Using Model-Integrated Computing to Compose Web Services for Distributed Real-time and Embedded Applications

Nanbor Wang
Balachandran Natarajan
{nanbor,bala}@cs.wustl.edu

Dept. of Computer Science
Washington University
One Brookings Drive
St. Louis, MO 63130

Douglas C. Schmidt
schmidt@uci.edu

Dept. of Electrical and Computer Engineering
University of California
616E Engineering Tower
Irvine, CA 92697

Aniruddha Gokhale
a.gokhale@vanderbilt.edu

Institute for Software Integrated Systems
Vanderbilt University
P O Box 36, Peabody
Nashville, TN 37203

Keywords: Middleware, Model-Integrated Computing, Model Driven Architectures, Architectural and CORBA Component Model

1 Introduction

Commercial-of-the-shelf (COTS) middleware technologies have matured considerably over the past decade. They are now widely used to help enhance the quality and reduce the time to develop an increasingly broad range of application domains. Historically, middleware has been applied to *enterprise applications*, which comprise a large class of applications that perform important business functions, such as planning enterprise resource usage, automating key business functions, and managing supply chains and customer relationships. Examples of enterprise applications include airline reservation systems, bank asset management systems, and just-in-time inventory control systems.

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

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

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

Examples of COTS component middleware include the CORBA Component Model (CCM) [1], Java 2 Enterprise Edition (J2EE) [2], and the Component Object Model (COM) [3], which use different APIs, different protocols, and different component models.

Ironically, one of the original motivations for middleware was to reduce system heterogeneity via layers that made the software infrastructure appear *virtually* homogeneous [4]. Unfortunately, the proliferation of middleware technologies over the past decade has created a new level of heterogeneity that needs to be addressed. Since the cost of abandoning existing, working middleware can be prohibitively high, there is a trend towards so-called *Web services* [5] that help integrate different types of middleware technologies. Emerging Web services technologies are positioning themselves to become the "middleware of middleware" [6] by intentionally accommodating heterogeneity in various layers, including applications, network protocols, operating systems, and the middleware itself.

To achieve these goals, Web services use ubiquitous protocol infrastructure (such as TCP/IP, HTTP, and SMTP) and XML-based messages and metadata (such as SOAP [7] or WSDL [8]) to exchange information with clients. By exchanging XML-formated messages, Web services for business applications can easily adopt existing business standards, such as Electronic Data Interchange (EDI). Web services can support either RPC-styled or message-passing communication models depending on the requirements of the applications. Clients can bind to Web services using location services, such as the *universal description, discovery, and integration* (UDDI) service that queries the locations and the descriptions of available Web services.

Although today's Web services offerings strive to integrate software applications that use different middleware technologies, a number of key technical challenges remain. Chief among these challenges include the following:

1. Reconciling different middleware technologies. To provide sufficient QoS support and take advantage of new technologies as they arise, middleware must inevitably work with heterogeneous OS platforms, interface with legacy systems written in different languages, and interoperate with multiple technologies from many suppliers. Support for heterogeneity is essential since different middleware technologies have different pros and cons for different types of system environments and application requirements. For example, the middleware capabilities needed to manage supply chains over the Internet are different than those needed to intercept and destroy cruise missiles in flight.

Web services can be extended to integrate systems that comprise many existing component middleware technologies, some of which are wedded to particular programming languages and platforms. For example, Microsoft's .NET [9] supports XML/SOAP Web services and is based on a common language runtime (CLR) and COM [3]. Conversely, Sun's ONE [10], IBM's WebSphere [11], and BEA's WebLogic [12] are based on Java, J2EE, and CORBA. These technologies provide tools that can expose existing middleware applications as Web services so they can be accessed by any clients capable of using Web services.

Integrating different middleware technologies using Web services remains hard, however, since different middleware technologies have their own interaction model, such as component lifecycle, component addressing, and error notification [13]. What is needed is a way to ensure the behavioral differences between middleware technologies are represented and reflected across Web service points.

2. Satisfying multiple quality of service (QoS) requirements in real-time. An increasing number of DRE applications, such as controllers for surface-mount component pick-and-place machines or total ship computing environments, require stringent QoS demands that must be satisfied simultaneously in real-time. In large-scale DRE systems, these QoS demands cross-cut multiple system layers and require end-to-end enforcement. Today's Web services technologies were designed for applications with conventional business-oriented QoS requirements, such as data persistence and transactional support, so they do not yet enforce the stringent QoS requirements of DRE applications effectively.

QoS-aware Web services are particularly useful for integrating DRE systems where application requirements constrain the choice of hardware, languages, operating systems, and middleware. Military command and control (C2) and intelligence, surveillance, and reconnaissance (ISR) systems (such as AWACS and JSTARS), are a good example of such systems since they monitor enemy air and group movements and deliver tracks and targeting information to coalition forces in real-time. By necessity, these large-scale "systems of systems" are heterogeneous since different sensors and processing equipment are provided by different suppliers at different points in time. Moreover, they must link together many diverse hardware devices, such as sensors, that communicate information, such as track reports and GPS coordinates, across wireless links. This information can be readily exchanged as XML data if DRE applications are composed of QoS-enabled Web services.

There are many benefits of enhancing Web services to meet the QoS requirements of DRE applications. For example, due to constraints on weight, power consumption, memory footprint, and performance, development techniques for DRE application software have lagged those used for mainstream desktop and enterprise software. In particular, DRE applications have historically been custom-programmed to implement their required QoS properties, making them so expensive to build and maintain that they cannot adapt readily to meet new functional or QoS requirements, hardware/software technology innovations, or market opportunities. What is needed therefore is a flexible component middleware infrastructure for Web services that preserves the existing support for heterogeneity, yet also provides multiple dimensions of QoS enforcement.

3. Accidental complexities in integrating software systems. To reduce lifecycle costs and time-to-market, application developers are attempting to assemble and deploy distributed applications that make up the backbone of Web services by selecting the right set of compatible COTS components, which in itself is a daunting task. The problem is further exacerbated by the existence of myriad strategies for configuring and deploying the underlying component middleware to leverage the environment advantages. Moreover, integrating application using multiple middleware technologies demands multiple skill sets which makes the task even more complicated.

Application developers therefore spend non-trivial amounts of time debugging problems associated with the selection of incompatible strategies and components. What is needed is an integrated set of processes and tools that can (1) select and validate a suitable configuration of middleware components and (2) generate optimized Web service configurations automatically.

A promising way to address the application-to-application integration challenges described above is to apply *Model-Integrated Computing* technologies [14]. Model-Integrated Computing is an emerging paradigm for expressing application functionality and QoS requirements at higher levels of abstraction than is possible using third-generation programming languages, such as Visual Basic, Java, C++, or C#. In the context of DRE applications, Model-Integrated Computing tools can be applied to

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

Understanding how to integrate Model-Integrated Computing and component middleware is essential to resolve the configuration, management, and deployment challenges of integrating DRE applications using the Web services technologies described above. This paper provides the following three contributions toward the successful integration of Model-Integrated Computing and component middleware that is essential to develop QoS-enabled Web services to address the challenges presented above:

- We illustrate how the Model-Integrated Computing paradigm can be applied to simplify the development of large-scale DRE applications that integrate reusable component middleware services using QoS-enabled Web services.
- We describe how QoS-enabled component middleware enables modeling and synthesis tools to rapidly develop, assemble, and deploy flexible Web services that support heterogeneity, yet can be tailored readily to meet the needs of DRE applications with multiple simultaneous QoS requirements.
- We discuss how emerging standards, such as the OMG Model Driven Architecture [15] based on UML [16] and XML [17], and the CORBA Component Model [1] can be used to enhance and complement Model-Integrated Computing technologies, thereby providing a standards-based approach to assemble and deploy Web services.

The remainder of this paper is organized as follows: Section 2 describes how Model-Integrated Computing and component middleware can be combined to resolve key challenges associated with DRE application integration; Section 3 illustrates how the OMG Model Driven Architecture and CORBA Component Model are standardizing the Model-Integrated Computing paradigm that QoS-enabled Web services can leverage; Section 4 explains how we are applying these technologies to synthesize component-based applications from high-level models in the *Component-Integrated ACE ORB* (CIAO) and *Component Synthesis with Model-Integrated Computing* (CoSMIC) projects; Section 5 compared our work on CIAO and CoSMIC with related research; and Section 6 presents concluding remarks.

2 Component Middleware and Model-Integrated Computing: A Powerful Approach to Building DRE Applications

DRE applications have a range of QoS requirements, including bandwidth, bounded communication, latency, guaranteed resource allocation, dependability, scalability, and security. Much of the complexity of implementing QoS-aware Web services from composing reusable middleware components arises from interactions between the software and its environment, *i.e.*, the structure, actuator response times, and event intervals with which the software interacts. For example, an intelligent automotive engine management system must interact with the transmission control system and anti-lock brake system to actuate the fuel injection circuit in time to provide responsive performance. This section presents an overview of component middleware and Model-Integrated Computing and then describes how combining the best elements of these two technologies can address the complexities associated with developing DRE applications.

2.1 Overview of Component Middleware

Middleware capabilities. Middleware is reusable software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware [4]. Its primary role is to bridge the gap between application programs and the lower-level hardware and software infrastructure to coordinate how parts of applications are connected and how they interoperate. When implemented properly, middleware can help to:

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

Various technologies, such as OSF's Distributed Computing Environment (DCE) [18], IBM's MQ Series [19], and CORBA [20], emerged over the past two decades to alleviate complexities associated with developing software for enterprise applications. Their successes have added the middleware paradigm to the familiar operating system, programming language, networking, and database offerings used by previous generations of software developers. By decoupling application-specific functionality and logic from the accidental complexities inherent in the infrastructure, middleware enables application developers to concentrate on programming application-specific functionality, rather than wrestling repeatedly with lower-level infrastructure challenges. Moreover, since emerging Web services standards only address how information can be exchanged—not how to implement the services—Web service developers can also benefit from the abstraction provided by middleware to make Web service implementations themselves more efficient and portable.

Limitations with object-oriented middleware. The Object Management Architecture (OMA) in the CORBA 2.x specifications [20] defines an object-oriented middleware standard for building portable distributed applications. The CORBA 2.x specification focuses on *interfaces*, which are contracts between clients and servers that define how clients *view* and *access* object services provided by a server. These objects can be distributed or collocated throughout a network. Although this model has certain virtues, such as location transparency, it has the following limitations [21]:

- **Lack of functional boundaries.** The CORBA 2.x object model treats all interfaces as client/server contracts. This object model, however, does not provide sufficient mechanisms to prevent tight coupling among collaborating object implementations. For example, object implementations that depend on other objects need to discover and connect to these objects explicitly. To construct complex distributed applications, therefore, application developers need to program the connections among interdependent services, which can yield brittle and non-reusable implementations.
- **Lack of generic application servers.** CORBA 2.x does not specify a generic *application server* framework to perform common "bookkeeping" work, including initializing the broker and its QoS policies, providing common services (such as an event service), and managing the runtime environment of components. Although CORBA 2.x standardized the interactions between object implementations and object request brokers (ORBs), server developers are still responsible for determining how object implementations are installed in an ORB and the interaction between the ORB and object implementations. The lack of a generic application server standard has yielded tightly coupled, *ad-hoc* application server implementations, which increase the complexity of software upgrades and reduce the reusability and flexibility of CORBA-based applications.

Promising solution → component middleware. In recent years, *component middleware* [22] has emerged to address the limitations with object-oriented middleware outlined above. Component middleware addresses these issues by creating a virtual boundary around application components with well-defined interfaces and composing and executing components in generic application servers. Popular COTS component middleware platforms being used for various distributed applications today include the CCM [1], J2EE [2], and COM [3], which provide the foundation for many Web services development frameworks.

Large-scale DRE applications require seamless integration of many hardware and software systems. As shown in Figure 1, these systems may be separated physically from each others, *e.g.,* an air traffic control system processes flight information from multiple regional radars. These systems can also be collocated physically, yet be disjoint *virtually*. For example, a custom real-time flight bulletin board may reside in the same airport as the approach flight management system that runs on a mainframe computer and processes the information required by the bulletin board. It is important for component middleware to meet the designed QoS requirements for Web services to deliver the expected quality of service.

Figure 1 also shows how *components* implement the DRE processing and control logic and how *containers* provide a common interface that allow different components to interact

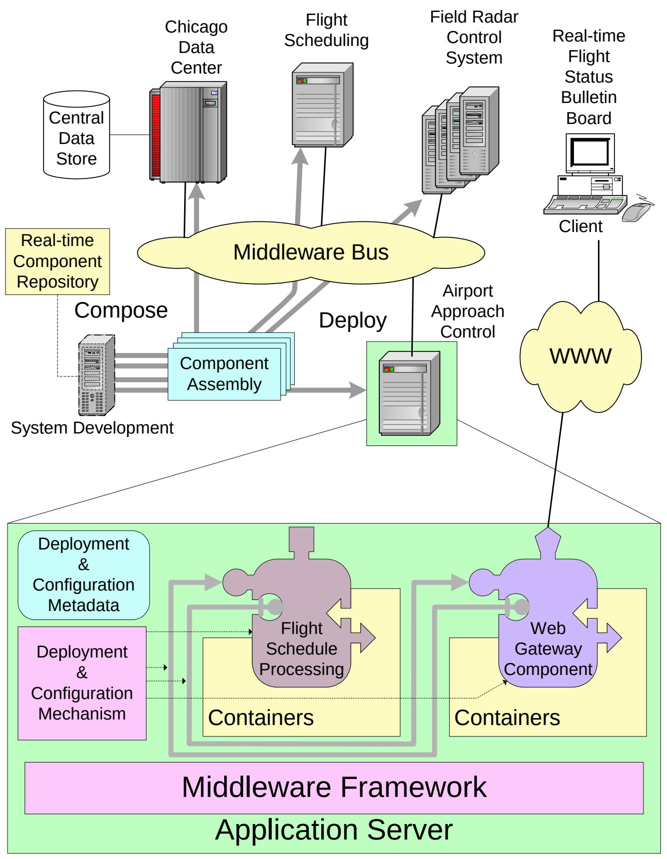


Figure 1: **Integrating DRE Applications with Component Middleware**

with the underlying middleware platform. In addition, this figure shows how generic application servers can be used to instantiate and manage containers and execute the components configured into them. Metadata associated with components provide instructions that application servers use to configure and connect components.

The many interdependent components in complex DRE applications often reside in multiple—possibly distributed—application servers. Each application server consists of some number of components that implement certain services for clients. These components in turn may include other collocated or remote services. In general, component middleware helps reduce initial software development efforts by integrating custom application components with reusable COTS components into generic application server frameworks. Moreover, as the requirements of DRE applications change, component middleware can help make it easier to migrate and redistribute certain services to adapt to new environments, while preserving key application QoS properties.

2.2 Overview of Model-Integrated Computing

Model-Integrated Computing (MIC) [14] is a development paradigm that applies domain-specific modeling languages systematically to engineer computing systems ranging from small-scale real-time embedded systems to large-scale distributed enterprise applications. MIC provides rich, domain-specific modeling environments, including model analysis and model-based program synthesis tools [23]. In the MIC paradigm, application developers model an integrated, end-to-end view of the entire application, including the interdependencies of its components. Rather than focusing on a single, custom application, therefore, MIC models capture the essence of a class of applications. MIC also allows the modeling languages and environments themselves to be modeled by so-called *meta-models* [24], which help to synthesize domain-specific modeling languages that can capture the nuances of domains they are designed to model.

When implemented properly, MIC technologies help to:

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

Early computer-aided software engineering (CASE) technologies have evolved into sophisticated tools, such as *objectiF* and *in-Step* from MicroTool and *Paradigm Plus*, *VISION*, and *COOL* from Computer Associates. This class of products has evolved over the past two decades to alleviate complexities associated with developing software for enterprise applications. Their successes have added the Model-Integrated Computing paradigm to the familiar programming languages and language processing tool offerings used by previous generations of software developers. Popular examples of MIC tools being used today include the Generic Modeling Environment (GME) [23] and Ptolemy [25] (which are used primarily in the real-time and embedded domain) and UML/XML tools based on the OMG Model Driven Architecture (MDA) [15] (used primarily in the enterprise application domain thus far).

As shown in Figure 2, MIC uses a set of tools to

- Analyze the interdependent features of the system captured in a model and

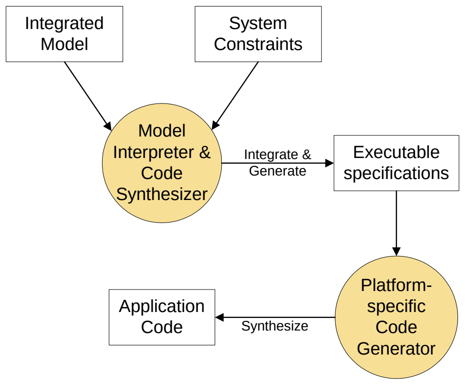

Integrated Model
System Constraints
Model Interpreter & Code Synthesizer
Integrate & Generate
Executable specifications
Platform-specific Code Generator
Synthesize
Application Code

Figure 2: **The Model-Integrated Computing Process**

- Determine the feasibility of supporting different non-functional system aspects, such as QoS requirements, in the context of the specified constraints.

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

2.3 Application Integration using Model-Integrated Computing and Component Middleware

As described above, MIC and component middleware have evolved independently from different perspectives. Although these two paradigms have achieved relatively good success independently, each also has the following limitations:

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

Lack of sophisticated modeling tools. Previous efforts at model-based development and code synthesis attempted by CASE tools generally failed to deliver on their potential for the following reasons [26]:

- They attempted to generate entire applications, including the infrastructure and the application logic, which often lead to inefficient, bloated code that was hard to optimize, validate, evolve, or integrate with legacy code.
- Due to the lack of sophisticated domain-specific languages and associated modeling tools, it was hard to achieve *round-trip engineering*, *i.e.*, moving back and forth seamlessly between model representations and the synthesized code.
- Since CASE tools and modeling languages dealt primarily with a restricted set of platforms (such as mainframes) and legacy programming languages (such as COBOL) they did not adapt well to the distributed computing paradigm that arose from advances in PC and Internet technology and newer object-oriented programming languages, such as Java, C++, and C#.

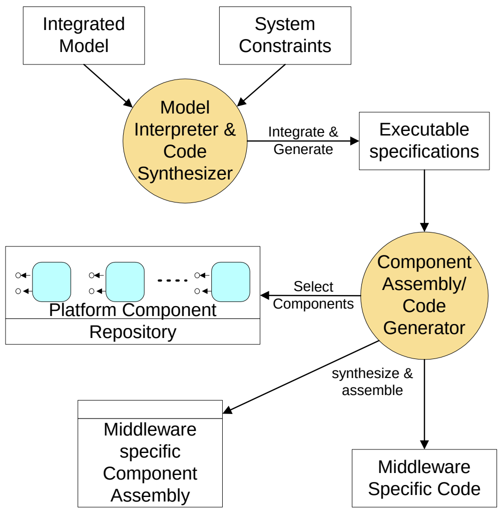

Integrated Model
System Constraints
Model Interpreter & Code Synthesizer
Integrate & Generate
Executable specifications
Component Assembly/ Code Generator
Select Components
Platform Component Repository
synthesize & assemble
Middleware specific Component Assembly
Middleware Specific Code

Figure 3: **Integrating Model-Integrated Computing and Component Middleware**

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

- Combining MIC with component middleware helps to overcome problems with earlier-generation CASE tools since it does not require the modeling tools to generate all the code. Instead, large portions of applications can be *composed* from reusable, prevalidated middleware components, as shown in Figure 3.
- Combining MIC and component middleware helps address environments where control logic and procedures change at rapid pace, by synthesizing and assembling newer extended components that implement the new procedures and processes.
- Combining component middleware with MIC helps to make middleware more flexible and robust by automating

the configuration of many QoS-critical aspects, such as concurrency, distribution, resource reservation, security, and dependability. Moreover, MIC-synthesized code can help bridge the interoperability and portability problems between different middleware for which standard solutions do not yet exist.

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

Figure 4 illustrates six points at which Model-Integrated Computing can be integrated into component middleware architectures and applied to DRE applications. We describe each of these six integration points below:

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

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

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

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

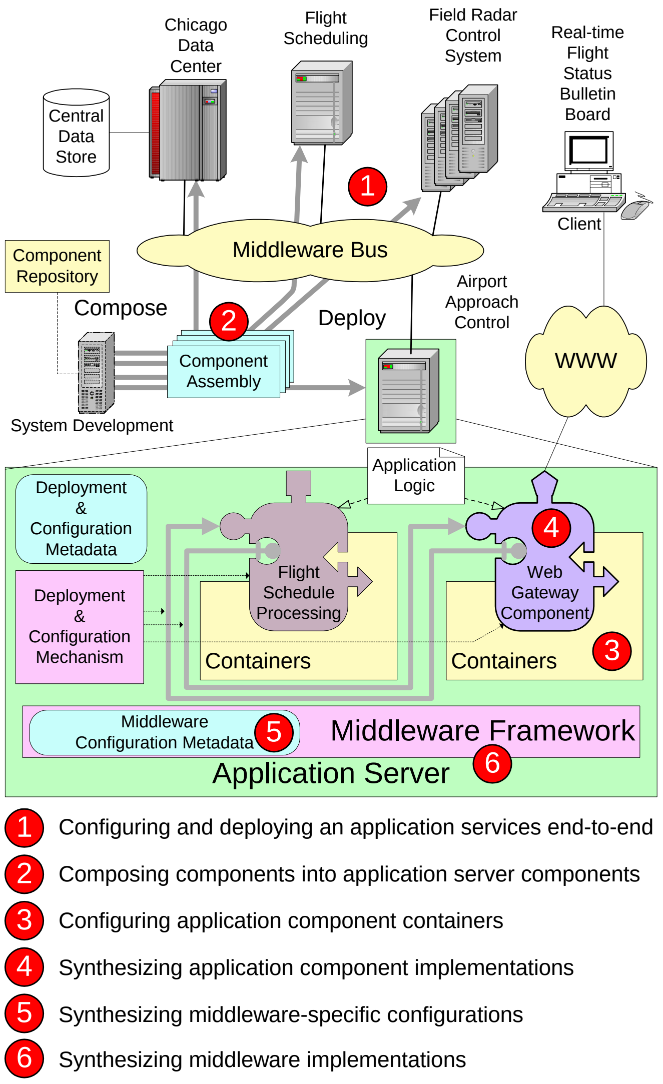

Chicago Data Center
Flight Scheduling
Field Radar Control System
Real-time Flight Status Bulletin Board
Central Data Store
Client
1
Middleware Bus
Component Repository
Compose
2
Deploy
Airport Approach Control
WWW
Component Assembly
System Development
Application Logic
Deployment & Configuration Metadata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

Figure 4: **Integrating Model-Integrated Computing with Component Middleware**

end-to-end application and QoS requirements. In particular, the *Simulink* tool allows DRE application developers to model, analyze, simulate, verify, and rapidly protoype applications.

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

- Selecting a set of suitable, semantically compatible components from reuse repositories.
- Specifying the functionality required by new components to isolate the details of DRE systems that (1) operate in environments where DRE processes change periodically

and/or (2) interface with third-party software associated with external systems.

- Determining the interconnections and interactions between components in metadata.
- Packaging the selected components and metadata into an assembly that can be deployed into the application server.

CASE tools, such as *Matlab* and *Simulink*, provide visual tools for composing DRE application servers.

3. Configuring application component containers. Application components use containers to interact with the application servers in which they are configured. Containers provide many policies that distributed applications can use to fine-tune underlying component middleware behavior, such as its security, transactional, and quality of service properties. Since DRE applications consist of many interacting components, their containers must be configured with consistent and compatible QoS policies.

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

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

The shift toward high-level design languages and modeling tools is creating an opportunity for increased automation in generating and integrating application components. The goal is to bridge the gap between specification and implementation via sophisticated aspect weavers [27] and generator tools [23] that can synthesize platform-specific code customized for specific application properties, such as resilience to equipment failure, prioritized scheduling, and bounded worst-case execution under overload conditions. Research in this area is now transitioning into commercial products that support narrow, well-defined domains, such as

- *SimuLink* and *StateFlow* from MathWorks, which generate signal processing and control applications from high-level models
- The *Reactis* product family from Reactive Systems, which provides a modeler, simulator, validator, and a code generator for embedded software systems
- *ObjecTime* from Rational, which generates call processing applications from state chart models and
- *+1Reuse* from +1 Software Engineering, which uses modeling concepts to synthesize application component integration.

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

- Various types of transports and protocols
- Various levels of fault tolerance
- Middleware initialization options
- Efficiency of (de)marshaling event parameters
- Efficiency of demultiplexing incoming method calls
- Threading models and thread priority settings and
- Buffer sizes, flow control, and buffer overflow handling

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

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

Advanced meta-programming techniques, such as adaptive and reflective middleware [28, 29, 30, 31] and aspect-oriented programming [27], are being developed to configure middleware options so they can be tailored for particular DRE application use cases.

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

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

The customizable middleware architectural framework *Quarterware* [32] falls under this category of integration. Quarterware abstracts basic middleware functionality and allows application-specific specializations and extensions. The framework can generate core facilities of CORBA, RMI, and MPI. The framework-generated code is optimized for performance, which the authors demonstrate is comparable—and often better—than many commercially available middleware implementations.

3 An Overview of the Model Driven Architecture and CORBA Component Model

The Object Management Group (OMG) has recently adopted the Model Driven Architecture (MDA) [15] to standardize the integration of MIC paradigm with component middleware technologies. This section describes how the OMG MDA can be used to develop, assemble and deploy complex DRE applications based on QoS-enabled Web services. In particular, we show how the application functionality specified as models can be used to synthesize new components that implement the web service, as well as to assemble them with semantically compatible reusable components provided by the CORBA Component Model [1] and Real-time CORBA [33]. Section 4 then presents how we are synthesizing QoS-enabled CCM applications from UML models and exposing them as Web services.

3.1 Overview of the OMG Model Driven Architecture

The OMG MDA defines standard ways to address many of the challenges facing complex applications, such as the DRE applications outlined in Section 1. The MDA builds upon years of research on model-integrated computing [14, 34, 35] to provide standard modeling notations based on the Unified Modeling Language (UML) [16]. Figure 5 illustrates the structure of the MDA.

The MDA defines platform-independent models (PIMs) and platform-specific models (PSMs) that streamline platform integration issues and protect investments against the uncertainty

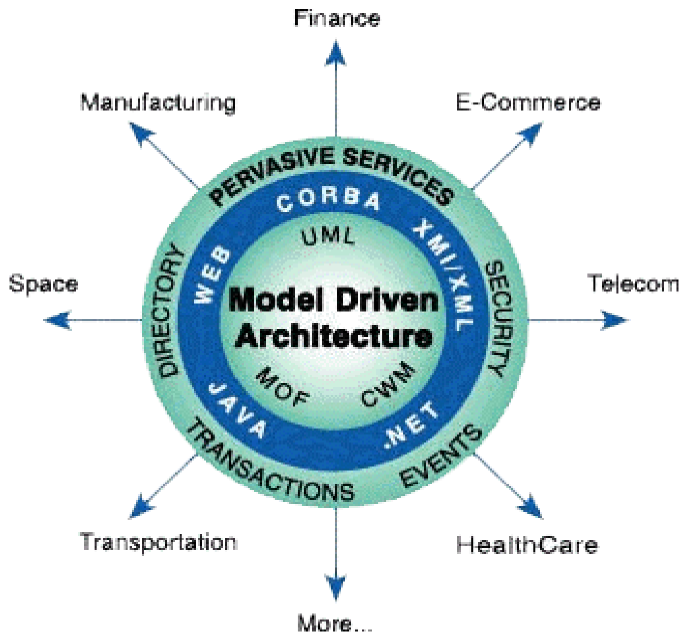


Figure 5: **Overview of the OMG Model Driven Architecture (Copyright OMG, reproduced by permission)**

of changing platform technology. These two levels of models can be differentiated as follows:

- The PIMs describe at a high-level how applications will be structured and integrated, without concern for the middleware/OS platforms or programming languages, on which they will be deployed. PIMs provide a formal definition of an application's functionality, as well as a representation of the application as a computation-independent business model or a military strategy, also referred to as a *Domain Model*. For example, a transaction in a business system or target tracking and identification in a military ISR mission, can be modeled in generically using modeling tools based on UML.
- The PSMs are so-called *constrained* formal models since they express platform-specific details. The PIM models are mapped into PSMs via translators. For example, the generic operation that is specified in the PIM could be mapped and refined to the domain-specific operation, such as information exploitation of sensor data, in the underlying Real-time CORBA platform.

Both PIM and PSM descriptions of applications are formal specifications built using modeling standards, such as UML, which can be used to model application functionality and system interactions. The MDA also defines a platform-independent meta-modeling language that allows platform-specific models to be modeled at an even higher level of abstraction.

Figure 5 also references the Meta-Object Facility (MOF), which provides a framework for managing any type of metadata. The MOF has a layered metadata architecture with a meta-meta-modeling layer and an object modeling language—closely related to UML—that ties together the meta-models and models. The MOF also provides a repository to store meta-models.

The Common Warehouse Model (CWM) provides standard interfaces that can manage many different databases and schemas throughout organizations as diverse as a military command and control system or a financial services enterprise. The CWM interfaces are designed to support management decision making and exchange of domain-specific business metadata or between diverse warehouse tools to help present a coherent picture of business conditions at a single point in time. The OMG has defined the XML metadata Interchange (XMI) for representing and exchanging CWM metamodels in XML.

The OMG defined three levels where MDA-based specifications are useful:

1. The **Pervasive services** level constitutes a suite of PIM specifications of essential CORBA services, such as events, transactions, directory, and security, that are useful for large-scale application development. Additional services may be added at a later date from the broad range of existing CORBA object services.
2. The **Domain facilities** level constitutes a suite of PIM specifications from different domains, such as defense, manufacturing, healthcare, and life science research within the OMG.
3. The **Applications** level constitutes a suite of PIM created by software developers for their applications.

The three levels outlined above allows a broad range of services and application designs to be reused across multiple platforms. For instance, some domain-specific services from the OMG could be reused for other technology platforms, such as .NET or J2EE, rather than designing them from scratch.

3.2 Overview of the CORBA Component Model

The OMG has addressed the limitations with object-oriented middleware described in Section 2.1 by defining the CORBA Component Model (CCM) [1]. CCM is modeled closely on the Enterprise Java Beans (EJB) specification. Unlike EJB, however, CCM uses the CORBA object model as its underlying object interoperability architecture and is therefore not bound to a particular programming language. CCM and CORBA are also related to the Microsoft COM family of middleware technologies. Unlike CORBA, however, Microsoft's COM was designed to support a collocated component programming model initially and later DCOM added the ability to distribute COM objects.

Figure 6 shows an overview of the run-time architecture of the CCM model. *Components* are the implementation entities

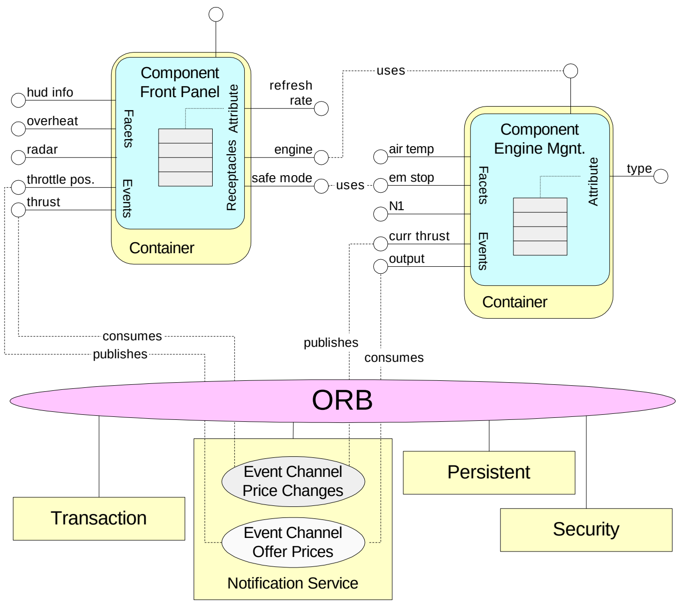


Figure 6: **Overview of the CCM Run-time Architecture**

that export a set of interfaces to clients. Components can also express their intent to collaborate with other components by defining interfaces called *ports*. There are three types of ports in CCM:

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

A *container* provides the run-time environment for a component. It contains various pre-defined hooks that provide strategies, such as persistence, event notification, transaction, and security, to the component it manages. Each container manages one type of component and is responsible for initializing this component and connecting it to other components and ORB services. Developer-specified metadata can be used to instruct the CCM deployment mechanism how to create these containers.

In addition to the building blocks outlined above, the CCM also standardizes component implementation, packaging, and deployment. The CCM Component Implementation Framework (CIF) helps generate the component implementation skeletons and persistent state management automatically using the Component Implementation Definition Language (CIDL). The CCM also extends the Open Software Description (OSD), which is a vocabulary of XML defined by W3C, to specify component packaging and assembly descriptors that is used by the CCM deployment mechanisms to configure the component connections and container configurations. The CCM deployment mechanism enables an MDA model to be synthesized to configure and deploy distributed applications, as shown by integration point 1 in Figure 4.

The CCM is an effective component middleware technology to serve as the basis for composing DRE applications using MDA and hosted as a Web service for the following reasons:

1. Metadata specifies the interconnections among components and application servers, which provides a natural mechanism for MDA to compose new end-to-end DRE application functionality and to describe Web services interface using WSDL, as shown by integration point 1 and 3 in Figure 4.
2. The well-defined virtual boundaries of components simplifies the validation of individual component functionality, as well as the functionality of Web services in DRE applications, and helps to address the challenges shown by integration points 1 and 4 in Figure 4.
3. A rich set of existing common middleware services, such as the CORBA Notification and Scheduling services, can be componentized and reused to compose complex Web services, which helps to address the challenges identified by integration points 1 and 2 in Figure 4.
4. The portability and mature QoS capabilities of Real-time CORBA enables CCM to run predictably on most OS platforms, which makes it easier to integrate and interact with legacy DRE systems written in a variety of programming languages and running on a wide range of OS platforms. Dynamically configurable CORBA implementations, such as dynamicTAO [36] and TAO [37], can help address integration points 5 and 6 in Figure 4.
5. CCM can be extended to specify component QoS requirements in the metadata. The CCM container mechanisms provide a standard interaction point to extend the QoS-related interaction between components and the ORB. These extensions provide the enabling mechanisms for QoS-aware Web services and address integration points 1, 2, and 3 in Figure 4.

Although the CCM specification has recently been finalized by the OMG, it is still not part of the Core CORBA specification. A number of CCM implementations are available based on the current draft [1], including *OpenCCM* by the Universite des Sciences et Technologies de Lille, France, *K2 Containers* by iCMG, *MicoCCM* by FPX, and *CIAO* by the DOC groups at Washington University St. Louis, Vanderbilt University, and University of California Irvine. The influence of the architectural patterns found in CCM is also evident in other popular component middleware technologies, such as J2EE [38] and .NET.

4 Resolving Distributed Application Challenges with Model-Integrated Computing and Middleware

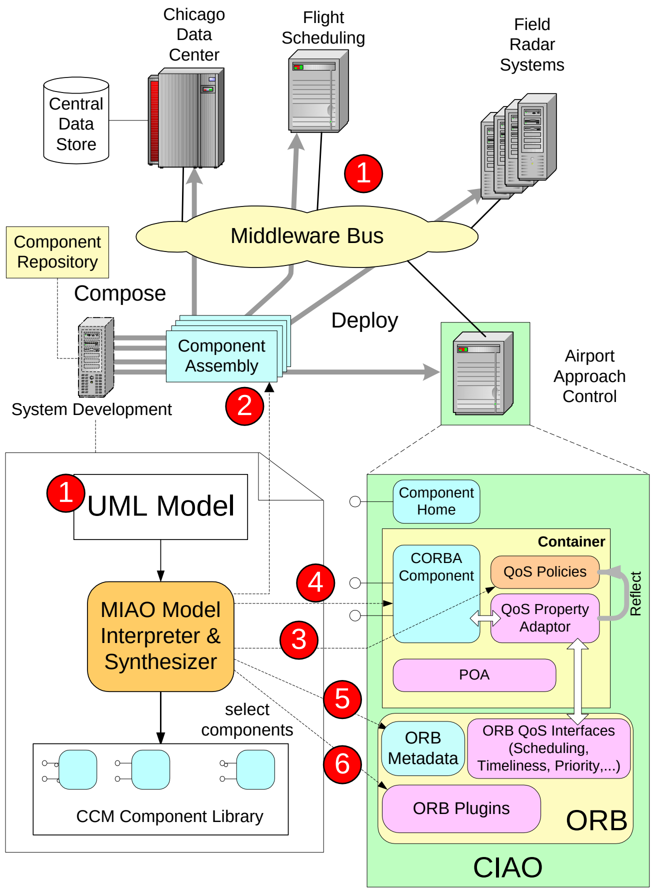
As described in Section 1, developing and deploying DRE applications and Web services using today's COTS middleware technologies requires application developers to handle the proliferation of middleware technologies, satisfy multiple QoS requirements simultaneously, and eliminate accidental complexities arising from the manual assembly of components and applications. A promising way to address these challenges is to integrate MIC modeling tools with component middleware. As discussed in Section 2.3, this integration helps developers to more effectively

- Model application functionality at a higher level of abstraction and
- Analyze and partition responsibility of application servers.

MIC tools can then be used to synthesize, assemble, and deploy the component assemblies based on the models and the feasibility analysis.

Figure 7 illustrates how we are developing the *Component Synthesis with Model Integrated Computing (CoSMIC)* toolset, which provides MDA-based tools designed to (1) *model and analyze* distributed application functionality and QoS requirements as a platform-independent model (PIM) and (2) *synthesize* CCM-specific deployment metadata, which is a platform-specific model (PSM), required to deliver end-to-end QoS. Rather than using the CORBA Interface Definition Language (IDL) as the PSM, the synthesized CCM-specific PSM in CoSMIC consists of metadata in XML format describing component compositions using existing component definitions that can be mapped to modeling entities using MDA tools. A QoS-enabled CCM provides a good target platform for MDA since multiple QoS properties required by components can be realized by modifying QoS-related meta-information.

The CoSMIC project is therefore developing synthesis tools targeted at the *Component-Integrated ACE ORB* (CIAO), which is our CCM implementation based on *The ACE ORB* (TAO) [33]. TAO is an open-source, high-performance, highly configurable Real-time CORBA ORB that implements key



1 Configuring and deploying an application services end-to-end

2 Composing components into application server components

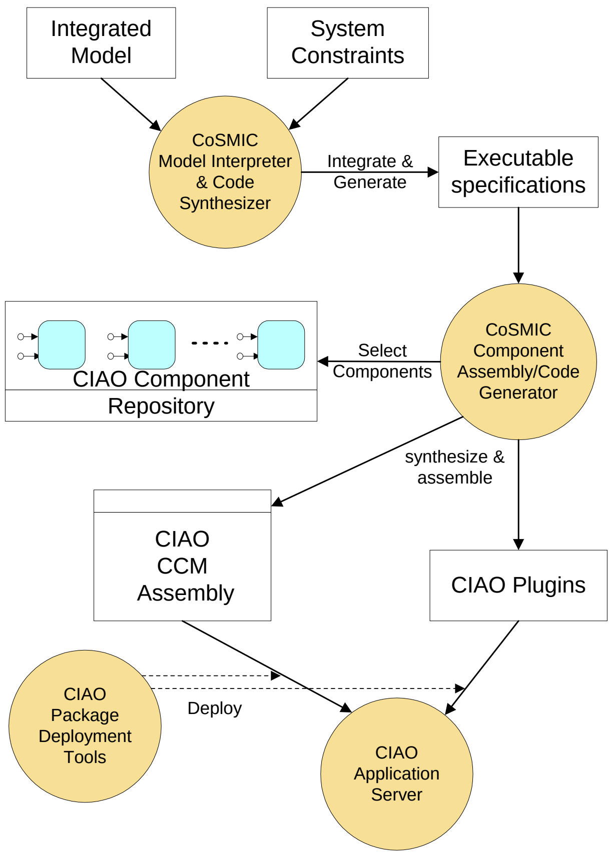
3 Configuring application component containers

4 Synthesizing application component implementations

5 Synthesizing middleware-specific configurations

6 Synthesizing middleware implementations

Figure 7: **Synthesizing CCM Middleware from MDA Tools**



Integrated Model
System Constraints
CoSMIC Model Interpreter & Code Synthesizer
Integrate & Generate
Executable specifications
CIAO Component Repository
Select Components
CoSMIC Component Assembly/Code Generator
synthesize & assemble
CIAO CCM Assembly
CIAO Plugins
CIAO Package Deployment Tools
Deploy
CIAO Application Server

Figure 8: **Interaction betwwen CoSMIC and CIAO**

patterns [39] to meet the demanding QoS requirements of distributed systems. CIAO abstracts component QoS requirements into metadata that can be specified in a component assembly after a component has been implemented. Decoupling QoS requirements from component implementations greatly simplifies the conversion and validation of an application model with multiple QoS requirements into CCM deployment of DRE applications.

The remainder of this section describes how we are combining the CoSMIC design tools and procedures with the CIAO component middleware platform to address key challenges faced by the developers of DRE applications. Figure 5 illustrates the interfaction between CoSMIC and CIAO.

Challenge 1: Reconciling Different Middleware Technologies

Problem. There are increasing number of component middleware technologies, such as CCM, J2EE, and .NET. Although each has its strengths and weaknesses, no technology provides a comprehensive *one-fits-all* solution. For example, middleware technologies, such as Sun's J2EE and the Microsoft's emerging .NET web services, may be challenged to provide a complete end-to-end solution to build distributed applications due to their dependence on an implementation language, such as Java, or a platform, such as Windows, respectively. Integrating DRE applications using the appropriate middleware via Web services allows developers to leverage technologies best suited for a particular task.

Business organizations, government agencies, and armed services have a considerable investment in legacy applications and equipment that do not use today's middleware technologies. In particular, large-scale DRE applications often consist of components and subsystems based on multiple software technologies developed over long periods of time. Although Web services provide a starting point to integrate these applications together, it is a non-trivial task since different component middleware technologies have their own unique interaction model. For example, applications can not pass CORBA object references via a Web service request since the receiver may not understand CORBA object reference semantics.

Solution. Using the MIC paradigm can help to shield DRE applications from the differences between diverse middleware technologies and avoid tightly coupling applications to specific middleware. For example, the tools offered by CoSMIC help resolve these challenges by using higher level modeling languages, such as UML [16], to model application and system behavior as indicated by integration point 1 and 2 in Figure 7. Other CoSMIC tools can be used to synthesize middleware-specific assemblies. In our work, we are initially targeting the CCM and Real-time CORBA to demonstrate concretely how MDA tools can generate and compose components and assemblies. There is no reason, however, why different middleware technologies, such as J2EE or .NET, cannot be supported since the component models are based on similar patterns.

Due to the clean separation of component implementations in the CCM, MDA provides a natural extension to compose component interconnection for the CCM. Moreover, the CCM provides the mechanisms for composing an applications by reusing existing components and middleware infrastructure. To achieve this, CoSMIC takes advantage of the deployment facility provided by CIAO. We are using this *model-first/generate-next* strategy to implement finer grained functionality of components. As shown by integration point 4 in Figure 7, we are developing tools that help model and synthesize component implementations based on the component function specification defined in higher level modeling tools.

Challenge 2: Satisfying Multiple Quality of Service (QoS) Requirements Simultaneously

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

Solution. A benefit of MDA is its ability to employ complex modeling tools that can check for certain properties of the implementation, *e.g.*, check the correctness of an algorithm or ensure that a series of constraints are enforced. Although the OMG MDA standard has adopted the UML-based PIM and PSM for CORBA, it does not yet adequately address a broad spectrum of DRE application QoS issues. In particular, it does not address the integration of priority propagation, resource allocations, dependability, and predictability that are crucial to DRE applications.

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

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

Figure 7 illustrates how CoSMIC can be used to synthesize and assemble QoS-enabled, CCM middleware for DRE applications. This synthesis uses the following iterative process to assemble and deploy QoS-enabled distributed applications:

1. **Model the overall application** using CoSMIC visual modeling tools and specify the application's QoS requirements as constraints. This step defines and partitions the functionality and QoS requirements demanded by each application module based on the overall model of the application, as described by integration point 1 of Figure 7
2. **Compose application servers** using CoSMIC application server composition tools to combine component assemblies by mixing and matching existing off-the-shelf components and partitioning or defining the functionality of new components, as needed, as shown in Point 2 of Figure 7. The metadata in a component assembly also contain QoS requirements for each components that the composition tools derived from the model.
3. **Model and synthesize components**—If new component implementations are needed from the previous step, each can be modeled by using CoSMIC's modeling tool. CoSMIC's component implementation synthesizer will generate the actual implementations based on the models, as indicated by integration point 4 of Figure 7.
4. **Validate applications** via's CoSMIC tools that check whether an application composition implements its model definitions correctly.
5. **Deploy the resulting system for testing and tuning** via tools that fine-tune CIAO's QoS requirements for assemblies. Later iterations of this process can use these adjustments as feedback to improve the overall system model.

Challenge 3: Addressing Accidental Complexities in Integrating Software Systems

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

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

The CCM specification does not yet address how to associate component QoS requirements with a component deployment. Our CCM implementation (CIAO) therefore supports the configuration of certain component QoS properties via the component deployment metadata shown by integration point 2 of Figure 7. Since we providing component QoS management services through containers in our CCM implementation[40], the synthesizing tools will also generate container configurations in a component assembly, as depicted in Point 3 of Figure 7.

To support QoS requirements that were not foreseen by the component middleware implementation, CoSMIC can also synthesize middleware modules that CIAO uses to customize its behavior to support non-native QoS supports required by other systems using the Web services. CIAO's deployment framework then uses these customized modules to configure application servers before deploying the components, as shown by integration point 6 of Figure 7. The automation of semantic propagation described here ensures that all application servers providing Web services in an integrated DRE application perform their work as specified in the overall model, without undue programmer intervention.

5 Related Work

Large-scale application integration using Web services is an increasingly popular research topic. Integrating Web services implemented with different middleware technologies is hard due to the lack of a standard interaction model for Web services [6, 13]. There are R&D efforts to define languages that describe the interaction models of Web services, such as Web Services Flow Language [41] and X-Lang [42]. Research on QoS-enabled Web Services currently focuses on either QoS properties related to business applications or the design of Web Services protocols, such as SOAP and HTTP [43]. Recently, there have been efforts on an embedded SOAP implementation, called *eSOAP* (`www.embedded.net/eSOAP`) used for data exchange in network appliances.

Our research on QoS-enabled middleware and model driven architecture extends earlier work on Model-Integrated Computing (MIC) [14, 44, 35] to model and synthesize component middleware code for DRE applications. The MIC infrastructure provides a unified software architecture and framework for creating a Model-Integrated Program Synthesis (MIPS) environment [23]. The core components of the MIC infrastructure include (1) a customizable generic model editor for creation of multiple-view, domain-specific models, (2) model databases for storage of the created models, and (3) a model interpretation technology that assists in the creation of domain-specific, application-specific model interpreters for transformation of models into executable/analyzable artifacts. The new environment is domain-specific and includes tools and functionality to support the creation and storage of system models, in addition to generation of executable/analyzable artifacts from these models.

In the MIC technology, the modeling concepts to be instantiated in the MIPS environment are specified in a meta-modeling language [24]. A metamodel of the modeling paradigm is constructed that specifies the syntax, static semantics, and the presentation semantics of the domain-specific modeling paradigm. The metamodel uses a Unified Modeling Language (UML) class diagram to capture information about the objects that are needed to represent the system information and the inter-relationships between different objects. The meta-modeling language also provides for the specification of visual presentation of the objects in the graphical model editor.

Popular examples of MIC technology being used today include GME [23] and Ptolemy [25] (which are used primarily in the real-time and embedded domain) and MDA [15] based on UML [16] and XML [17] (which is used primarily in the business domain). Our work uses the GME tool and UML modeling language to model and synthesize component middleware for use in provisioning collaborative DRE applications. In particular, we are enhancing the GME tool to produce meta-models for DRE applications, as well as developing and validating new UML profiles to support DRE applications.

To support QoS-enable Web services, the implementation of a service must be able to honor the QoS requirements that the service advertises. Our past research on building Real-time CORBA middleware to support DRE systems with stringent QoS requirements has identified key patterns [39] that we have applied to implement a highly configurable [45], QoS-enabled ORB called TAO [33, 46]. Our current research builds upon these results to provide the foundation to support the QoS-enabled CCM.

Kon and Campbell [28] apply reflective middleware techniques to extend TAO to reconfigured the ORB at run-time by dynamically linking in the required modules according to the features required by the applications. Although their research provides a proof-of-concept for dynamic configurable middleware framework, their research may not be suitable for DRE applications since dynamic loading and unloading ORB components may incur undue overhead and prevent the ORB from meeting application's QoS demands. Our work on the Component-Integrated ACE ORB (CIAO) relies upon MIC tools to analyze the required ORB components and their configurations. This approach ensures the ORB in an application server contains only the required components without compromising the predictability of the system.

Container architectures provide a useful way to apply meta-programming techniques [47] to provide QoS assurance control in component middleware, as previously identified in [40]. Containers can also apply aspect-oriented programming (AOP) [27] techniques to plug in different non-functional behaviors [48]. Pluggable ORB components can also be used to plug in QoS assurance mechanisms as CIAO will support.

The *QoSket* framework [49] developed at BBN Technologies provides reusable QoS behaviors that can be used to modify application QoS behaviors by packaging and installing these behaviors into an application. A QoSket is a collection of these behaviors that bundles QoS specifications, middleware components that implement, monitor, and control QoS properties, and application specific adaptive behaviors in one place. The QoSket is build on the *Quality Objects* (QuO) [50] distributed object computing middleware, which applies aspect-oriented programming (AOP) [27] techniques to adaptive applications running over wide-area networks. Our CIAO project provides a QoSket-like mechanism to install QoS behaviors for component middleware. CIAO behavior components are reusable by themselves, however, and can be composed along with metadata to specify the actual behaviors. By separating the behavioral metadata from the implementation, MDA tools can translate QoS constraints into specifications.

6 Concluding Remarks

Due to tight coupling between software modules, conventional methods for building distributed applications increase the time and effort required to develop and evolve the software. Moreover, many application quality aspects, such as persistent data store, security, and management of run-time resources, cut across multiple layers, which also tightly couples application software modules with the middleware infrastructure and its associated housekeeping tasks. These tight couplings yield brittle application implementation that are hard to reuse, maintain, and evolve.

One way to address these coupling issue is by refactoring common application logic into *object-oriented application frameworks* [51]. This solution has limitations, however, since application objects can still interact directly with each other, which encourages tight coupling. Moreover, framework-specific bookkeeping code is also required within the applications to manage the framework, which can tightly couple applications to the framework they are developed upon. It is therefore non-trivial to reuse application objects and port them to different frameworks.

Component middleware [22] has emerged as a promising solution to many limitations with object-oriented application frameworks. This type of middleware consists of reusable software artifacts that can be distributed or collocated throughout a network. A proliferation of component middleware technologies have emerged recently to address various requirements of distributed applications. These types of applications are increasingly being assembled from components belonging to disparate middleware technologies, which increases the effort required to integrate and deploy semantically compatible and interoperable components across multiple middleware platforms. Moreover, distributed applications must increasingly support multiple simultaneous QoS properties, such as dependability, security, and scalability.

Proliferation of incompatible middleware technologies, however, has become an impeding force against developers to take advantage of existing applications using different technologies. The emerging trend of exposing application functionality via *Web services* reduce the cost of middleware and application integration. Nonetheless, developers still face the problems outlined above when implement the services. Our solution to these problems involves combining Model-Integrated Computing with QoS-enabled component middleware to create flexible Web services that support heterogeneity, yet can be tailored readily to meet the needs of DRE applications with multiple simultaneous QoS requirements.

This paper explores the benefits of Model-Integrated Computing for developing DRE applications. Our focus is on the OMG Model Driven Architecture (MDA) standard and the CORBA Component Model (CCM). We describe how component middleware enables modeling and synthesis tools to rapidly develop, assemble, and deploy middleware and applications that possess multiple, simultaneous QoS requirements. This combination is important because it does not require the modeling tools to generate all the code. Instead, large portions of applications can be reused and/or customized from existing middleware components. These middleware components handle many critical QoS aspects, such as concurrency, distribution, transactions, security, and dependability.

We are developing a Model-Integrated Computing toolsuite called CoSMIC, which extends the popular GME modeling and synthesis tools [23] to support the development, assembly, and deployment of QoS-enabled distributed applications using component middleware. To ensure these QoS requirements can be realized in the middleware layer, we are also developing a QoS-aware CCM implementation called CIAO, which is based on our TAO Real-time CORBA ORB. CIAO allows Model-Integrated Computing tools to specify the QoS requirements of components in the accompanying metadata.

References

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

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

[3] Donald Box, *Essential COM*, Addison-Wesley, Reading, Massachusetts, 1997.

[4] 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, 2001.

[5] Francisco Curbera and Matthew Duftler and Rania Khalaf and William Nagy and Nirmal Mukhi and Sanjiva Weerawarana, "Unraveling the Web Services Web – An Introduction to SOAP, WSDL, and UDDI," *IEEE Internet Computing*, vol. 6, no. 2, pp. 86–93, March/April 2002.

[6] Steve Vinoski, "Where is Middleware?," *IEEE Internet Computing*, vol. 6, no. 2, pp. 83–85, March/April 2002.

[7] W3C, "Simple Object Access Protocol (SOAP) 1.1," `http://www.w3c.org/TR/SOAP`, May 2000.

[8] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana, "Web Services Description Language (WSDL) 1.1," `http://www.w3.org/TR/wsdl`, March 2001.

[9] Microsoft, ".NET Web Services Platform," `http://www.microsoft.com/net`.

[10] Sun Microsystems, "SUN^{TM} Open Net Environment," http://www.sun.com/sunone/index.html, 2001.

[11] IBM, "WebSphere," `http://www.ibm.com/software/info1/websphere/index.jsp`.

[12] BEA, "WebLogic," `http://www.bea.com/products/weblogic/server/index.shtml`.

[13] Steve Vinoski, "Web Services Interaction Models — Part 1: Current Practice," *IEEE Internet Computing*, vol. 6, no. 3, pp. 89–91, May/June 2002.

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

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

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

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

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

[19] IBM, "MQSeries Family," www-4.ibm.com/software/ts/mqseries/, 1999.

[20] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.6 edition, Dec. 2001.

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

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

[23] Akos Ledeczi, Arpad Bakay, Miklos Maroti, Peter Volgysei, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, pp. 44–51, November 2001.

[24] Jonathan M. Sprinkle, Gabor Karsai, Akos Ledeczi, and Greg G. Nordstrom, "The New Metamodeling Generation," in *IEEE Engineering of Computer Based Systems*, Washington DC, Apr. 2001, IEEE, p. 275.

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

[26] Paul Allen, "Model Driven Architecture," *Component Development Strategies*, vol. 12, no. 1, January 2002.

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

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

[29] Gordon S. Blair and G. Coulson and P. Robin and M. Papathomas, "An Architecture for Next Generation Middleware," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, London, 1998, pp. 191–206, Springer-Verlag.

[30] Fábio M. Costa and Gordon S. Blair, "A Reflective Architecture for Middleware: Design and Implementation," in *ECOOP'99, Workshop for PhD Students in Object Oriented Systems*, June 1999.

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

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

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

[34] David Harel and Eran Gery, "Executable Object Modeling with Statecharts," *IEEE Computer*, vol. 30, no. 7, pp. 31–42, July 1997.

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

[36] F. Kon, M. Roman, P. Liu, J. Mao, T Yamane, L. Magalhaes, and R. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," in *Proceedings of the Middleware 2000 Conference*. ACM/IFIP, Apr. 2000.

[37] Douglas C. Schmidt and Chris Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, no. 4, April 1999.

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

[39] 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.

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

[41] Frank Leymann, "Web Services Flow Language (WSFL 1.0)," `http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`, May 2001.

[42] Satish Thatte, "XLang – Web Services for Business Process Design," `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/`, May 2001.

[43] Anbazhagan Mani and Arun Nagarajan, "Understanding quality of service for Web services," `http://www.ibm.com/developerworks/library/ws-quality.html`, Jan 2002.

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

[45] Douglas C. Schmidt and Chris Cleeland, "Applying a Pattern Language to Develop Extensible ORB Middleware," in *Design Patterns in Communications*, Linda Rising, Ed. Cambridge University Press, 2000.

[46] Douglas C. Schmidt 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.

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

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

[49] Richard Schantz, Joseph Loyall, Michael Atighetchi, and Partha Pal, "Packaging Quality of Service Control Behaviors for Reuse," in *Proceedings of the* 5^{th} *IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, Crystal City, VA, April/May 2002, IEEE/IFIP.

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

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