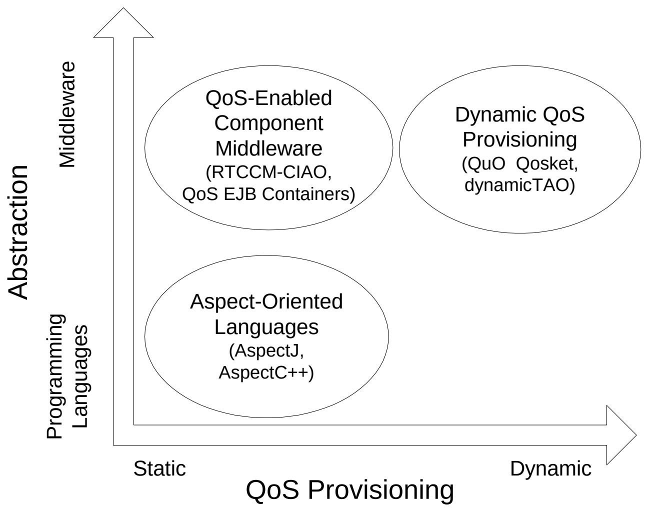

Abstraction
Middleware
Programming Languages
QoS-Enabled Component Middleware (RTCCM-CIAO, QoS EJB Containers)
Dynamic QoS Provisioning (QuO Qosket, dynamicTAO)
Aspect-Oriented Languages (AspectJ, AspectC++)
Static
Dynamic
QoS Provisioning

Figure 19: **Taxonomy of QoS Provisioning Enabling Mechanisms**

Figure 19 is when QoS provisioning is performed, *i.e.*, static versus dynamic QoS provisioning, as described in Section 1. Some enabling mechanisms allow static QoS provisioning before the startup of a system, whereas and others provide abstractions to define dynamic QoS provisioning behaviors during runtime based on resources available at the time.

Another dimension depicted in Figure 19 is the level of abstraction. Both middleware-based approaches shown in the figure, *i.e.*, CIAO and BBN's QuO Qoskets, offer higher levels of abstraction for QoS provisioning specification and modeling. Conversely, the programming language-based approach offers meta-programming mechanisms for injecting QoS provisioning behaviors. We will review previous research in the area of QoS provisioning mechanisms along these two dimensions.

Model-based software development. Our research extends earlier work on Model-Integrated Computing [8, 41, 42, 43] to model and synthesize component middleware code for DRE applications. The MIC infrastructure provides a unified software architecture and framework for creating a Model-Integrated Program Synthesis (MIPS) environment [26], as described in Section 4.

Examples of MIC technology used today include GME [26] and Ptolemy [28] – used primarily in the real-time and embedded domain, and MDA [29] based on UML [44] and XML [45] – used primarily in the business domain. Our work on CoSMIC combines the GME tool and UML modeling language to model and synthesize component middleware for use in provisioning collaborative DRE applications. In particular, we are enhancing the GME tool to produce meta-models for DRE applications, as well as developing and validating new UML profiles to support DRE applications.

As part of an ongoing research collaboration between Vanderbilt University, University of Utah, and BBN Technologies, work is being done to apply GME MIC techniques to model an effective resource management strategy for CPU resources on the Timesys Linux real-time operating system. Timesys Linux allow an application to specify CPU reservations for an executing thread which guarantee that a thread will have a certain amount of CPU time, regardless of the priorities of other threads in the system. By using GME modeling to develop the QoS management strategy, it will be easier to simulate and verify the strategy for its correctness and ability to meet end-to-end QoS requirements for CPU processing.

Dynamic QoS Provisioning. In their *dynamicTAO* project, Kon and Campbell [34] apply reflective middleware techniques to extend TAO to reconfigure the ORB at runtime by dynamically linking selected modules, according to the features required by the applications. Their work falls into the same category as *Qoskets* as shown in Figure 19 in that both provide the mechanisms to realizing QoS provision in the middleware level. Qoskets offers a more comprehensive QoS provisioning abstraction, however, whereas Kon and Campbell's work concentrates on configuring the middleware capability.

Moreover, although Kon and Campbell's work can also provide QoS adaptation behavior by dynamically (re)configuring middleware framework, their research may not be suitable for DRE applications since dynamic loading and unloading ORB components can incur non-deterministic overhead and prevent the ORB from meeting application deadlines. Our work on the Component-Integrated ACE ORB (CIAO) relies upon MIC tools to analyze the required ORB components and their configurations. This approach ensures the ORB in an application server contains only the required components, without compromising end-to-end predictability.

QoS-enabled Component Middleware. Middleware can apply the Quality Connector pattern [37] to apply meta-programming techniques [39] and specify the QoS behaviors and the configuring the supporting mechanisms for these QoS behaviors. The container architecture in component-based middleware frameworks provide the vehicle for applying meta-programming techniques that provide QoS assurance control in component middleware, as previously identified in [39]. Containers can also help applying Aspect-Oriented Software Development (AOSD) [22] techniques to plug in different non-functional behaviors [46]. These projects are similar to CIAO in that they provide the mechanism to inject "aspects" into applications statically at the middleware level.

de Miguel further extended the work on QoS-enabled containers by extending an QoS EJB container interface to support `QoSContext` interface which allow exchanging QoS related information with component instances [47]. To take advantage of the QoS-container, however, a component must implement `QoSBean` and `QoSNegotiation` interfaces. This requirement, however, adds unnecessary dependency on component implementations. Section 2.4 presents the limitations of implementing QoS behavior logic in component implementations.

Aspect-Oriented Programming Languages. Aspect-Oriented programming (AOP) [22] languages provide language-level abstraction to weave different aspects that cross-cut multiple layers of a system. Example of AOP languages includes AspectJ [48] and AspectC++ [49]. Similar to AOPs, CIAO supports injection of aspects into systems at middleware level using meta-programming technique. Both CIAO and AOP weave in aspects statically, *i.e.*, before programming execution and neither define an abstraction for dynamic QoS provisioning behaviors.

7 Concluding Remarks

Distributed real-time and embedded (DRE) applications possess stringent support for QoS properties such as predictability, latency, and dependability. To meet these requirements, DRE applications have historically been custom-programmed to implement their QoS provisioning needs, making them expensive to build and maintain that they cannot adapt readily to meet new functional or different QoS provisioning strategy, hardware/software technology innovations, or market opportunities. This approach is increasingly infeasible, however, since the tight coupling between custom DRE software modules increases the time and effort required to develop and evolve DRE software. Moreover, QoS provisioning cross-cuts multiple layers in applications and requires end-to-end enforcement which make DRE applications even harder to develop, maintain, and adapt.

One way to address these coupling issue is by refactoring common application logic into *object-oriented application frameworks* [50]. This solution has limitations, however, since application objects can still interact directly with each other, which encourages tight coupling. Moreover, framework-specific bookkeeping code is also required within the applications to manage the framework, which can tightly couple applications to the framework they are developed upon. It is therefore non-trivial to reuse application objects and port them to different frameworks.

Component middleware [16] has emerged as a promising solution to many limitations with object-oriented application frameworks. This type of middleware consists of reusable software artifacts that can be distributed or collocated throughout a network. Existing component middleware, however, does not address DRE application's QoS provisioning needs as they often spread beyond component boundary. A QoS-enabled middleware is necessary to separate the QoS provisioning concerns from application functional concerns.

This paper describes how the standard CCM specification is being augmented by the CIAO middleware to support static QoS provisioning that pre-allocates resources for DRE application. We also describe how BBN's QuO Qoskets middleware framework provides powerful abstractions that help define and implement reusable dynamic QoS provisioning behaviors. By combining QuO Qoskets and CIAO, we are providing an integrated QoS provisioning solution for DRE applications.

When augmented with Model-Integrated Computing tools, such as CoSMIC, QoS-enabled component middleware and applications can be provisioned even more effectively.

References

[1] Aniruddha Gokhale, Douglas C. Schmidt, Balachandra Natarajan, and Nanbor Wang, "Applying Model-Integrated Computing to Component Middleware and Enterprise Applications," *The Communications of the ACM Special Issue on Enterprise Components, Service and Business Rules*, vol. 45, no. 10, Oct. 2002.

[2] Alan Burns and Andy Wellings, *Real-Time Systems and Programming Languages, 3rd Edition*, Addison Wesley Longmain, Mar. 2001.

[3] Object Management Group, *CORBA 3.0 New Components Chapters*, OMG TC Document ptc/2001-11-03 edition, Nov. 2001.

[4] Sun Microsystems, "Java^{TM} 2 Platform Enterprise Edition," http://java.sun.com/j2ee/index.html, 2001.

[5] Don Box, *Essential COM*, Addison-Wesley, Reading, MA, 1998.

[6] Bruce Trask, "A Case Study on the Application of CORBA Products and Concepts to an Actual Real-Time Embedded System," in *OMG's First Workshop On Real-Time & Embedded Distributed Object Computing*, Washington, D.C., July 2000, Object Management Group.

[7] Douglas C. Schmidt, Rick Schantz, Mike Masters, Joseph Cross, David Sharp, and Lou DiPalma, "Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems," *CrossTalk*, Nov. 2001.

[8] Janos Sztipanovits and Gabor Karsai, "Model-Integrated Computing," *IEEE Computer*, vol. 30, no. 4, pp. 110–112, Apr. 1997.

[9] John A. Zinky, David E. Bakken, and Richard Schantz, “Architectural Support for Quality of Service for CORBA Objects,” *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 1–20, 1997.

[10] Richard E. Schantz and Douglas C. Schmidt, “Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications,” in *Encyclopedia of Software Engineering*, John Marciniak and George Telecki, Eds. Wiley & Sons, New York, 2002.

[11] Ward Rosenberry, David Kenney, and Gerry Fischer, *Understanding DCE*, O’Reilly and Associates, Inc., 1992.

[12] IBM, “MQSeries Family,” www-4.ibm.com/software/ts/mqseries/, 1999.

[13] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 3.0 edition, June 2002.

[14] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.6.1 edition, May 2002.

[15] Nanbor Wang, Douglas C. Schmidt, and Carlos O’Ryan, “An Overview of the CORBA Component Model,” in *Component-Based Software Engineering*, George Heineman and Bill Councill, Eds. Addison-Wesley, Reading, Massachusetts, 2000.

[16] George T. Heineman and Bill T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, Reading, Massachusetts, 2001.

[17] Arthur van Hoff, Hadi Partovi, and Tom Thai, “The Open Software Description Format (OSD),” http://www.w3c.org/TR/NOTE-OSD.html.

[18] Floyd Marinescu and Ed Roman, *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*, John Wiley & Sons, New York, 2002.

[19] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, “The Design and Performance of Real-Time Object Request Brokers,” *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

[20] 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, 2000.

[21] Nanbor Wang, Douglas C. Schmidt, Ossama Othman, and Kirthika Parameswaran, “Evaluating Meta-Programming Mechanisms for ORB Middleware,” *IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies*, vol. 39, no. 10, Oct. 2001.

[22] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin, “Aspect-Oriented Programming,” in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[23] Richard E. Schantz, John A. Zinky, David A. Karr, David E. Bakken, James Megquier, and Joseph P. Loyall, “An object-level gateway supporting integrated-property quality of service,” in *Proceedings of The 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 99)*, May 1999.

[24] Joseph P. Loyall, David E. Bakken, Richard E. Schantz, John A. Zinky, David Karr, Rodrigo Vanegas, and Ken R. Anderson, “QoS Aspect Languages and Their Runtime Integration,” *Proceedings of the Fourth Workshop on Languages, Compilers and Runtime Syste,s for Sclable Components*, May 1998.

[25] Rodrigo Vanegas, John A. Zinky, Joseph P. Loyall, David Karr, Richard E. Schantz, and David E. Bakken, “QuO’s Runtime Support for Quality of Service in Distributed Objects,” *Proceedings of Middleware 98, the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing*, September 1998.

[26] Akos Ledeczi, Arpad Bakay, Miklos Maroti, Peter Volgysei, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai, “Composing Domain-Specific Design Environments,” *IEEE Computer*, Nov. 2001.

[27] Jonathan M. Sprinkle, Gabor Karsai, Akos Ledeczi, and Greg G. Nordstrom, “The New Metamodeling Generation,” in *IEEE Engineering of Computer Based Systems*, Washington, DC, Apr. 2001, IEEE.

[28] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” *International Journal of Computer Simulation, Special Issue on Simulation Software Development Component Development Strategies*, vol. 4, Apr. 1994.

[29] Object Management Group, *Model Driven Architecture (MDA)*, OMG Document ormsc/2001-07-01 edition, July 2001.

[30] Paul Allen, “Model Driven Architecture,” *Component Development Strategies*, vol. 12, no. 1, Jan. 2002.

[31] Gabor Karsai, Sandeep Neema, Arpad Bakay, Akos Ledeczi, Feng Shi, and Aniruddha Gokhale, “A Model-based Front-end to ACE/TAO: The Embedded System Modeling Language,” in *Proceedings of the Second Annual TAO Workshop*, Arlington, VA, July 2002.

[32] Sandeep Neema, Ted Bapty, Jeff Gray, and Aniruddha Gokhale, “Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems,” in *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE’02)*, Pittsburgh, PA, Oct. 2002.

[33] Timesys, “Predictable Performance for Dynamic Load and Overload,” `http://www.timesys.com/prodserv/whitepaper/Predictable_Performance_1_0.%pdf`, 2002.

[34] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell, “The Case for Reflective Middleware,” *Communications of the ACM*, vol. 45, no. 6, pp. 33–38, June 2002.

[35] 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*, London, 1998, pp. 191–206, Springer-Verlag.

[36] 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.

[37] Joseph K. Cross and Douglas C. Schmidt, “Applying the Quality Connector Pattern to Optimize Distributed Real-time and Embedded Middleware,” in *Patterns and Skeletons for Distributed and Parallel Computing*, Fethi Rabhi and Sergei Gorlatch, Eds. Springer Verlag, 2002.

[38] Roy Campbell, Ashish Singhai, and Aamod Sane, “Quarterware for Middleware,” in *Proceedings of ICDCS 98*. IEEE, 1998.

[39] Nanbor Wang, Douglas C. Schmidt, Michael Kircher, and Kirthika Parameswaran, “Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications,” *IEEE Distributed Systems Online*, vol. 2, no. 5, July 2001.

[40] Sumedh Mungee, Nagarajan Surendran, Yamuna Krishnamurthy, and Douglas C. Schmidt, “The Design and Performance of a CORBA Audio/Video Streaming Service,” in *Design and Management of Multimedia Information Systems: Opportunities and Challenges*, Mahbubur Syed, Ed. Idea Group Publishing, Hershey, PA, 2000.

[41] David Harel and Eran Gery, “Executable Object Modeling with Statecharts,” in *Proceedings of the 18th International Conference on Software Engineering*. 1996, pp. 246–257, IEEE Computer Society Press.

[42] Man Lin, “Synthesis of Control Software in a Layered Architecture from Hybrid Automata,” in *HSCC*, 1999, pp. 152–164.

[43] Jeffery Gray, Ted Bapty, and Sandeep Neema, “Handling Crosscutting Constraints in Domain-Specific Modeling,” *Communications of the ACM*, pp. 87–93, Oct. 2001.

[44] Object Management Group, *Unified Modeling Language (UML) v1.4*, OMG Document formal/2001-09-67 edition, Sept. 2001.

[45] W3C Architecture Domain, “Extensible Markup Language (XML),” `http://www.w3c.org/XML`.

[46] Denis Conan, Erik Putrycz, Nicolas Farcet, and Miguel DeMiguel, “Integration of Non-Functional Properties in Containers,” *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP)*, 2001.

[47] Miguel A. de Miguel, “QoS-Aware Component Frameworks,” in *The 10^{th} International Workshop on Quality of Service (IWQoS 2002)*, Miami Beach, Florida, May 2002.

[48] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, “An overview of AspectJ,” *Lecture Notes in Computer Science*, vol. 2072, pp. 327–355, 2001.

[49] Olaf Spinczyk and Andreas Gal and Wolfgang Schröder-Preikschat, “AspectC++: An Aspect-Oriented Extension to C++,” in *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Feb. 2002.

[50] Ralph Johnson, “Frameworks = Patterns + Components,” *Communications of the ACM*, vol. 40, no. 10, Oct. 1997.