Integrating Publisher/Subscriber Services in Component Middleware for Distributed Real-time and Embedded Systems *

George T. Edwards
Institute for Software Integrated Systems
Vanderbilt University
Box 1829, Station B
Nashville, TN 37235
g.edwards@vanderbilt.edu

Douglas C. Schmidt
Institute for Software Integrated Systems
Vanderbilt University
Box 1829, Station B
Nashville, TN 37235
d.schmidt@vanderbilt.edu

Aniruddha Gokhale
Institute for Software Integrated Systems
Vanderbilt University
Box 1829, Station B
Nashville, TN 37235
a.gokhale@vanderbilt.edu

ABSTRACT

Although component-based software development has widespread acceptance in the enterprise business and desktop application domains, developers of distributed real-time and embedded (DRE) systems have encountered limitations with the available component middleware platforms, such as the CORBA Component Model (CCM) and the Java 2 Enterprise Edition (J2EE). These limitations often preclude developers of DRE systems from fully exploiting the benefits of component software. In particular, component middleware platforms lack standards-based publisher/subscriber communication mechanisms that support key quality-of-service (QoS) requirements, such as low latency, bounded jitter, and end-to-end operation priority propagation. QoS-enabled publisher/subscriber services are available in object middleware platforms, such as Real-time CORBA, but such services have not been integrated into component middleware due to a number of development and configuration challenges.

This paper provides three contributions to the integration of publisher/subscriber services in component middleware. First, we outline key challenges associated with integrating publisher/subscriber services into component middleware. Second, we describe a methodology for resolving these challenges based on software patterns. Third, we describe a pattern-oriented component middleware platform that we have developed to integrate publisher/subscriber services into component middleware applications.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time Systems:Event-based Systems; D.2.13 [**Software Engineering**]: Reusable Software:Patterns

Keywords

Real-time Event Service, Component Middleware, CORBA Component Model, Model-based Systems.

1. INTRODUCTION

To reduce the complexity of designing robust, efficient, and scalable distributed real-time and embedded (DRE) software systems, developers increasingly rely on *middleware* [1], which is software that resides between applications and the lower-level run-time infrastructure, such as operating systems, network protocol stacks, and hardware. Middleware isolates DRE applications from lower-level infrastructure complexities, such as heterogeneous platforms and error-prone network programming mechanisms. It also enforces essential end-to-end quality of service (QoS) properties, such as low latency and bounded jitter; fault propagation/recovery across distribution boundaries; authentication and authorization; and weight, power consumption, and memory footprint constraints.

Over the past decade, middleware has evolved to support the creation of applications via composition of reusable and flexible software *components* [2]. Components are implementation/integration units with precisely-defined interfaces that can be installed in application server run-time environments. Examples of commercial-off-the-shelf (COTS) component middleware include the CORBA Component Model (CCM) [3], J2EE [4], and .NET [5].

Component middleware generally supports two models for component interaction: (1) a *request-response* communication model, in which a component invokes a point-to-point operation on another component and (2) an *event-based* communication model, in which a component transmits arbitrarily-defined, strongly-typed messages, called *events*, to other components. Event-based communication models are particularly relevant for large-scale DRE systems (such as avionics mission computing [6], distributed audio/video processing [7], and distributed interactive simulations [8]) because they help reduce software dependencies and enhance system composability and evolution. In particular, the *publisher/subscriber* architecture [9] of event-based communication allows application components to communicate anonymously

*This work was sponsored in part by grants from NSF ITR CCR-0312859, Siemens, and DARPA/AFRL contract #F33615-03-C-4112

ACMSE '04, April 2-3, 2004, Huntsville, Alabama, USA.Copyright 2004 ACM 1-58113-870-9/04/04...$5.00.

and asynchronously [10]. The publisher/subscriber communication model defines the following three software roles:

- **Publishers** generate events to be transmitted. Depending on the architecture design and implementation, publishers may need to describe the events they generate *a priori*.
- **Subscribers** receive events via hook operations. Subscribers also may need to declare the events they receive *a priori*.
- **Event channels** accept events from publishers and deliver events to subscribers. Event channels perform event filtering and routing, QoS enforcement, and fault management. In distributed systems, event channels propagate events across distribution domains to remote subscribers.

The publisher/subscriber design is an especially powerful architecture for event-based communication because it provides (1) *anonymity* by decoupling event publishers and subscribers and (2) *asynchrony* by automatically notifying subscribers when a specified event is generated.

Figure 1 illustrates the relationships and information flow between these three types of components.

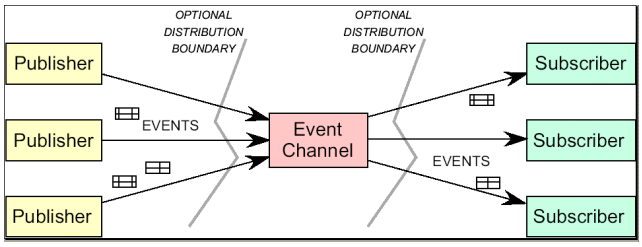

Figure 1: Relationships Between Components in a Publisher/Subscriber Architecture

QoS-enabled component middleware platforms leverage the benefits of component-centric software development while simultaneously preserving the optimization patterns and principles of distributed object computing middleware. Before DRE application developers can derive benefits from QoS-enabled components, however, common middleware services must be integrated with component middleware in a manner that minimizes the complexity associated with configuration and deployment. This paper describes a novel scheme that employs patterns to integrate a family of publisher/subscriber services within QoS-enabled CORBA component middleware.

Our previous work on publisher/subscriber architectures focused on the patterns and performance optimizations of event channels in the context of real-time object middleware [11], specifically a *highly scalable* [8] and *real-time* [12] CORBA Event Service [13]. This paper extends our previous work by describing how patterns can be applied to simplify the integration of publisher/subscriber services in QoS-enabled component middleware.

We have developed a QoS-enabled CCM implementation, the *Component-Integrated ACE ORB* (CIAO) [14], which supports the development, assembly, configuration, and deployment of component-based applications. This paper focuses on the implementation of a framework that supports application component access to publisher/subscriber services in CIAO.

The remainder of this paper is organized as follows: Section 2 briefly describes the structure of the CORBA Component Model (CCM); Section 3 details the key challenges associated with implementing and configuring publisher/subscriber services in component middleware; Section 4 describes a pattern-oriented component middleware framework we developed to address these challenges; and Section 5 presents concluding remarks and outlines future work.

2. OVERVIEW OF CCM

The CORBA Component Model (CCM) forms a key part of the CORBA 3.0 standard [15]. CCM is designed to address the limitations with earlier versions of CORBA 2.x middleware that supported a distributed object computing (DOC) model [16]. Figure 2 depicts the layered architecture of the CCM model. The remainder

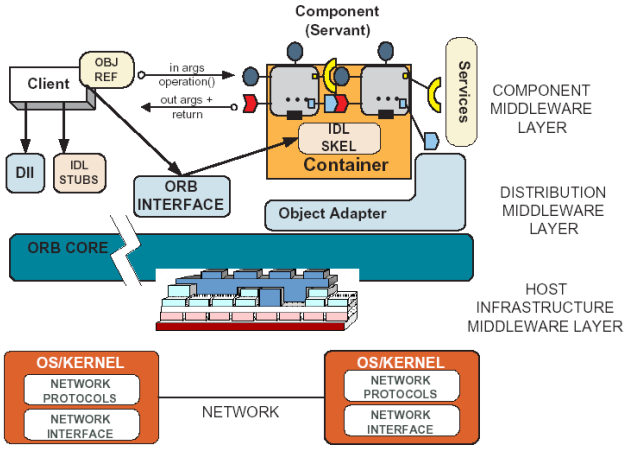

Figure 2: Layered CCM Architecture

of this section describes the key CCM elements in Figure 2.

Components. *Components* in CCM are implementation entities that collaborate with each other via *ports*. CCM supports several types of ports, including (1) *facets*, which define an interface that accepts point-to-point method invocations from other components, (2) *receptacles*, which indicate a dependency on point-to-point method interface provided by another component, and (3) *event sources/sinks*, which indicate a willingness to exchange typed messages with one or more components.

Container. A *container* in CCM provides the run-time environment for one or more components that manages various pre-defined hooks and strategies, such as persistence, event notification, transaction, and security, used by the component(s). Each container is responsible for (1) initializing instances of the component types it manages and (2) connecting them to other components and common middleware services. Developer-specified metadata expressed in XML can be used to instruct CCM deployment mechanisms how to control the lifetime of these containers and the components they manage. The meta-data is present in XML files called *descriptors*.

Component assembly. In a distributed system, a component may need to be configured differently depending on the context in which it is used. As the number of component configuration parameters and options increase, it can become tedious and error-prone to configure applications consisting of many individual components.

To address this problem, the CCM defines an *assembly* entity to group components and characterize the meta-data that describes these components in an assembly. Each component's meta-data in turn describes the features available in it (*e.g.*, its properties) or the features that it requires (*e.g.*, its dependencies). CCM assemblies are defined using XML Schema templates, which provide an implementation-independent mechanism for describing component properties and generating default configurations for CCM components. These assembly configurations can preserve the required QoS properties [17] and establish the necessary configuration and interconnections among groups of components.

Component server. A *component server* is an abstraction that is responsible for aggregating *physical* entities (*i.e.*, implementations of component instances) into *logical* entities (*i.e.*, distributed application services and subsystems). A CCM component server is a singleton [18] that plays the role of a factory to create containers and standardizes the role of a server process in the CORBA 2.x object model. Each component server is typically assigned a particular set of capabilities within a distributed system.

Component packaging and deployment. In addition to the runtime building blocks outlined above, the CCM also standardizes component implementation, packaging, and deployment mechanisms. Packaging involves grouping the implementation of component functionality – typically stored in a dynamic link library (DLL) – together with other meta-data that describes properties of this particular implementation. The CCM Component Implementation Framework (CIF) helps generate the component implementation skeletons and persistent state management automatically using the Component Implementation Definition Language (CIDL).

3. PUBLISHER/SUBSCRIBER SERVICE INTEGRATION CHALLENGES IN COMPONENT MIDDLEWARE

This section describes the R&D challenges associated with providing QoS-enabled publisher/subscriber service access via the container programming model. We describe the context in which these challenges arise, identify the specific problems that must be addressed, and outline an approach to resolving the challenges. Section 4 then describes how we applied those solution approaches in CIAO by using patterns to enhance CIAO's container framework to support a range of publisher/subscriber services.

Context. The *container programming model* establishes a paradigm for component interaction with a run-time execution environment. Specifically, the container programming model designates a software entity, *i.e.*,the *container*, to manage a set of components. The container supports an API framework through which developers control component lifecycles and access common middleware services, including publisher/subscriber services. The container architecture decouples components from application server implementations, thereby enhancing flexibility and reuse.

A CCM container provides component implementations with access to common CORBA services, including (but not limited to) two distinct publisher/subscriber services: the Event Service [13] and the Notification Service [19]. CIAO supports both these services, as well as an extended version of the Event Service, the Real-Time Event Service [20]. Each publisher/subscriber service that CIAO supports has different capabilities and is accessed via a distinct interface. Components can also elect to have events propagated *directly*, which causes the container itself to play the role of an event channel, bypassing other intermediate publisher/subscriber components.

Although the direct mechanism is lightweight and efficient for components collocated in the same process, it does not scale well for distributed components, such as remote event channel configurations that place one or more event channels between the publisher and subscriber components [8].

Problem. The standardized CCM container interface is designed to encapsulate only a subset of the Notification Service, rendering the broader range of CORBA-based publisher/subscriber services inaccessible. The container interface is generic by design, simplifying its use and enabling component interoperability among CCM implementations. Its design, however, prevents access to any advanced publisher/subscriber service capabilities, such as QoS guarantees. For example, the standard container interface lacks any mechanism to specify priorities, timeouts, or event correlation, which are essential capabilities for effective DRE application operation. Likewise, exposing components to dissimilar and proprietary publisher/subscriber services requires developers to manipulate overly complicated and confusing interfaces, prevents compliance with relevant OMG standards, and bloats component memory footprint. As shown in Figure 3, component QoS policies must be captured within the component logic.

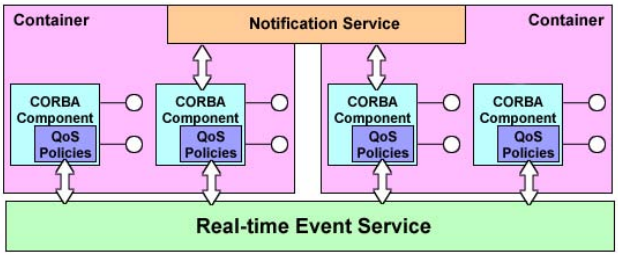

Figure 3: A generic CCM container framework

Solution approach → Enhance containers to encapsulate, implement, and configure publisher/subscriber services. CIAO's container framework presents application components with a uniform interface (shown as the *QoS adapter* in Figure 4) through which various publisher/subscriber services can be selected and

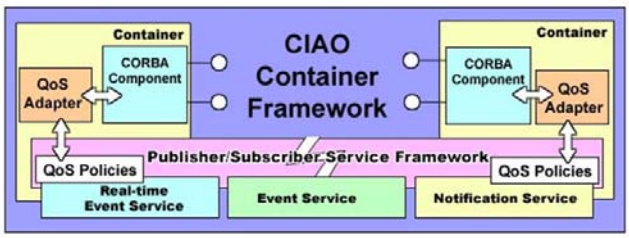

Figure 4: CIAO Publisher/Subscriber Service Framework

configured. This interface enables application components to use any combination of the publisher/subscriber services supported by CIAO. More importantly, by encapsulating service-specific QoS specification operations within a high-level interface, the container framework supports QoS configuration of publisher/subscriber services (*e.g.*, assignment of priorities and latency thresholds), without

exposing components to service implementation details. Components can therefore invoke these services in a straightforward manner (*i.e.*, without becoming tightly coupled to low-level CORBA programming details), while preserving the flexibility and customizability of the underlying services. Figure 4 illustrates the different aspects of QoS-enabled publisher/subscriber service support within the CIAO container framework. In contrast to Figure 3, the QoS policies are decoupled from application component logic via the QoS adapter encapsulated within the container.

Section 4 describes how we applied patterns to enhance CIAO's container framework to support a range of publisher/subscriber services.

4. PATTERN-ORIENTED SOLUTIONS IN CIAO

We employed pattern-oriented software [18, 9] techniques to address the publisher/subscriber service access challenge discussed in Section 3. This section describes our solution approach in detail.

4.1 Overview of the CIAO Container Framework Architecture

The CIAO publisher/subscriber service framework exposes a local CORBA interface, `CIAO::EventServiceBase`, that encapsulates the heterogeneity of the available publisher/subscriber services. Internally, the framework defines four implementations of `EventServiceBase`: one for direct communication, one for the Event Service, one for the Real-Time Event Service, and one for the Notification Service. The `EventServiceBase` interface contains operations to connect, disconnect, and publish, (*a.k.a.*, *push*) events. A container instantiates the appropriate implementation of `EventServiceBase` for each event supplier within its context. The class interactions are shown in Figure 5.

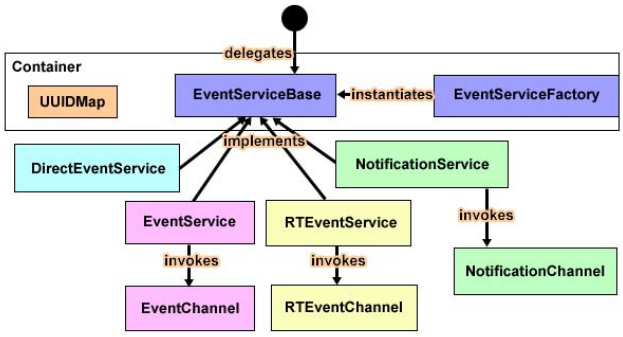

Figure 5: Class Interactions in the CIAO Publisher/Subscriber Service Framework

Although a container may create many `EventServiceBase` objects, only a single event or notification channel is created per-container. All components managed by a container then use this channel. Each `EventServiceBase` object maintains the event configuration information for the supplier and all associated consumer components. When a container receives a `connect()`, `disconnect()`, or `push()` operation request, it delegates the call to the associated `EventServiceBase` object. The container uses unique connection identifiers based on the component's universally unique id (UUID) to map individual suppliers and consumers to `EventServiceBase` objects.

4.2 CIAO Container Framework Design Goals and Implementation Strategies

The CIAO publisher/subscriber service architecture shown in Figure 4 employs patterns to address the design goals of the container programming model outlined in Section 3. For example, while maintaining efficiency and reliability requirements, CIAO preserves the lightweight nature of components. An individual component need know nothing about the services that implement event-passing – the container encapsulates that complexity. It therefore follows that component application developers need not be concerned with these details, further simplifying the design of the core component logic.

For each design goal mandated by the CIAO container programming model, our pattern-oriented solution is detailed below. Figure 6 demonstrates the interactions between patterns in the publisher/subscriber service framework.

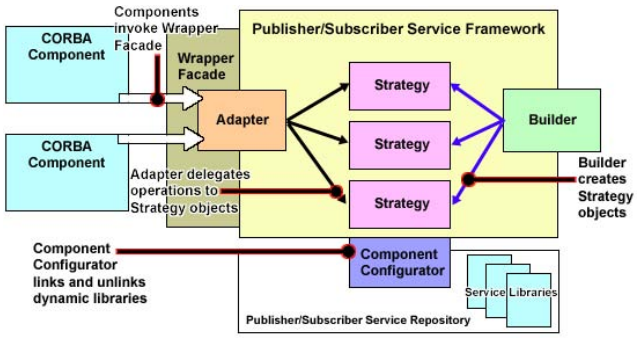

Figure 6: Pattern Interactions in the CIAO Publisher/Subscriber Service Framework

Design goal 1 → Simplify component development by exposing a simple publisher/subscriber service interface. To achieve this design goal in CIAO, we used the *Adapter* pattern [18], which converts one interface into a different one expected by clients. Since the CORBA publisher/subscriber services were designed prior to CCM their interfaces are not ideal for components. The container therefore implements an adapter that provides components with a simple, uniform interface and translates calls on that interface into calls on a specific publisher/subscriber service interface. The benefits of this design are twofold: (1) component developers need not concern themselves with peculiar configuration interfaces and (2) no matter what changes occur to the underlying publisher/subscriber services, the interface exposed to components does not change.

Design goal 2 → Enhance reuse and extensibility by allowing new publisher/subscriber services to be easily plugged-in. To achieve this design goal in CIAO we used the *Strategy* pattern [18], which defines classes that encapsulate different algorithms and declares an interface common to all supported algorithms. In CIAO, a local CORBA interface serves as the common interface, and the various implementations encapsulate algorithms for the different publisher/subscriber services. This design results in service implementations that are interchangeable from the container perspective. After object creation, the container has no knowledge of the actual algorithm being used, which enables fast operation delegations and simplifies container design.

Design goal 3 → Reduce the memory footprint of the container by decoupling the creation of publisher/subscriber service instances. To achieve this design goal in CIAO we used the *Builder* pattern [18], which separates the construction of objects from their representation, allowing the same process to create multiple implementations of the same object type. The creation of publisher/subscriber service instances is somewhat complex in CIAO since (1) they must be initialized properly and (2) different implementations are possible. CIAO defines a builder class that encapsulates the complexity of creating and initializing publisher/subscriber service implementations. The result is finer control of the construction process, isolation of construction code, and the ability to vary the publisher/subscriber service implementation.

Design Goal 4 → Ensure components only incur the cost of services that are required by deferring publisher/subscriber service selection and configuration decisions until run-time. To achieve this design goal in CIAO we used the *Component Configurator* pattern [21], which allows an application to link and unlink its component implementations at run-time. In CIAO, publisher/subscriber service libraries are loaded dynamically on-demand to avoid encumbering the application with unused services, while still allowing components to wait until deployment time to select a particular service. This mechanism provides the flexibility to initiate, suspend, resume, and terminate services. More generally, CIAO enables entire applications to be composed of independently developed services, thereby simplifying composition and deployment.

Design Goal 5 → Enable component access to the full set of QoS features available in publisher/subscriber services by encapsulating service-specific QoS specification operations within a high-level interface. To achieve this design goal in CIAO we used the *Wrapper Facade* pattern [22], which defines a concise, robust, portable, and maintainable interface to encapsulate low-level functions and data structures. The CIAO publisher/subscriber framework implements CORBA interfaces that contain operations to configure QoS parameters for an individual publisher or subscriber connection. The operations of these interfaces forward invocations to the corresponding service-specific operations for each publisher/subscriber service. This design results in a concise and robust programming interface capable of configuring the QoS features in multiple dissimilar publisher/subscriber services.

5. CONCLUDING REMARKS AND FUTURE WORK

The integration of QoS-enabled publisher/subscriber services into component middleware allows developers of DRE systems to leverage the benefits of component-centric software development. This paper described how QoS-enabled publisher/subscriber services can be integrated into a component middleware container framework using patterns. The integration techniques we described allow DRE system developers to utilize the full extent of publisher/subscriber service capabilities without becoming tightly-coupled to service implementations. Finally, this paper demonstrated each integration strategy in CIAO. The CIAO open-source CCM distribution is available for download at `www.dre.vanderbilt.edu/CIAO/`.

Our future work will focus on applying Model-Driven Middleware (MDM) [23] to simplify the configuration of publisher/subscriber services in QoS-enabled component middleware. MDM is an emerging technology that integrates *model-based software development paradigms* (including Model-Integrated Computing [24, 25] and the OMG's Model Driven Architecture [26]) with component middleware (including Real-time CORBA [27] and QoS-enabled CCM [14]) to resolve the software development and validation challenges of large-scale DRE middleware and applications.

MDM tools and techniques are particularly important for event-based DRE software components that require custom QoS configurations to target multiple OS, network, and hardware platforms, each of which may have slightly different requirements. In such cases, event QoS requirements (such as latency thresholds and priorities) may only be known when components are deployed, rather than when they are developed. Contemporary component middleware frameworks use XML descriptor files to specify the publisher/subscriber configurations and QoS constraints associated with particular software deployments. The component deployment mechanism is responsible for parsing these files and making the appropriate invocations on the publisher/subscriber configuration interface provided by the container.

It is tedious and error-prone, however, to *manually* specify the QoS requirements of large-scale event-based DRE component deployments. Component middleware has become complex to configure due to an increasing number of operating policies (such as transaction and security properties, persistence and lifecycle management, and publisher/subscriber QoS configurations) that exist at multiple middleware layers and employ legacy specification mechanisms not based on XML. To further complicate matters, many combinations of policies are semantically invalid and will result in system failure.

To address these challenges, we are developing a MDM toolsuite named *Component Synthesis with Model-Integrated Computing* (CoSMIC) [28, 29] that employs our QoS-enabled CCM implementation, CIAO. The CoSMIC toolsuite supports the development, assembly, and deployment of QoS-enabled CCM component applications. CoSMIC currently uses CIAO as the default target middleware since it is the only standards-based component middleware platform that supports DRE systems. We also plan support for other component middleware as their support for DRE systems matures.

6. ADDITIONAL AUTHORS

Additional Authors: Balachandran Natarajan (Institute for Software Integrated Systems, Vanderbilt University, email: `bala@dre.vanderbilt.edu`)

7. REFERENCES

[1] 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.

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

[3] Object Management Group, *CORBA Components*, OMG Document formal/2002-06-65 edition, June 2002.

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

[5] Microsoft Corporation, "Microsoft .NET Development," msdn.microsoft.com/net/, 2002.

[6] Christopher D. Gill, David L. Levine, and Douglas C. Schmidt, "The Design and Performance of a Real-Time

CORBA Scheduling Service," *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, vol. 20, no. 2, Mar. 2001.

[7] David A. Karr, Craig Rodrigues, Yamuna Krishnamurthy, Irfan Pyarali, and Douglas C. Schmidt, "Application of the QuO Quality-of-Service Framework to a Distributed Video Application," in *Proceedings of the 3rd International Symposium on Distributed Objects and Applications*, Rome, Italy, Sept. 2001, OMG.

[8] Carlos O'Ryan, Douglas C. Schmidt, and J. Russell Noseworthy, "Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations," *International Journal of Computer Systems Science and Engineering*, vol. 17, no. 2, Mar. 2002.

[9] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture—A System of Patterns*, Wiley & Sons, New York, 1996.

[10] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 332–383, Aug. 2001.

[11] Douglas C. Schmidt and et al., "TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems," *IEEE Distributed Systems Online*, vol. 3, no. 2, Feb. 2002.

[12] Douglas C. Schmidt and Carlos O'Ryan, "Patterns and Performance of Real-time Publisher/Subscriber Architectures," *Journal of Systems and Software, Special Issue on Software Architecture - Engineering Quality Attributes*, 2002.

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

[14] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Craig Rodrigues, Balachandran Natarajan, Joseph P. Loyall, Richard E. Schantz, and Christopher D. Gill, "QoS-enabled Middleware," in *Middleware for Communications*, Qusay Mahmoud, Ed. Wiley and Sons, New York, 2003.

[15] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 3.0.2 edition, Dec. 2002.

[16] 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.

[17] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Christopher D. Gill, Balachandran Natarajan, Craig Rodrigues, Joseph P. Loyall, and Richard E. Schantz, "Total Quality of Service Provisioning in Middleware and Applications," *The Journal of Microprocessors and Microsystems*, vol. 27, no. 2, pp. 45–54, mar 2003.

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

[19] Object Management Group, *Notification Service Specification*, Object Management Group, OMG Document formal/2002-08-04 edition, Aug. 2002.

[20] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, Atlanta, GA, Oct. 1997, ACM, pp. 184–199.

[21] Douglas C. Schmidt and Stephen D. Huston, *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*, Addison-Wesley, Reading, Massachusetts, 2002.

[22] 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.

[23] Aniruddha Gokhale, Douglas C. Schmidt, Balachandran Natarajan, Jeff Gray, and Nanbor Wang, "Model Driven Middleware," in *Middleware for Communications*, Qusay Mahmoud, Ed. Wiley and Sons, New York, 2003.

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

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

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

[27] Arvind S. Krishna, Douglas C. Schmidt, Ray Klefstad, and Angelo Corsaro, "Real-time CORBA Middleware," in *Middleware for Communications*, Qusay Mahmoud, Ed. Wiley and Sons, New York, 2003.

[28] Aniruddha Gokhale, Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Arvind Krishna, George T. Edwards, Gan Deng, Emre Turkay, Jeffrey Parsons, and Douglas C. Schmidt, "Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications," *Submitted to The Journal of Science of Computer Programming: Special Issue on Model Driven Architecture*, 2004.

[29] Aniruddha Gokhale, "Component Synthesis using Model Integrated Computing," `www.dre.vanderbilt.edu/cosmic`, 2003.