# Applying Model-Integrated Computing to Component Middleware and Enterprise Applications

Aniruddha Gokhale
a.gokhale@vanderbilt.edu

Douglas C. Schmidt
schmidt@uci.edu

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

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

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

Dept. of Computer Science

Washington University
One Brookings Drive
St. Louis, MO 63130

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

## 1 Introduction

The term *enterprise application* applies to 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. These types of applications constitute the majority of worldwide information technology (IT) investment and staffing, with annual expenditures expected to exceed $7.3 billion US dollars by 2004 and with millions of IT professionals employed.

Enterprise applications historically ran on mainframes and were developed using custom built in-house applications and proprietary systems, such as *HighExPlus*, *BancsConnect*, and *EX*. Due to deregulation, time-to-market pressures, and stiff global competition for human and economic resources, however, enterprise applications increasingly run on servers and PC's and are developed using commercial-off-the-shelf (COTS) component middleware. Component middleware encapsulates specific services or sets of services to provide reusable building blocks that can be composed to develop enterprise applications more rapidly and robustly than those built entirely from scratch. In particular, component middleware offers enterprise application developers the following reusable capabilities:

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

Examples of COTS component middleware include the Common Object Request Broker Architecture (CORBA) (`www.omg.org`), Java 2 Enterprise Edition (J2EE) (`java.sun.com/j2ee`), and emerging web services middleware, such as .NET (`www.microsoft.com/net/default.asp`) and ONE (`wwws.sun.com/software/sunone/index.html`), based on XML (`www.w3c.org/XML`) and SOAP (`www.w3c.org/2000/xp/Group/`).

Despite advances in the ubiquity and quality of component middleware, however, developers of enterprise applications still face the following challenges:

**Proliferation of middleware technologies.** Large-scale, long-lived enterprise applications require component middleware platforms to work with heterogeneous platforms and languages, interface with legacy code written in different languages, and interoperate with multiple technologies from many suppliers. However, COTS component middleware technologies do not yet provide complete end-to-end solutions that support enterprise application development in diverse environments.

**Satisfying multiple quality of service requirements simultaneously.** An increasing number of enterprise applications, such as high-volume e-commerce systems and automated stock trading systems, have stringent quality of service (QoS) demands, such as efficiency, scalability, dependability, and security, that must be satisfied simultaneously and that cross-cut multiple layers and require end-to-end enforcement. Conventional implementations of component middleware cannot en-

force complex QoS requirements of enterprise applications effectively, however, since they were designed for applications with less stringent requirements.

**Accidental complexities in assembling components.** To reduce lifecycle costs and time-to-market, application developers are attempting to assemble and deploy enterprise applications 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. Application developers therefore spend non-trivial amounts of time debugging problems associated with the selection of incompatible strategies and components.

A promising way to address the challenges described above is to apply *Model-Integrated Computing* (MIC) technologies [1]. MIC is a paradigm for expressing application functionality and QoS requirements at higher levels of abstraction than is possible with programming languages like Visual Basic, Java, C++, or C#. In the context of enterprise applications, MIC tools can be applied to

1. Analyze different—but interdependent—characteristics of system behavior, such as scalability, safety, and security. Tool-specific model interpreters translate the information specified by models into the input format expected by analysis tools. These tools 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 enterprise application properties, such as isolation levels of a transaction, recovery strategies to handle various runtime failures, and authentication and authorization strategies modeled at a higher level of abstraction.

Understanding how to integrate MIC and component middleware is essential to resolve the configuration, management, and deployment challenges of enterprise applications described above. The remainder of this paper 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 key challenges associated with developing enterprise applications.

# 2 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 [2]. 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.

Various technologies, such as OSF's Distributed Computing Environment (DCE) (`www.opengroup.org/dce`), IBM's MQ Series (`www-3.ibm.com/software/ts/mqseries/`), and CORBA, 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.

**Limitations with object-oriented middleware.** The Object Management Architecture (OMA) in the CORBA 2.x specifications [3] 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 [4]:

- **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 large-scale enterprise 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 a transaction 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 the past several years, *component middleware* [5] 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 enterprise applications today include the CORBA Component Model [6], J2EE, and emerging web services middleware, such as .NET and ONE, that are based on XML and SOAP.

# 3 Overview of Model-Integrated Computing

Model-Integrated Computing (MIC) [1] is a development paradigm that systematically applies domain-specific modeling languages to engineer computing systems ranging from small-scale real-time embedded systems to large-scale enterprise applications. MIC provides rich, domain-specific modeling environments, including model analysis and model-based program synthesis tools [7]. 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*, which help to synthesize domain-specific modeling languages that can capture the nuances of domains they are designed to model.

Various technologies, such as Computer-Aided Software Engineering (CASE) and related OOA/OOD approaches by Yourdon and others, 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 being used today include the Generic Modeling Environment (GME) [7] and Ptolemy [8] (which are used primarily in the real-time and embedded domain) and UML/XML tools based on the OMG Model Driven Architecture (MDA) [9] (used primarily in the business domain thus far).

As shown in Figure 1, MIC uses a set of tools to analyze the interdependent features of the system captured in a model and determine the feasibility of supporting different 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 in turn can be used to synthesize application software.

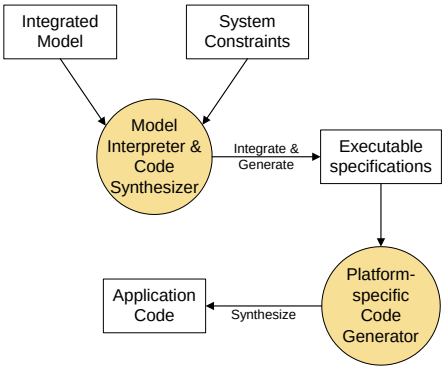


Figure 1: **The Model-Integrated Computing Process**

# 4 Combining Model-Integrated Computing with Component Middleware

As described in the previous two sections, MIC and component middleware have evolved independently from different perspectives. Although these two paradigms have achieved good success independently, each 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 specification, such as CORBA, J2EE, and .NET, and for each target deployment, such as the various OS, network, and hardware configurations. Developing, assembling, validating, and evolving *all* this middleware manually is costly, time-consuming, tedious, and error-prone, particularly for runtime platform variations and complex application use-cases. This problem is getting worse as more middleware, target platforms, and complex enterprise 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 [10]:

- 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

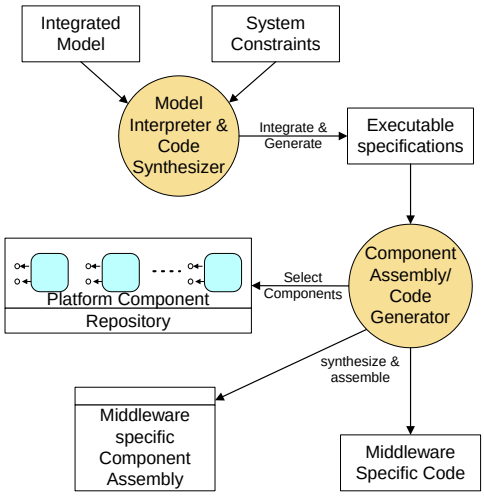


Figure 2: **Combining Model-Integrated Computing and Component Middleware**

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#.

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 2.
- Combining MIC and component middleware helps address environments where business procedures and rules change at rapid pace, by synthesizing and assembling newer extended components that conform to new business rules.
- 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, transactions, 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.

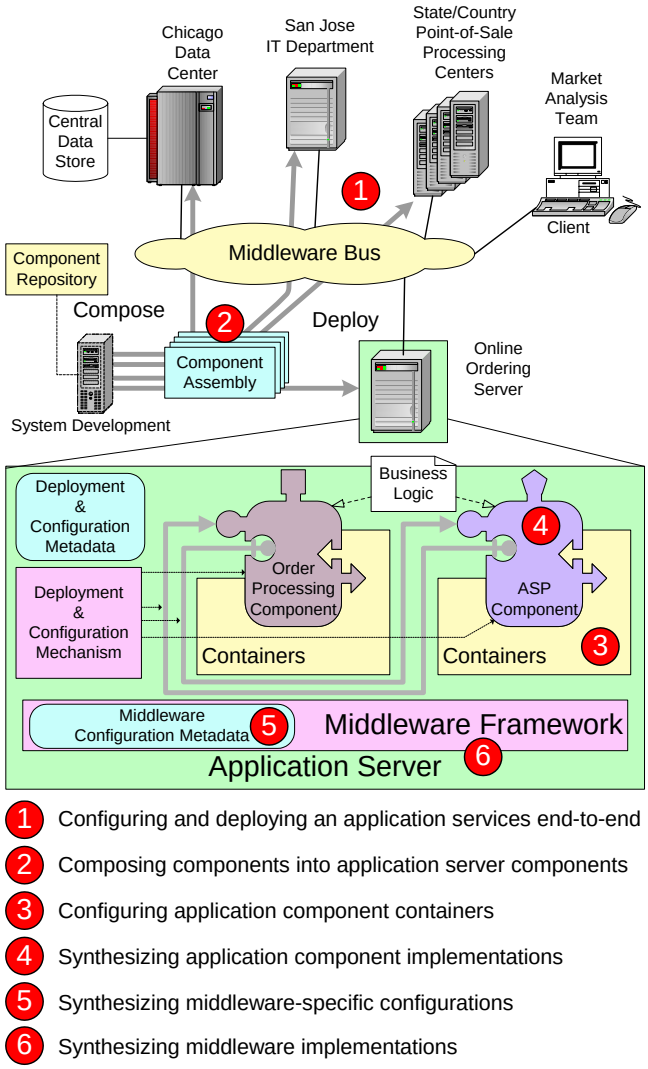


Figure 3: **Integration Points for Model-Integrated Computing and Component Middleware**

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

Figure 3 illustrates six points at which Model-Integrated Computing can be integrated into component middleware architectures. We describe each of these six integration points below:

**1. Configuring and deploying application services end-to-end.** Developing large-scale enterprise applications requires application developers to handle a variety of configuration and deployment challenges, such as

- Locating the appropriate existing services
- Partitioning and distributing business processes 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 business 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 application services end-to-end can help application 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 *ArcStyler*, *objectiF*, *case/4/0* and *Dezign for Databases* allow application developers to express their end-to-end application architecture graphically.

**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 business systems that (1) operate in environments where business processes change periodically and/or (2) interface with third-party software associated with external information 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 *objectIf* and *ArcStyler* provide visual tools for composing 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 enterprise applications can use to fine-tune underlying component middleware behavior, such as its security, transactional, and quality of service properties. Since enterprise applications consist of many interacting components, their containers must be configured with consistent and compatible policies.

Due to the number of policies and the intricate interactions among them, it is tedious and error-prone for an 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 enterprise applications today involves programming new components that add application-specific functionality. Likewise, new components must be programmed to interact with external information systems, such as supplier ordering systems, that are not internal to the application. Since these components involve substantial knowledge of application domain concepts, such as government regulations, business rules, organizational structure, and legacy systems, it would be ideal if they could be developed in conjunction with end-users or business 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 [11] and generator tools that can synthesize platform-specific code customized for specific application properties, such as resilience to denial of service attacks, robust behavior under heavy load, and good performance for normal load.

**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.

Advanced meta-programming techniques, such as reflection [12] and aspect-oriented programming [11], are being developed to configure middleware options so they can be tailored for particular 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.

# 5 Concluding Remarks

Due to tight coupling between software modules, conventional methods for building enterprise 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 enterprise applications that are hard to reuse, maintain, and evolve.

*Component middleware* 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 enterprise 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, enterprise applications must increasingly support multiple simultaneous QoS properties, such as dependability, security, and scalability.

This paper describes a solution to these problems that involves combining Model-Integrated Computing (MIC) with component middleware. 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 MIC toolsuite called CoSMIC (`deuce.doc.wustl.edu/CoSMIC`), which extends the popular GME modeling and synthesis tools [7] to support the development, assembly, and deployment of QoS-enabled enterprise 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. CIAO allows MIC tools to specify these QoS requirements of components in the accompanying metadata.

# References


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

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

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

[4] 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.

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

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

[7] 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, Nov. 2001.

[8] 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.

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

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

[11] 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.

[12] 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.