Model-driven Configuration and Deployment of Component Middleware Publish/Subscribe Services

George Edwards, Gan Deng, Douglas C. Schmidt, Aniruddha Gokhale, and Bala Natarajan
{edwardgt,dengg,schmidt,gokhale,bala}@dre.vanderbilt.edu

Department of Electrical Engineering and Computer Science, Vanderbilt University
Nashville, TN 37235, USA*

Abstract

Quality of service (QoS)-enabled publish/subscribe services are available in component middleware platforms, such as the CORBA Component Model (CCM). Today, however, these platforms lack a simple and intuitive way to integrate publish/subscribe service configurations and deployments. This paper illustrates how generative model-driven techniques and tools can automate many service configuration and deployment tasks associated with integrating publish/subscribe services into QoS-enabled component-based systems. We evaluate these techniques in the context of a real-time avionics mission computing problem involving a system with over 50 components. Our evaluation finds that an automated model-driven configuration of a reusable component middleware framework not only significantly reduces handwritten code and but also simultaneously achieves high reusability and composability of CCM components.

Keywords: Real-time Publish/subscribe Service, Component Middleware, CORBA Component Model, Model-based Systems.

1 Introduction

Emerging trends. 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 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 conventional commercial-off-the-shelf (COTS) component middleware include the CORBA Component Model (CCM) [3] and Java 2 Enterprise Edition (J2EE) [4].

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 messages, called *events*, to other components [5]. Event-based communication models are particularly relevant for large-scale DRE systems[1] (such as avionics

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

[1] In this context, *systems* include the OS/hardware/network, middleware, and applications.

mission computing [6, 7], distributed audio/video processing [8, 9], and distributed interactive simulations [10, 11]) since they help reduce software dependencies and enhance system composability and evolution. In particular, the *publish/subscribe* architecture [12] of event-based communication allows application components to interact anonymously and asynchronously [13]. The publish/subscribe 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.

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

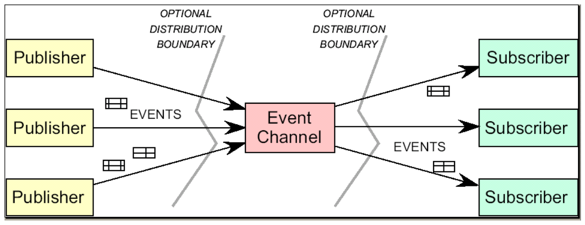


Fig. 1. Relationships Between Components in a Publisher/subscribe Architecture

Applying model-driven middleware to publish/subscribe architectures. Our previous work on publish/subscribe architectures focused on the patterns and performance optimizations of event channels in the context of QoS-enabled *distributed object computing* (DOC) middleware [14], specifically a highly scalable [11] and real-time [15, 6] CORBA Event Service [16]. This paper extends our previous work on DOC middleware as follows:

- We describe key challenges associated with configuring and deploying publish/subscribe services in QoS-enabled *component middleware*. Component middleware enhances DOC middleware to enable the composition, configuration, and deployment of reusable services and applications more rapidly and robustly.
- We present a methodology for resolving these challenges based on *Model-Driven Middleware* (MDM) [17], which is a generative programming paradigm that integrates (1) model-driven development technologies, such as Model-Integrated Computing [18, 19] and the OMG's Model Driven Architecture [20], and (2) QoS-enabled component middleware technologies, such as Real-time CORBA [21] and the *Component-Integrated ACE ORB* (CIAO), which is our QoS-enabled implementation of CCM.
- We describe the *Event QoS Aspect Language (EQAL)*, which is an MDM tool that supports graphical representations of crosscutting concerns (such as component event port connections and event channel configuration) associated with publish/subscribe QoS configurations and federated publish/subscribe service deployments. This paper explores EQAL support for two generative aspects – service configuration and deployment – that help automate much of the integration of publish/subscribe services into QoS-enabled component-based DRE systems.
- We evaluate empirically how MDM reduces the amount of handwritten code in developing component applications that utilize publish/subscribe services. Our results applying MDM

to a 50 component avionics mission computing scenario show that it dramatically reduces handwritten code via automated configuration of a reusable component framework, while simultaneously eliminating accidental complexities incurred when hand-crafting 100+ XML-based descriptor files, which are the standard way to describe configure and deploy publish/-subscribe services in CCM.

Paper organization. The remainder of this paper is organized as follows: Section 2 outlines the deployment and configuration capabilities in the CORBA Component Model (CCM) that is leveraged by our work; Section 3 describes the key challenges and solution approaches associated with configuring and deploying publish/subscribe services in QoS-enabled component middleware; Section 4 demonstrates in detail how our EQAL MDM tool addresses the publish/-subscribe configuration and federation deployment challenges; Section 5 empirically evaluates the extent to which EQAL reduces the amount of handwritten code in developing component-based avionics mission computing systems that utilize publish/subscribe services; Section 6 compares our MDM approach with related work; and Section 7 presents concluding remarks and outlines future work.

2 Overview of CCM Configuration and Deployment Capabilities and DAnCE

The CORBA Component Model (CCM) [3] specification describes a component architecture and standardizes component implementation, packaging, and deployment mechanisms [22]. Components in DRE systems may need to be configured differently, depending on the context in which they are used. For example, it might be necessary to collocate or replicate related components to improve performance or resilience to failures by distributing functionality encapsulated as components. CCM implementations provide the capabilities described below that DRE systems can use to (1) collocate and/or distribute components depending on application QoS needs, (2) make the necessary connections between communicating components, (3) group components together to form reusable artifacts, and (4) deploy groups of components on various nodes in a target environment.

Component packaging groups implementations of component functionality (typically stored in a dynamic link library) together with metadata that describes the features available in it (*e.g.*, its properties) or the features that it requires (*e.g.*, its dependencies). A component package is the vehicle for deploying a single component implementation. The CCM Component Implementation Framework (CIF) uses the Component Implementation Definition Language (CIDL) to generate the component implementation skeletons and persistent state management automatically.

Component assembly groups components and characterizes the metadata that describes these components in an assembly [22] via a *component assembly descriptor*, which specify how components are connected together, on what hosts they will run, how the components are instantiated, among numerous other properties. A component assembly package is the vehicle for deploying a set of interrelated component implementations. CCM assemblies are defined via 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 and establish the necessary configuration and interconnections among groups of related components.

Component deployment installs and connects a logical component topology to form a physical computing environment. The component topology is specified by an assembly package. A *deployment tool* deploys individual components and assemblies of components to an installation site, *e.g.*, a set of hosts on a network. Based on an assembly descriptor and user input, a CCM deployment tool installs and activates component homes and instances. It also configures component properties and connects components together via interface and event ports, as designated by an assembly descriptor.

Figure 2 illustrates the Deployment and Configuration Engine (DAnCE) [23], which is our implementation of the OMG deployment and configuration (DnC) specification [22], addressing the DnC crosscutting concerns of DRE systems, DAnCE supports the creation, control, and termination of components on the nodes of the target environment. DAnCE takes the DnC XML descriptors and creates an in-memory representation of the metadata. Run-time services in DAnCE then populate a global deployment plan, which provides an in-memory representation. Depending on the number of nodes needed for a particular deployment, DAnCE splits the global plan into multiple local deployment plans that are optimally configured for deployment on that platform.

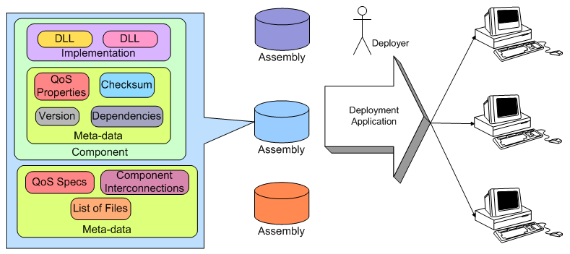


Fig. 2. DAnCE Deployment and Configuration Engine

3 Meeting the Challenges of Configuring and Deploying Publish/subscribe Systems

QoS-enabled component middleware platforms, such as the CIAO [24] and Qedo [25], leverage the benefits of component-based software development and preserve the optimization patterns and principles of DOC middleware. Before developers of event-based DRE systems can derive benefits from QoS-enabled component middleware, however, they must reduce the complexity of configuring and deploying publish/subscribe services. In particular, DRE system developers must resolve the following challenges associated with publish/subscribe mechanisms provided by conventional component middleware:

1. **Configuring publish/subscribe service QoS**, where there are currently no standard means of configuring component middleware mechanisms to deliver appropriate QoS to DRE systems, and
2. **Deploying federated publish/subscribe services**, where there are currently no standard policies and mechanisms to deploy a federation of publish/subscribe services for DRE systems.

This section explains the context in which each challenge outlined above arises, identifies the specific problems that must be addressed, and outlines solution approaches that help resolve the challenge. Section 4 then illustrates how we have applied these solutions using Model-Driven Middleware (MDM).

3.1 Challenge 1: Configuring Publish/Subscribe Quality-of-Service

Context. *Configurability* is an important requirement for many publish/subscribe services developed using middleware. For example, various operating policies (such as threading and buffering strategy) of the CORBA publish/subscribe services can be customized programmatically via invocations on a configuration interface. The drawbacks with DOC middleware approaches to configurability, however, are (1) *reduced flexibility* due to tight coupling of application logic with crosscutting configuration and deployment concerns, such as publish/subscribe relationships and choice of various types of publish/subscribe services, such as the CORBA-based Event, Real-time Event, and Notification Services. and (2) *impeded reuse* due to tight coupling of application logic with specific QoS properties, such as event latency thresholds and priorities.

In contrast, component middleware publish/subscribe services enhance flexibility and reuse by using meta-programming techniques (such as the XML descriptor files in CCM) to specify component configuration and deployment concerns. This approach enables QoS requirements to be specified *later* (*i.e.*, just before run-time deployment) in a system's lifecycle, rather than *earlier* (*i.e.*, during component development). For example, the configuration framework provided by the CIAO component middleware parses XML configuration files and make appropriate invocations on a publish/subscribe service configuration interface. This approach is useful for DRE systems that require custom QoS configurations for various target OS, network, and hardware platforms that have different capabilities and properties.

Problem. Conventional component middleware relies upon *ad hoc* techniques based on *manually* specifying the QoS requirements for DRE component systems. Unfortunately, configuring component middleware manually is hard [26] due to the number and complexity of operating policies, such as transaction and security properties, persistence and lifecycle management, and publish/subscribe QoS configurations. These policies exist at multiple layers of middleware and often employ non-standard legacy specification mechanisms, such as configuration files that use proprietary text-based formats.

Moreover, given component interoperability needs across various platforms (*e.g.*, CCM and J2EE) and the existence of multiple publish/subscribe services within individual platforms (*e.g.*, the CORBA Event Service and Notification Service), a component-based application may use several publish/subscribe services. To further complicate matters, certain combinations of policies are semantically invalid and can result in system failure. For example, if multiple levels of priorities for events are supported, a priority-based thread pool model should be used rather than a reactive threading model [27]. Care should be taken to ensure that lower level configurations support end-to-end priorities, *e.g.*, using Real-time CORBA priority-banded connections [28].

Most publish/subscribe services based on DOC middleware (including the CORBA Event and Notification Services) do not validate QoS specifications automatically. It is hard, moreover, to *manually* validate QoS configurations for semantic compatibility. This process is particularly daunting for large-scale, mission-/safety-critical DRE systems, where the cost of human error is most egregious.

Solution approach → Develop MDM tools to create publish/subscribe service configuration models. MDM tools can help application developers create QoS specifications for DRE systems more rapidly and correctly by automatically generating configuration descriptor files and enforcing constraints among publish/subscribe policies via model checkers [29]. These benefits are particularly important when component applications are maintained and evolved over an extended period of time since (1) QoS configurations can be modified more easily to reflect changing OS, network, and hardware platforms and (2) QoS configurations for system enhancements can be checked systematically for compatibility with legacy specifications.

To attain these benefits, we developed the *Event QoS Aspect Language* (EQAL), which is an MDM tool that models configurations for three CORBA-based publish/subscribe services: (1) the Event Service [16], (2) Real-time Event Service [6, 15], and (3) Notification Service [30]. EQAL informs users if invalid combinations of QoS policies are specified. After publish/subscribe

QoS models are complete and validated, EQAL can also synthesize the XML configuration files used by the underlying component middleware to configure itself. Section 4.2 illustrates how we applied EQAL to configure key QoS properties of component middleware publish/subscribe services.

3.2 Challenge 2: Deploying Publish/Subscribe Services in Target Networks

Context. *Scalability* is another important requirement for many publish/subscribe systems. Large-scale publish/subscribe systems consist of many components and event channels distributed across network boundaries and possibly different administrative domains, and each event channel may have many consumers. Naive implementations of publish/subscribe services send a separate event across the network for each remote consumer, which can transmit the same data multiple times (often to the same target host) and incur network and host overhead that is excessive for many resource-constrained DRE applications. As the number of channels and/or consumers grows, these types of publish/subscribe services can become a bottleneck.

To minimize the overhead of publish/subscribe services, multiple event channels can be linked together to form *federated* configurations [11, 15], where event channels are assigned to particular hosts and events received by one channel are propagated automatically to other channels in the federation. Figure 3 illustrates how CIAO's publish/subscribe services support federated event channels. In CIAO's federated publish/subscribe services, suppliers and con-

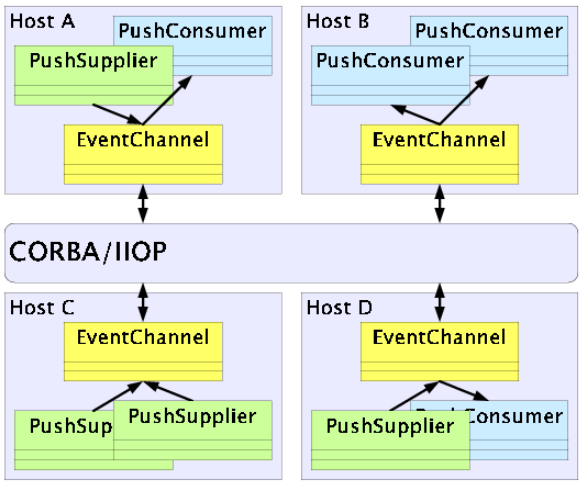


Fig. 3. Federated Event Channels in CIAO

sumers that are collocated on the same host connect to a local event channel. Each local event channel communicates with other event channels when events sent by suppliers are destined for consumers on remote hosts. This design reduces latency in large-scale DRE systems when consumers and suppliers exhibit *locality-of-reference*, *i.e.*, where event consumers are on the same host as event suppliers. In such cases, only local C++ method calls are needed instead of remote CORBA operation calls. Moreover, if multiple remote consumers are interested in the same event, only one message is sent to each remote event channel, thereby reducing network utilization.

In CIAO's federated publish/subscribe services, event channel *gateways* are used to mediate the communication between remote event channels, while suppliers and consumers communicate with each other via local event channels. Each gateway is a CORBA component that connects to the local event channel as a supplier and connects to the remote event channel as a consumer.

CIAO supports three types of event channel gateways: CORBA IIOP, UDP, and IP multicast. In CIAO's federated publish/subscribe services, application developers need not write tedious and error-prone code manually to perform bookkeeping operations, such as creating and initializing gateways that federate event channels.

As discussed in Section 2, CCM deployment tools install individual components and assemblies of components on target sites, which are normally a set of hosts on a network. Similarly, event channels must be assigned to hosts in the target network. CIAO's federated publish/-subscribe services are integrated via its DAnCE component deployment tool. The input to DAnCE is an XML data file that (1) specifies the event channel deployment sites and (2) automatically creates and initializes the event channel gateways at the appropriate sites.

Problem. Although the DAnCE CCM deployment tool provided by CIAO shields application developers from having to write bookkeeping code for its federated publish/subscribe services, application developers still must *hand-craft* federation deployment descriptor metadata using an XML schema based on the OMG Deployment and Configuration specification [22]. Hand-crafting descriptor metadata involves determining information about the type of federations (*i.e.*, CORBA IIOP, UDP, or IP multicast), identifying remote event channels, and identifying local event channels. Moreover, the metadata must address the following deployment requirements: (1) each host could have its own event channels, event consumers, event suppliers, and event channel gateways, (2) each event consumer and event supplier only communicates with a event channel collocated in the same host, (3) event channels distributed across network boundaries are connected through event channel gateways, (4) each connection between an event channel and an event supplier should be uniquely identified, (5) each connection between an event channel and an event consumer should be identified by using existing events, as defined in step (4), and (6) each connection between an event channel and an event channel gateway should also be identified by using existing events, as defined in step (4).

Experience has shown [31, 32] that it is hard for DRE developers to keep track of many complex dependencies when deploying federated publish/subscribe services. Without tool support, therefore, the effort required to deploy a federation involves hand-crafting deployment descriptor metadata in an *ad hoc* way. Since large-scale DRE systems may involve many different types of events and event channels, *ad hoc* ways of writing metadata to deploy publish/subscribe services are tedious and error-prone. Addressing this challenge requires techniques that can analyze, validate, and verify the correctness and robustness of federated event channel deployments.

Solution approach → Develop MDM tools to deploy event channel federations in a visual, intuitive way. MDM tools can synthesize the metadata for deploying a federated publish/-subscribe service from *models* of the interactions among different event-related components (*e.g.*, event suppliers, event consumers, event channels, and various types of event channel gateways). MDM tools can also generate the metadata needed to deploy federated publish/subscribe services that are syntactically and semantically valid. Section 4.3 shows how the EQAL MDM tool was applied to deploy publish/subscribe service federations more effectively than existing approaches.

4 Resolving Publish/subscribe Service Configuration and Deployment Challenges in CoSMIC

This section describes how we have employed Model-Driven Middleware (MDM) techniques to address the challenges of publish/subscribe service configuration and federated deployment discussed in Section 3. We present an overview of the *Event QoS Aspect Language* (EQAL) and show how EQAL helps to resolve the challenges of (1) publish/subscribe configuration described in Section 3.1 and federated event service deployment described in Section 3.2. As shown in Figure 4, EQAL is part of the configuration generative tools of the *Component Synthesis via Model Integrated Computing* (CoSMIC) [23] MDM toolsuite. CoSMIC contains a collection of domain-specific modeling languages and their associated analysis/synthesis tools that support various phases of DRE system development, assembly, configuration, and deployment. CoSMIC

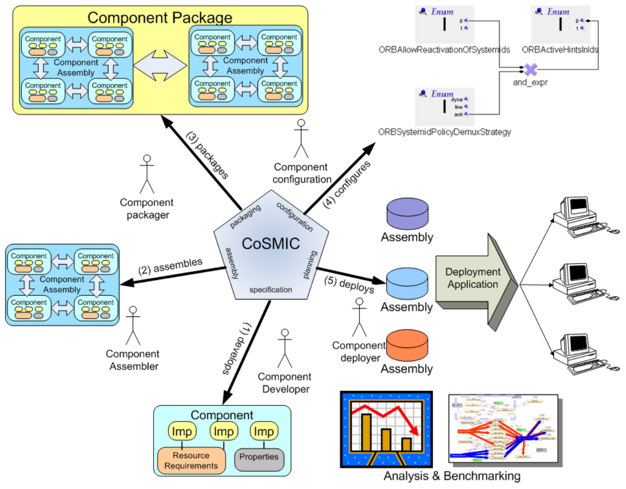


Fig. 4. CoSMIC MDM Toolsuite

generates crosscutting concerns, such as XML descriptors, that can be woven into component applications via the DAnCE deployment and configuration engine outlined in Section 2.

4.1 Overview of the Event QoS Aspect Language (EQAL)

Conventional component middleware frameworks use XML files to describe publish/subscribe service configurations and deployments. These textual specifications are unnecessarily complex and error-prone, however, as discussed in Sections 3. To address these problems we created the EQAL MDM tool, which supports graphical representations of QoS configurations and federated deployments of publish/subscribe services.

EQAL is developed using the Generic Modeling Environment (GME) [33], which is a generative technology for creating domain-specific modeling languages and tools [18]. GME can be programmed via *metamodels* and *model interpreters*. Metamodels define modeling languages (called *paradigms*) that specify the syntax and semantics of the modeling element types, their properties and relationships, and presentation abstractions defined by a domain-specific modeling language. Model interpreters can traverse a paradigm's modeling elements and perform various actions, such as analyzing model properties and generating code.

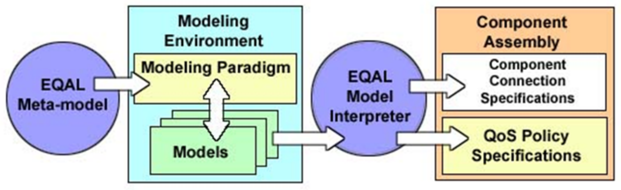


Fig. 5. The Event QoS Aspect Language Architecture

The EQAL paradigm in GME consists of the two complementary entities shown in Figure 5 and described below:

• **EQAL metamodel** , which defines a modeling paradigm in which modelers specify the desired publish/subscribe service (*e.g.*, the CORBA Event Service, Notification Service, or Real-time Event Service) and the configuration of that service for each component event connection. Based on application needs, modelers can also specify how event channels are assigned to different hosts and whether/how they must be linked together to form federations.

• **EQAL model interpreters** that can (1) validate configuration and deployment models and (2) synthesize text-based middleware publish/subscribe service and federation service configuration- and deployment-specific descriptor files from models of a given component assembly. Component deployers can build publish/subscribe service configurations for component applications using the EQAL modeling paradigm and its model interpreters.

The remainder of this section describes how EQAL addresses the challenges presented in Section 3.

4.2 Configuring Publish/subscribe Quality-of-Service in EQAL

To address the publish/subscribe service configuration challenge described in Section 3.1, the EQAL *configuration paradigm* specifies publish/subscribe QoS configurations, parameters, and constraints. For example, the EQAL metamodel contains a distinct set of modeling constructs for each publish/subscribe service supported by CIAO. Example policies and strategies that can be modeled include filtering, correlation, timeouts, locking, disconnect control, and priority.

Publish/subscribe service policies can have different scopes, ranging from a single port to an entire event channel. EQAL's publish/subscribe service configurations can therefore be provisioned at the following three levels of granularity:

- **Channel scope**, which applies to all components using the channel. Each event channel is specified with a number of policies that control its behavior, such as event filtering, event correlation, timeouts, and locking. These policies control how the channel handles all connections and events.
- **Proxy scope**, which applies to a single component port. Each event port is associated with a proxy object. Certain QoS policies are configured at the proxy level, such as threading control settings, average execution time, and worst-case execution time. QoS parameters can be provided for each connection by configuring the proxy. Naturally, connection-level parameters for a proxy must be consistent with channel-level policies.
- **Event scope**, which applies to an event instance. A limited number of QoS settings, such as timeouts, can be specified for an individual event instance.

The EQAL metamodel allows modelers to provision reusable and sharable configurations at each level of granularity outlined above. Modelers assign configurations to individual event connections and then construct filters for each connection. EQAL supports two forms of event generation using the push model: (1) a component may be an exclusive supplier of an event type or (2) a component may supply events to a shared channel.

Dependencies among publish/subscribe QoS policies, strategies, and configurations can be complex. Ensuring coherency among policies and configurations is therefore a non-trivial source of complexity in component middleware [34]. During the modeling phase, EQAL ensures that dependencies between configuration parameters are enforced by declaring constraints on the contexts in which individual options are valid, *e.g.*, priority-based thread allocation policies are only valid with component event connections that have assigned priorities. EQAL can then automatically validate configurations and notify users of incompatible QoS properties during model validation, rather than at component deployment- and run-time.

To ensure semantically consistent configurations, violation of constraint rules should be detected early in the modeling phase rather than later in the component deployment phase. To support this capability, EQAL provides a constraint model checker that validates the syntactic and semantic compatibility of event channel configurations to ensure the proper functioning of publish/subscribe services. EQAL's model checker uses GME's constraint manager, which is a lightweight model-checker that implements the standard OMG Object Constraint Language (OCL) specification [35].

EQAL's model interpreters perform the following two distinct configuration aspects:

• **XML descriptor generation.** EQAL contains an interpreter that synthesizes XML descriptors used by the DAnCE component deployment framework to indicate the QoS requirements of individual component event connections. Since the CCM specification does not explicitly address the mechanisms for ensuring component QoS properties, the EQAL-generated descriptors are based on a schema developed for the Boeing Bold Stroke project for their Prism [34] extensions to CCM (the XML descriptors remain compliant with the CCM specifications, however). Boeing's Bold Stroke schema has been carefully crafted, refined, tested, and optimized in the context of production DRE avionics mission computing systems [31, 32].

EQAL generates the XML descriptors for one service at a time. To complete the interpretation process, EQAL makes multiple passes through the model hierarchy, corresponding to each different type of publish/subscribe services, until all the service connections are configured. To simplify the interpreter implementation, EQAL [36] uses the Visitor pattern [37], which utilizes a double-dispatch mechanism to apply file-generation operations to different type of modeling elements.

• **Service configuration file generation.** EQAL also contains an interpreter that generates event channel service configuration files, called `svc.conf` files, that are used by the underlying publish/subscribe services to select the appropriate behaviors of event channel resource factories [11]. These factories are responsible for creating many strategy objects that control the behavior of event channels. CIAO supports many (*i.e.*, more than 40) options/policies for different types of publish/subscribe services, which increases the complexity for application developers who must consider numerous design choices when configuring the publish/subscribe services. Interactions between event channel policies are complex due to the possibility of incompatible groupings of options, making hand-crafting these files hard, *e.g.*, priority-based thread allocation policies are only valid with component event connections that have assigned priorities.

Much of the complexity associated with validating event channel QoS configurations is accomplished by EQAL's modeling constraints and GME's lightweight model checker. These constraints prevent application developers from specifying inconsistent or invalid combinations of policies. After a set of policy settings is validated via modeling constraints, EQAL generates an *event channel descriptor* (`.ecd`) file that contains valid combinations of policy settings chosen for a particular service configuration.

4.3 Deploying Publish/subscribe Service Federations in EQAL

To address the publish/subscribe federation service deployment challenge outlined in Section 3.2, the EQAL *deployment paradigm* specifies how components and event channels are assigned to hosts on a target network. To address the scalability problem in any large-scale event-based architecture, CIAO provides publish/subscribe services that supports event channel federation. With CIAO's publish/subscribe services, an event channel federation can be implemented via CORBA gateways. Application developers can configure the location of the gateways to utilize network resources effectively.

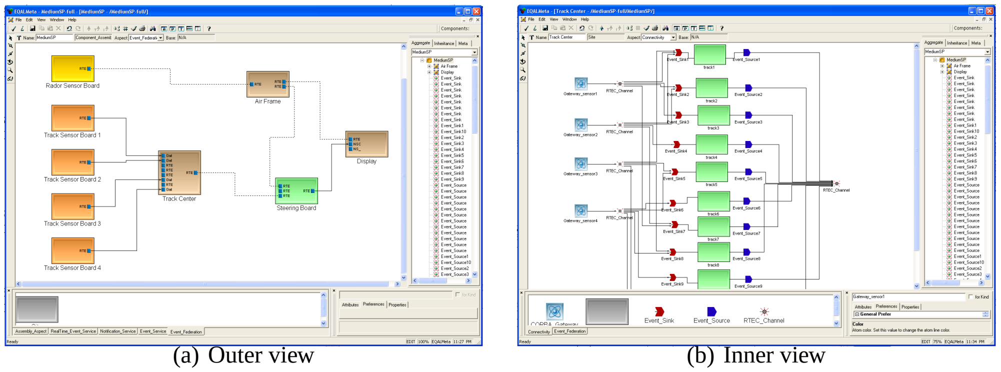
For example, collocating a gateway with its consumer event channel (*i.e.*, the one it connects to as a supplier) eliminates the need to transmit events that have no consumer event channel subscribers. Application developers can also choose different types of gateways based on different application deployment scenarios with different networking and computing resources. These deployment decisions have no coupling with, or bearing on, component application logic. The same set of components can therefore be reused and deployed into different scenarios without modifying application code manually.

The EQAL modeling paradigm allows three types of federation (*i.e.*, CORBA IIOP, UDP, or IP multicast) to be configured in a deployment. For event channel federation models, the EQAL metamodel defines two levels of syntactic elements:

– The **outer-level**, which contains the host elements as basic building blocks and allows users to define the hosts present in the DRE system and
– The **inner-level**, which represents a host containing a set of elements (including event channels, CORBA IIOP gateways, UDP senders and receivers, IP multicast senders and receivers, and event type references) that allow users to configure the deployment of these artifacts inside a host.

These two levels are associated with each other via *link parts*, which act as connection points between two different views of a model (such as adjacent layers of a hierarchical model) to indicate some form of association, relationship, or dataflow between two or more models. The inner-level elements are exposed to the outer-level in the form of link parts from the outside view, which can be used to connect them to form a federation.

Figure 6 (a) is a screenshot that illustrates how we used EQAL to model the outer-level view of CIAO federated publish/subscribe service in the real-time avionics mission computing application outlined in Section 1. This figure shows the outer-level model of the deployment of the federated publish/subscribe service, which includes nine physically distributed locations that host CCM components. Figure 6 (b) shows the inner-level of the federation configurations, which establish four CORBA gateways in the track center module to form a federation that reduces network traffic.



(a) Outer view (b) Inner view

Fig. 6. EQAL Deployment Model for a Real-time Avionics System

To ensure the validity of event channel federation models during the deployment phase, each event channel's configurations and settings must be model-checked to ensure that they are consistent with the federation types. For example, IP multicast uses the Observer pattern [37] capabilities of CIAO's event channels. When a user chooses IP multicast as the type of event channel federation, this observer functionality must be enabled for IP multicast to work properly. These constraints can be checked automatically using EQAL.

Section 4.2 described how EQAL's model interpreters handle configuration aspects. EQAL also contains a model interpreter for the deployment aspect, which synthesizes the federated publish/subscribe service assembly and deployment descriptor XML files. The information captured in these files includes the relationship between each artifact, the physical location of each supplier, consumer, event channel, and CORBA gateway. This file is subsequently fed into the DAnCE CCM deployment tool, which deploy the federated system to its designed target nodes.

5 Evaluating the Merits of Model-Driven Middleware

The EQAL MDM tool described in Section 4 is designed to reduce the configuration and development effort of DRE applications. This section evaluates how the EQAL MDM tool helps alleviate common sources of complexity in a representative DRE application based on QoS-enabled component middleware. We conduct this evaluation in the context of Boeing Bold Stroke, which is an open experimental platform (OEP) [34] in the DARPA PCES [38] program for real-time avionics mission computing. The Bold Stroke OEP integrates and demonstrates model-based, language-based, and middleware-based technologies to productively program and evolve crosscutting aspects that support composable DRE middleware for publish/subscribe-based avionics systems. Crosscutting concerns addressed by the Bold Stroke OEP include synchronization, memory management, fault tolerance, real-time deadlines, end-to-end latencies, and bandwidth and CPU management.

We implemented the OEP's *Medium-sized* (MediumSP) scenario, which is a challenge problem product scenario in the DARPA PCES OEP [38], using EQAL and CIAO CCM middleware platform. The MediumSP scenario is representative of real-time avionics mission computing systems that employ event-driven data flow and control [31, 32]. This scenario consists of 50+ components with complex event dependencies that control embedded sensors and perform calculations to maintain displays. In this type of mission-critical DRE system, reliability and stringent QoS assurance are essential.

5.1 Resolving Complexities of Configuring and Deploying Publish/Subscribe Services

The configuration and deployment of publish/subscribe services is an example *crosscutting concerns* that need to be refactored, modularized, and then composed back into the middleware. EQAL helps to automate this process by enabling different types of services to be configured based on application needs and available resources. For example, EQAL's integrated model checker and model interpreters can flag invalid configurations at design- and/or deployment-time, thereby eliminating sources of accidental complexity that would otherwise manifest themselves at run-time. EQAL also significantly increases component reusability and maintainability by decoupling (1) the type of publish/subscribe service configured into the system from (2) the functionality of application components.

In the Bold Stroke MediumSP scenario, there are over 50 components and most components act as both suppliers and consumers of events. Without automated support from an MDM tool like EQAL, application developers would face the following challenges when implementing the MediumSP scenario:

No systematic approach to specify policies/options. Manually configuring the publish/subscribe services via *ad hoc* techniques can overwhelm application developers since configuration of each service is intertwined with 20+ configuration parameters spread throughout different subsystems and layers of the component middleware and applications. For example, configuration parameters in the CIAO Real-time Event Service include the number of threads, cached execution time, worst-case execution time, average execution time, level of importance, entry point name, etc.

Low-level, complex glue code. Application developers would either have to write/maintain special-purpose "glue code" manually for different types of publish/subscribe services or treat the different services in the same way, *e.g.*, write different publish/subscribe-related glue code for different types of publish/subscribe services (such as Event Service, Real-time Event Service, and Notification Service) or a degenerate case of components communicating directly with each other.

Potential run-time errors. If the component middleware was configured improperly, application functionality and QoS behavior can be incorrect. For example, if a component server and its containers were not configured with support for real-time priorities and concurrency, it would be hard for an *ad hoc* configuration and deployment process to detect erroneous conditions, such as an application developer configuring a real-time event channel within the component middleware. These conditions would be detected only when the system was actually deployed and operating, at which point it might be too late to prevent serious run-time errors.

5.2 Evaluating How Component Middleware Minimizes Handwritten Code

In earlier generation of DOC middleware, developers had to explicitly handle the complexity of connecting to, and configuring the policies of, the underlying middleware. For example, in the CORBA 2.x DOC model, developers often manually configured the policies of middleware entities, such as publish/subscribe services, transaction services, and security services. Moreover, event channels were deployed by hand-crafting DOC middleware application servers to create and destroy event channels and gateways.

Figure 7 shows how application developers using CORBA DOC middleware must write code manually in the “Server” and “Impl” files. These manual programming activities result in the

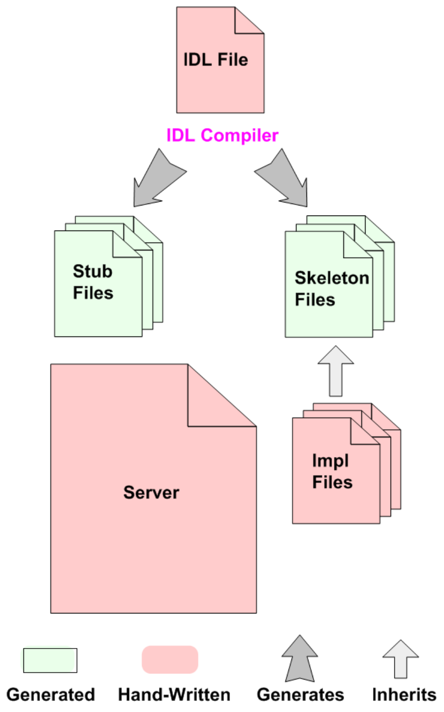


Fig. 7. Generated vs. Handwritten Code in the CORBA DOC Model

production of considerable repetitive boilerplate code, which may actually be larger than the application logic requiring publish/subscribe functionality! Moreover, these activities are error-prone since they require application developers to wrestle with middleware details, which are often low-level and proprietary.

In contrast, CCM middleware defines the container and component server elements to handle most of the glue code. Moreover, CCM shields component developers from low-level details of the underlying middleware via containers that provide the execution context in which application components run and mediate access to underlying middleware services. Likewise, generic component servers provide a standardized OS process in which components and event channels can be composed and run, thereby alleviating the need to write and deploy custom servers.

As illustrated in Figure 8, CCM middleware increases the amount of generated code compared with handwritten code because much of the boilerplate component server code, XML component descriptors, servants, and other glue code can be generated automatically. Likewise,

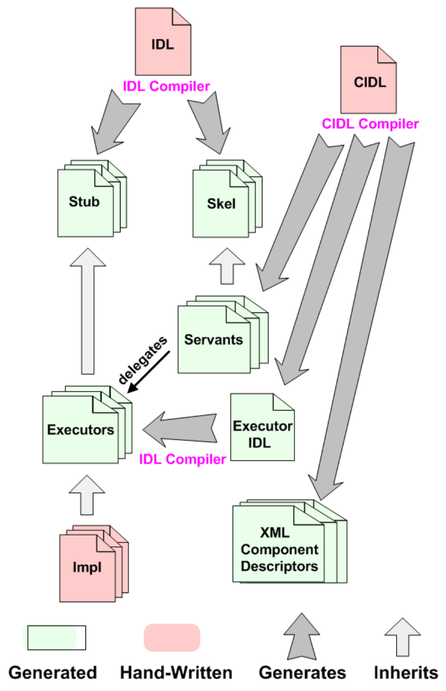


Fig. 8. Generated vs. Handwritten Code in the CORBA Component Model

the configuration and deployment of publish/subscribe services can be automated by CCM middleware. Below we highlight concretely how EQAL can automatically generate code for the MediumSP scenario that would otherwise be written manually.

Creating event channels. To create an event channel, the following steps must be carried out: (1) *declaring the channel*, where a name is assigned to the channel, (2) *setting channel attributes*, where the appropriate service configurator file is specified to configure the channel, (3) *instantiating the channel*, where memory is allocated for the channel and its constructor is called, (4) *activating the channel*, where the channel's servant is enabled, and (5) *registering the channel*, where the channel is registered with a CORBA server. The implementation of the event channel creation member function requires 22 lines of C++.

Specifying service policies. When configuring a real-time service, the QoS requirements for a supplier or consumer must be specified. This specification process involves the following steps: (1) *building a filter*, where a consumer specifies the logical conditions (*e.g.*, based on the supplier and type of an event) under which an event should be delivered, and (2) *specifying real-time properties*, where desired QoS properties (*e.g.*, priorities and required execution times) are given for each event supplier and consumer. The implementation for specifying service policies requires 334 lines of C++.

Connecting suppliers and consumers. Connecting a supplier or consumer to a channel invokes the following steps: (1) *obtaining an administrative object*, where the channel supplies a reference to a configuration entity known as an admin object, (2) *obtaining a proxy*, where the admin supplies a reference to a connection point, known as a proxy object, (3) *creating a consumer/supplier servant*, where a servant is declared, instantiated, and activated, and (4) *connecting to the event channel*, where a reference to the servant is passed to the proxy object. The implementations of connection methods for suppliers and consumers required 45 and 54 lines of C++, respectively.

Implementing supplier and consumer servants. Implementing a supplier or consumer servant involves inheriting from an abstract base class and implementing several hook methods. The supplier and consumer servant methods are then dispatched by an event channel when an event is pushed or when a disconnection occurs. Code to handle these occurrences in accordance with specific requirements of an application must be provided by application developers. The amount of code necessary is heavily dependent on application requirements, but required 94 lines of C++ in the Bold Stroke MediumSP scenario.

5.3 Applying EQAL to Real-time Avionics Mission Computing

Based on the discussions in Sections 5.1 and 5.2, we now focus on applying EQAL to model the Bold Stroke MediumSP scenario, which is shown in Figure 9. We use this scenario to fur-

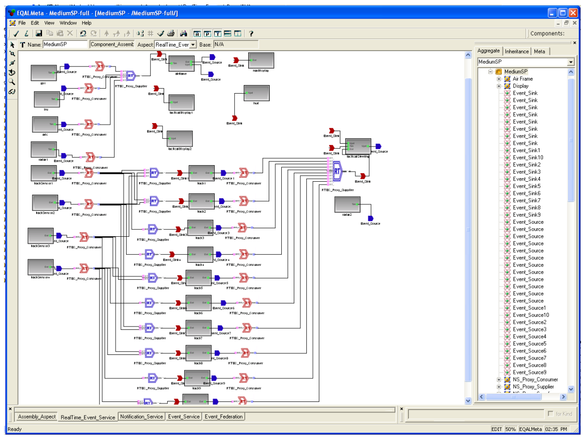


Fig. 9. EQAL Model for the Bold Stroke MediumSP Scenario

ther qualify the code reduction that results from employing the EQAL MDM tool and CIAO component middleware instead of handwritten glue-code and DOC middleware.

Based on the particular assignment of components to host sites, different event channel federations are possible. For the scenario depicted in Figure 9, our EQAL model distributes components among nine hosts. Given the number and location of suppliers, consumers, event channels, and gateways, we describe the effort (measured roughly in terms of the number of artifacts developed) needed to integrate publish/subscribe services in the following phases of the MediumSP scenario:

Phase 1: Managing event channel lifecycles, which involves creating and destroying event channels. Since EQAL's MediumSP model incorporates multiple publish/subscribe services, multiple event channels of differing types must be managed. In the MediumSP example, more than 10 event channels must be managed.

Phase 2: Initializing gateways, suppliers, and consumers, which involves (1) making interconnections between supplier/consumer components, event channels, and gateways and (2) implementing supplier/consumer servants. There are a total of 42 event suppliers and 38 event consumers in the MediumSP scenario that must be created and initialized. Based on the deployment scenario, we have set up 12 gateways to federate these event channels (other types of gateways may chose to form different types of federations).

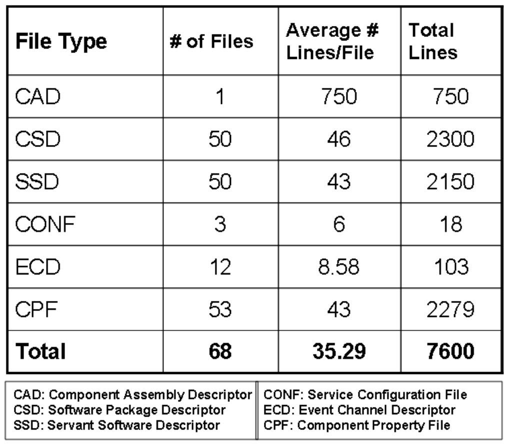
Phase 3: Specifying service policies, which involves establishing control strategies, such as filter criteria and QoS parameters. All 42 suppliers and 38 consumers of events specify real-time rate requirements and 8 of the consumers require filter specifications to correlate events. Each supplier and consumer must be configured individually with the appropriate policies, such as filtering, correlation, timeouts, locking, disconnect control, and priority.

In each phase above, glue-code and XML descriptors must be provided to configure and deploy the required publish/subscribe services. To avoid hand-crafting boilerplate software, EQAL combines component middleware with MDM technology to synthesize glue-code and XML descriptors. The resulting reduction in complexity for the MediumSP scenario is summarized below:

The assembly of 50+ components for the MediumSP scenario requires a complicated *component assembly* file that stores the connection information between component ports as XML descriptors, partitions for process collocation, and interrelationships with other descriptors (*e.g.*, the relationship between the interface definitions and component implementations) whose details are spread across other assembly files, such as the *implementation artifact descriptor* (`.iad`) file. EQAL shields DRE system developers from these low-level details by ensuring that all this metadata and dependencies are captured appropriately in various descriptor files it generates in conjunction with other CoSMIC tools, such as PICML [23].

Every component requires two descriptor files: (1) the *software package descriptor* for the component, which contains general information about the software (such as author, description, license information, and dependencies on other software packages), followed by one or more sections describing implementations of that software, and (2) the *servant software descriptor*, which CIAO deployment tools use to load the desired servant library. For ∼50 components, ∼100 files are therefore required. Once again, EQAL shields DRE system developer from these low-level details by generating these files automatically, thereby ensuring that all interdependencies are captured appropriately in the descriptor files.

For the publish/subscribe service in the MediumSP component assembly, a *component property descriptor* (CPF file) is generated for each component event port, an *event channel descriptor* (ECD file) will be generated for each real-time event channel filter, and CIAO's service configuration file (`svc.conf`) file) will be generated for each event channel configuration. As a result, 12 ECD files, 53 CPD files, and 1 `svc.conf` file are generated by EQAL. Figure 10 summarizes the lines of code saved by not having to hand-craft these files.



File Type	# of Files	Average # Lines/File	Total Lines
CAD	1	750	750
CSD	50	46	2300
SSD	50	43	2150
CONF	3	6	18
ECD	12	8.58	103
CPF	53	43	2279
Total	**68**	**35.29**	**7600**

CAD: Component Assembly Descriptor	CONF: Service Configuration File
CSD: Software Package Descriptor	ECD: Event Channel Descriptor
SSD: Servant Software Descriptor	CPF: Component Property File

Fig. 10. Amount of Code Reduction for Metadata in Bold Stroke MediumSP Scenario

Each component, event service, and their servants are distinguished via a unique identifier (called a UUID) within the descriptor files mentioned above. Moreover, it is necessary to ensure that when referring to a specific component or event service, the same UUID is referenced across the different descriptor files. This requirement can yield accidental complexities when descriptor files are hand-crafted manually. In the MediumSP scenario, this results in ∼100 UUIDs that are referred to across the ∼100 descriptor files. EQAL's generative tools eliminate these accidental complexities by synthesizing the proper UUID references in the descriptor files.

The XML tags used to represent real-time properties and resulting configurations in component middleware for integrating real-time event channels are not standardized by OMG yet. Different QoS-enabled component middleware, such as CIAO, therefore define their own XML tags using either XML DTDs or XML Schema definitions. For the MediumSP scenario, this results in enhancing the above-mentioned descriptor files with CIAO-specific real-time properties, and configuration XML tags and data. EQAL shields DRE system developers from these low-level details by synthesizing the proper middleware-specific tags and data in the appropriate descriptor files.

The XML output shown below illustrates an EQAL-generated *component property descriptor* (`.cpf`) file for a `BMDevice` component in the MediumSP scenario. This file contains the port real-time QoS properties, such as the number of threads and worst-case execution time.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE properties SYSTEM "properties.dtd">
<properties>
  <struct name="BMDevice-data_available-" type="ACEXML_RT_Info">
    <description>Real-time Scheduler info for
                 BMDevice::data_available</description>
    <simple name="port" type="string">
                 <value>data_available</value> </simple>
    <simple name="period" type="string"> <value>0</value> </simple>
    <simple name="entry_point" type="string"> <value></value> </simple>
    <simple name="criticality" type="string">
      <value>MEDIUM_CRITICALITY</value></simple>
    <simple name="enabled" type="string">
      <value>DEPENDENCY_NON_VOLATILE</value></simple>
    <simple name="importance" type="string">
      <value>MEDIUM_IMPORTANCE</value></simple>
    <simple name="quantum" type="string"> <value>0</value> </simple>
    <simple name="threads" type="string"> <value>1</value> </simple>
    <simple name="cached_execution_time" type="string">
                 <value>0</value> </simple>
    <simple name="worst_case_execution_time" type="string">
                 <value>0</value> </simple>
    <simple name="info_type" type="string">
                 <value>OPERATION</value> </simple>
    <simple name="typical_execution_time" type="string">
                 <value>0</value> </simple>
  </struct> </properties>
```

In summary, EQAL dramatically reduces an application developer's effort for an event-based DRE application, such as the Bold Stroke MediumSP scenario, by generating a considerable amount of glue-code and XML descriptors that are then used by other CCM-related tools, such as DAnCE [23], to configure and deploy the CIAO publish/subscribe service middleware.

6 Related Work

This section reviews related work on model-based software development and describes how modeling, analysis, and generative programming techniques have been used to model and provision QoS capabilities for QoS-enabled component middleware and applications.

Cadena [29] is an integrated development environment for modeling and model-checking component-based DRE systems. Cadena's model checking environment is particularly well suited to event-based inter-component communication via real-time event channels. Cadena also provides an Event Configuration Framework (EMF) [39] that allows modeling event channel properties, which can be model checked. We are integrating our CoSMIC toolsuite with Cadena to leverage its model checking capability. Both Cadena and CoSMIC have been used in the context of Boeing's OEP on avionics mission computing and for CCM-based applications.

Publish/subscribe service modeling research is also provided by Ptolemy II [40], which is a tool for modeling concurrent hybrid and embedded systems based on the actor-oriented design approach [41]. This approach uses the abstraction of actors, ports, interfaces, and model of computations to model the system. Publish/subscribe is a model of computation in Ptolemy II. Our efforts with CoSMIC are relatively orthogonal with Ptolemy II, *e.g.*, our target is QoS-component middleware and could fit in various families of applications.

The Aspect Oriented Middleware (AOM) research at the University of Toronto, is focusing on the extension and refinement of publish/subscribe service for effectively supporting information dissemination applications, such as modeling of uncertainty [42] or semantic matching [43] in publish/subscribe services. Compared with our work on EQAL, the AOM project focuses on enhancing the publish/subscribe services with uncertainty capabilities because in some situations exact knowledge to either specify subscriptions or publications is not available. Although this approach proposes a new publish/subscribe model based on possibility theory and fuzzy set theory to process uncertainties for both subscriptions and publications, it does not provide a visual modeling language to model the QoS-enabled DRE systems, which is provided by EQAL.

Zanolin et. al. [44] propose an approach to support the modeling and validation of publish/-subscribe service architectures. Application-specific components are modeled as UML statechart diagrams while the middleware is supplied as a configurable predefined component. As to validation, properties are described with live sequence charts (LSCs) and transformed into automata. Components, middleware, and properties are translated into Promela [44] and then passed to SPIN (linear temporal logic) to validate the architecture. The main difference between this approach and our work is that Zanolin et. al.'s approach is based on standard UML statecharts to model and validate the publish/subscribe service, but unlike EQAL they do not model the QoS properties and the federation aspect in the publish/subscribe architecture.

7 Concluding Remarks

This paper showed how a Model-Driven Middleware (MDM) tool called the *Event QoS Aspect Language* (EQAL) can automate and simplify the integration of publish/subscribe services into QoS-enabled component-based systems. EQAL verifies model validity and generates XML metadata to configure and deploy publish/subscribe service federations in QoS-enabled component middleware. Our experience applying EQAL in the context of DRE systems (such as the avionics mission computing system analyzed in Section 5) is summarized below:

– EQAL allows DRE system deployers to create rapidly and synthesize publish/subscribe QoS configurations and federation deployments via *models* that are much easier to understand and analyze than hand-crafted code.
– EQAL decouples configuration and deployment decisions from application logic, which enhances component reusability by allowing QoS specifications (and their associated implementations) to change based on the target network architecture.
– EQAL helps alleviate the complexity of validating the QoS policies of publish/subscribe services for DRE component applications, which is particularly important for large-scale DRE systems that evolve over long periods of time.
– EQAL reduces the amount of code written by application developers for event-based DRE systems by employing a configurable publish/subscribe service framework, which eliminates the need to write code that handles event channel lifecycles, QoS configurations, and supplier/consumer connections.

EQAL, CoSMIC, and CIAO are open-source software available for download at `www.dre.vanderbilt.edu/cosmic/`.

References

1. Schantz, R.E., Schmidt, D.C.: Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications. In Marciniak, J., Telecki, G., eds.: Encyclopedia of Software Engineering. Wiley & Sons, New York (2002)
2. Heineman, G.T., Councill, B.T.: 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 edn. (2002)
4. Sun Microsystems: Java^{TM} 2 Platform Enterprise Edition. java.sun.com/j2ee/index.html (2001)
5. Pietzuch, P.R., Shand, B., Bacon, J.: A Framework for Event Composition in Distributed Systems. In Endler, M., Schmidt, D., eds.: Proceedings of the 4th ACM/IFIP/USENIX International Conference on Middleware (Middleware '03), Rio de Janeiro, Brazil, Springer (2003) 62–82
6. Harrison, T.H., Levine, D.L., Schmidt, D.C.: The Design and Performance of a Real-time CORBA Event Service. In: Proceedings of OOPSLA '97, Atlanta, GA, ACM (1997) 184–199
7. Gill, C.D., Levine, D.L., Schmidt, D.C.: The Design and Performance of a Real-Time CORBA Scheduling Service. Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware **20** (2001)
8. Loyall, J., Gossett, J., Gill, C., Schantz, R., Zinky, J., Pal, P., Shapiro, R., Rodrigues, C., Atighetchi, M., Karr, D.: Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications. In: Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21), IEEE (2001) 625–634
9. Karr, D.A., Rodrigues, C., Krishnamurthy, Y., Pyarali, I., Schmidt, D.C.: 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, OMG (2001)
10. Noseworthy, R.: IKE 2 – Implementing the Stateful Distributed Object Paradigm . In: 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002), Washington, DC, IEEE (2002)
11. O'Ryan, C., Schmidt, D.C., Noseworthy, J.R.: Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations. International Journal of Computer Systems Science and Engineering **17** (2002)
12. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture—A System of Patterns. Wiley & Sons, New York (1996)
13. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and Evaluation of a Wide-Area Event Notification Service. ACM Transactions on Computer Systems **19** (2001) 332–383
14. Schmidt, D.C., Natarajan, B., Gokhale, A., Wang, N., Gill, C.: TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. IEEE Distributed Systems Online **3** (2002)
15. Schmidt, D.C., O'Ryan, C.: Patterns and Performance of Real-time Publisher/Subscriber Architectures. Journal of Systems and Software, Special Issue on Software Architecture - Engineering Quality Attributes (2002)
16. Object Management Group: Event Service Specification Version 1.1. OMG Document formal/01-03-01 edn. (2001)
17. Gokhale, A., Schmidt, D.C., Natarajan, B., Gray, J., Wang, N.: Model Driven Middleware. In Mahmoud, Q., ed.: Middleware for Communications. Wiley and Sons, New York (2004)
18. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-Integrated Development of Embedded Software. Proceedings of the IEEE **91** (2003) 145–164
19. Gray, J., Bapty, T., Neema, S.: Handling Crosscutting Constraints in Domain-Specific Modeling. Communications of the ACM (2001) 87–93
20. Object Management Group: Model Driven Architecture (MDA). OMG Document ormsc/2001-07-01 edn. (2001)
21. Krishna, A.S., Schmidt, D.C., Klefstad, R., Corsaro, A.: Real-time CORBA Middleware. In Mahmoud, Q., ed.: Middleware for Communications. Wiley and Sons, New York (2003)
22. Object Management Group: Deployment and Configuration Adopted Submission. OMG Document ptc/03-07-08 edn. (2003)
23. Gokhale, A., Balasubramanian, K., Balasubramanian, J., Krishna, A., Edwards, G.T., Deng, G., Turkay, E., Parsons, J., Schmidt, D.C.: Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications. The Journal of Science of Computer Programming: Special Issue on Model Driven Architecture (2004)

24. Wang, N., Schmidt, D.C., Gokhale, A., Rodrigues, C., Natarajan, B., Loyall, J.P., Schantz, R.E., Gill, C.D.: QoS-enabled Middleware. In Mahmoud, Q., ed.: Middleware for Communications. Wiley and Sons, New York (2003)
25. Ritter, T., Born, M., Unterschütz, T., Weis, T.: A QoS Metamodel and its Realization in a CORBA Component Infrastructure. In: Proceedings of the 36^{th} Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003, Honolulu, HW, HICSS (2003)
26. Memon, A., Porter, A., Yilmaz, C., Nagarajan, A., Schmidt, D.C., Natarajan, B.: Skoll: Distributed Continuous Quality Assurance. In: Proceedings of the 26th IEEE/ACM International Conference on Software Engineering, Edinburgh, Scotland, IEEE/ACM (2004)
27. Schmidt, D.C.: Evaluating Architectures for Multi-threaded CORBA Object Request Brokers. Communications of the ACM Special Issue on CORBA **41** (1998)
28. Pyarali, I., Schmidt, D.C., Cytron, R.: Techniques for Enhancing Real-time CORBA Quality of Service. IEEE Proceedings Special Issue on Real-time Systems **91** (2003)
29. Hatcliff, J., Deng, W., Dwyer, M., Jung, G., Prasad, V.: Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In: Proceedings of the 25th International Conference on Software Engineering, Portland, OR (2003)
30. Object Management Group: Notification Service Specification. Object Management Group. OMG Document formal/2002-08-04 edn. (2002)
31. Sharp, D.C.: Reducing Avionics Software Cost Through Component Based Product Line Development. In: Proceedings of the 10th Annual Software Technology Conference. (1998)
32. Sharp, D.C.: Avionics Product Line Software Architecture Flow Policies. In: Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC). (1999)
33. Ledeczi, A., Bakay, A., Maroti, M., Volgysei, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. IEEE Computer (2001)
34. Sharp, D.C., Roll, W.C.: Model-Based Integration of Reusable Component-Based Avionics System. In: Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003. (2003)
35. Object Management Group: Unified Modeling Language: OCL version 2.0 Final Adopted Specification. OMG Document ptc/03-10-14 edn. (2003)
36. Edwards, G., Schmidt, D.C., Gokhale, A., Natarajan, B.: Integrating Publisher/Subscriber Services in Component Middleware for Distributed Real-time and Embedded Systems. In: Proceedings of the 42nd Annual Southeast Conference, Huntsville, AL, ACM (2004)
37. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA (1995)
38. Office, D.I.E.: Program Composition for Embedded Systems (PCES). `www.darpa.mil/ixo/` (2000)
39. Singh, G., Maddula, B., Zeng, Q.: Event Channel Configuration in Cadena. In: Proceedings of the IEEE Real-time/Embedded Technology Application Symposium (RTAS), Toronto, Canada, IEEE (2004)
40. Liu, J., Liu, X., Lee, E.A.: Modeling Distributed Hybrid Systems in Ptolemy II. In: Proceedings of the American Control Conference. (2001)
41. Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: Actor-Oriented Design of Embedded Hardware and Software Systems. Journal of Circuits, Systems, and Computers (2003) 231–260
42. Liu, H., Jacobsen, H.A.: Modeling uncertainties in Publish/Subscribe System. In: Proceedings of The 20th International Conference on Data Engineering (ICDE04), Boston, USA (2004)
43. Petrovic, M., Burcea, I., Jacobsen, H.A.: S-ToPSS: Semantic Toronto Publish/Subscribe System. In: Proceedings of the 29th VLDB Conference, Berlin, Germany (2003)
44. Zanolin, L., Ghezzi, C., Baresi, L.: An Approach to Model and Validate Publish/Subscribe Architectures. In: Proceedings of the SAVCBS'03 Workshop, Helsinki, Finland (2003)