Middleware for Communications

Contents

Preface

1

Model Driven Middleware

Aniruddha Gokhale
Douglas C. Schmidt
Balachandran Natarajan

Institute for Software
Integrated Systems
Vanderbilt University
Box 1829, Station B
Nashville, TN 37235, USA

Jeff Gray

Dept. of Computer and
Information Science
University of Alabama
1300 University Blvd.
Birmingham, AL 35294, USA

Nanbor Wang

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

Middleware for Communications. Edited by Qusay H. Mahmoud

1.1 Introduction

Emerging trends and technology challenges. A growing number of computing resources are being expended to control distributed real-time and embedded (DRE) systems, including medical imaging, patient monitoring equipment, commercial and military aircraft and satellites, automotive braking systems, and manufacturing plants. Mechanical and human control of these systems are increasingly being replaced by DRE software controllers (Ogata 1997). The real-world processes controlled by these DRE applications introduce many challenging quality of service (QoS) constraints, including

- **Real-time requirements**, such as low latency and bounded jitter
- **High availability requirements**, such as fault propagation/recovery across distribution boundaries and
- **Physical requirements**, such as limited weight, power consumption, and memory footprint.

DRE software is generally harder to develop, maintain, and evolve (Joseph K. Cross and Patrick Lardieri 2001; Sharp 1998) than mainstream desktop and enterprise software due to conflicting QoS constraints, *e.g.*, bounded jitter vs. fault tolerance and high-throughput vs. minimal power consumption.

The tools and techniques used to develop DRE applications have historically been highly specialized. For example, DRE applications have traditionally been scheduled using fixed-priority periodic algorithms (Liu and Layland 1973), where time is divided into a sequence of identical frames at each processor and the processor executes each task for a uniform interval within each frame. DRE applications also often use frame-based interconnects, such as 1553, VME, or TTCAN buses where the traffic on an interconnect is scheduled at system design time to link the processors. Moreover, highly specialized platforms and protocols, such as cyclic executives (Locke 1992) and time-triggered protocols (Kopetz 1997), have been devised to support the development of DRE applications.

Highly specialized technologies have been important for developing traditional real-time and embedded systems, such as statically scheduled single-processor avionics mission computing systems (Harrison et al. 1997). These special-purpose technologies often do not scale up effectively, however, to address the needs of the new generation of large-scale DRE systems, such as air traffic and power grid management, which are inherently network-centric and dynamic. Moreover, as DRE applications grow in size and complexity, the use of highly specialized technologies can make it hard to adapt DRE software to meet new functional or QoS requirements, hardware/software technology innovations, or emerging market opportunities.

A candidate solution: QoS-enabled component middleware. During the past decade, a substantial amount of R&D effort has focused on developing *QoS-enabled component middleware* as a means to simplify the development and reuse of DRE applications. As shown in Figure 1.1, QoS-enabled component middleware is systems software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware and is responsible for providing the following capabilities:

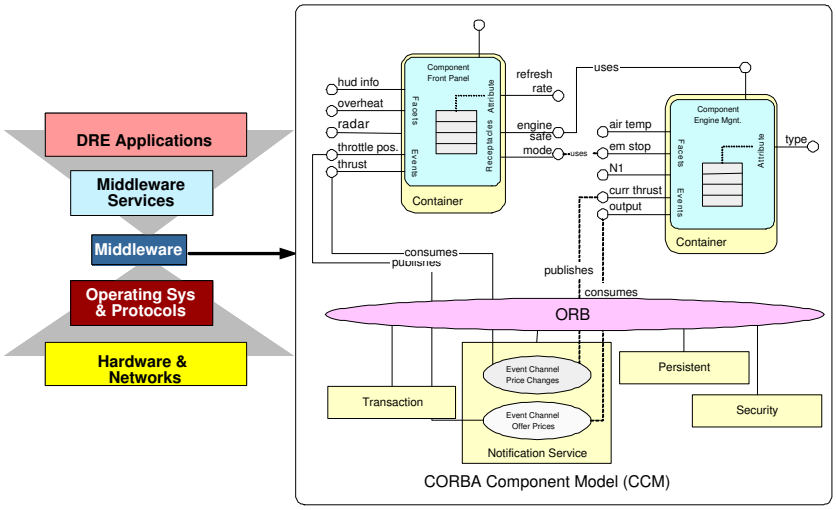

Figure 1.1 **Component Middleware Layers and Architecture**

(i) **Control over key end-to-end QoS properties.** One of the hallmarks of DRE applications is their need for strict control over the end-to-end scheduling and execution of CPU, network, and memory resources. QoS-enabled component middleware is based on the expectation that QoS properties will be developed, configured, monitored, managed, and controlled by a different set of specialists (such as middleware developers, systems engineers, and administrators) and tools than those responsible for programming the application functionality in traditional DRE systems.

(ii) **Isolation of DRE applications from the details of multiple platforms.** Standards-based QoS-enabled component middleware defines a communication model that can be implemented over many networks, transport protocols, and OS platforms. Developers of DRE applications can therefore concentrate on the application-specific aspects of their systems and leave the communication and QoS-related details to developers of the middleware.

(iii) **Reduction of total ownership costs.** QoS-enabled component middleware defines crisp boundaries between the components in the application, which reduces dependencies and maintenance costs associated with replacement, integration, and revalidation of components. Likewise, core components of component architectures can be reused, thereby helping to further reduce development, maintenance, and testing costs.

A companion chapter in this book (Wang et al. 2003b) examines how recent enhancements to standard component middleware – particularly Real-time CORBA (Obj 2002b) and the CORBA Component Model (Obj 2002a) – can simplify the development of DRE applications by composing static QoS provisioning policies and dynamic QoS provisioning behaviors and adaptation mechanisms into applications.

Unresolved challenges. Despite the significant advances in QoS-enabled component middleware, however, applications in important domains (such as large-scale DRE systems) that require simultaneous support for multiple QoS properties are still not well supported. Examples include shipboard combat control systems (Schmidt et al. 2001) and supervisory control and data acquisition (SCADA) systems that manage regional power grids. These types of large-scale DRE applications are typified by the following characteristics:

- **Stable applications and labile infrastructures** – Most DRE systems have a longer life than commercial systems (Cross and Schmidt 2002). In the commercial domain, for instance, it is common to find applications that are revised much more frequently than their infrastructure. The opposite is true in many large-scale DRE systems, where the application software must continue to function properly across decades of technology upgrades. As a consequence, it is important that the DRE applications interact with the changing infrastructure through well-managed interfaces that are *semantically stable*. In particular, if an application runs successfully on one implementation of an interface, it should behave equivalently on another version or implementation of the same interface.
- **End-to-end timeliness and dependability requirements** – DRE applications have stringent timeliness (*i.e.,* end-to-end predictable time guarantees) and dependability requirements (Rajkumar et al. 1998). For example, the timeliness in DRE systems is often expressed as an upper bound in response to external events, as opposed to enterprise systems where it is expressed as events-per-unit time. DRE applications generally express the dependability requirements as a probabilistic assurance that the requirements will be met, as opposed to enterprise systems, which express it as availability of a service.
- **Heterogeneity** – Large-scale DRE applications often run on a wide variety of computing platforms that are interconnected by different types of networking technologies with varying performance properties. The efficiency and predictability of execution of the different infrastructure components on which DRE applications operate varies based on the type of computing platform and interconnection technology.

Despite the advantages of QoS-enabled component middleware, the unique requirements of large-scale DRE applications described earlier require a new generation of sophisticated tools and techniques for their development and deployment. Recent advances in QoS-enabled component middleware technology address many requirements of DRE applications, such as heterogeneity and timeliness. However, the remaining challenges discussed below impede the rapid development, integration, and deployment of DRE applications using COTS middleware:

i. Accidental complexities in identifying the right middleware technology. Recent improvements in middleware technology and various standardization efforts, as well as market and economical forces, have resulted in a multitude of middleware stacks, such as those shown in Figure 1.2. This heterogeneity often makes it hard, however, to identify the right middleware for a given application domain. Moreover, there exist limitations on how much application code can be factored out as reusable patterns and components in various layers for each middleware stack.

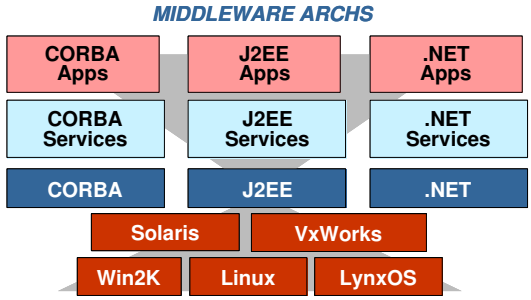

Figure 1.2 **Multiple Middleware Stacks**

This limit on refactoring in turn affects the optimization possibilities that can be implemented in different layers of the middleware. The challenge for DRE application developers is thus to choose the right middleware technology that can provide the desired levels of end-to-end QoS.

ii. Accidental complexities in configuring middleware. In QoS-enabled component middleware, both the components and the underlying component middleware framework may have a large number of configurable attributes and parameters that can be set at various stages of development lifecycle, such as composing an application or deploying an application in a specific environment. It is tedious and error-prone, however, to manually ensure that all these parameters are semantically consistent throughout an application. Moreover, such *ad hoc* approaches have no formal basis for validating and verifying that the configured middleware will indeed deliver the end-to-end QoS requirements of the application. An automated and rigorous tool-based approach is therefore needed that allows developers to formally analyze application QoS requirements and then synthesize the appropriate set of configuration parameters for the application components and middleware.

iii. Accidental complexities in composing and integrating software systems. Composing an application from a set of components with syntactically consistent interface signatures simply ensures they can be connected together. To function correctly, however, collaborating components must also have compatible semantics and invocation protocols, which are hard to express via interface signatures alone. *Ad hoc* techniques for determining, composing, assembling, and deploying the right mix of semantically compatible, QoS-enabled COTS middleware components do not scale well as the DRE application size and requirements increase. Moreover, *ad hoc* techniques, such as manually selecting the components, are often tedious, error-prone, and lack a solid analytical foundation to support verification and validation.

iv. Satisfying multiple QoS requirements simultaneously. DRE applications often possess multiple QoS requirements that the middleware must help to enforce simultaneously. Due to the uniqueness and complexity of these QoS requirements, the heterogeneity of the environments in which they are deployed, and the

need to interface with legacy systems and data, it is infeasible to develop a one-size-fits all middleware solution that can address these requirements. Moreover, it is also hard to integrate highly configurable, flexible, and optimized components from different providers while still ensuring that application QoS requirements are delivered end-to-end.

v. Lack of principled methodologies to support dynamic adaptation capabilities. To maintain end-to-end QoS in dynamically changing environments, DRE middleware needs to be adaptive. Adaptation requires instrumenting the middleware to reflect upon the runtime middleware, operating systems, and network resource usage data and adapting the behavior based on the collected data. DRE application developers have historically defined middleware instrumentation and program adaptation mechanisms in an *ad hoc* way and used the collected data to maintain the desired QoS properties. This approach creates a tight coupling between the application and the underlying middleware, while also scattering the code that is responsible for reflection and adaptation throughout many parts of DRE middleware and applications, which makes it hard to configure, validate, modify, and evolve complex DRE applications consistently.

To address the challenges described above, we need principled methods for specifying, programming, composing, integrating, and validating software throughout these DRE applications. These methods must enforce the physical constraints of the system. Moreover, they must satisfy stringent functional and systemic QoS requirements within an entire system. What is required is a set of standard integrated tools that allow developers to specify application requirements at higher levels of abstraction than that provided by low-level mechanisms, such as conventional general-purpose programming languages, operating systems, and middleware platforms. These tools must be able to analyze the requirements and synthesize the required metadata that will compose applications from the right set of middleware components.

A promising solution: Model Driven Middleware. A promising way to address the DRE software development and integration challenges described above is to develop *Model Driven Middleware* by combining the Object Management Group (OMG)'s *Model Driven Architecture* (MDA) technologies (Allen 2002; Obj 2001a) with *QoS-enabled component middleware* (de Miguel 2002; Ritter et al. 2003; Wang et al. 2003a,b). The OMG MDA is an emerging paradigm for expressing application functionality and QoS requirements at higher levels of abstraction than is possible using conventional third-generation programming languages, such as Visual Basic, Java, C++, or C#. In the context of DRE middleware and applications, MDA tools can be applied to:

(a) **Model** different functional and systemic properties of DRE applications in separate platform-independent models. Domain-specific aspect model weavers (Gray et al. 2003) can integrate these different models into composite models that can be further refined by incorporating platform-specific aspects.

(b) **Analyze** different—but interdependent—characteristics and requirement of application behavior specified in the models, such as scalability, predictability, safety, schedulability, and security. Tool-specific model interpreters (Ledeczi et al. 2001) translate the information specified by models into the input format

expected by model checking and analysis tools (Hatcliff et al. 2003; Stankovic et al. 2003). These tools can check whether the requested behavior and properties are feasible given the specified application and resource constraints.

(c) **Synthesize** platform-specific code and metadata that is customized for particular component middleware and DRE application properties, such as end-to-end timing deadlines, recovery strategies to handle various runtime failures in real-time, and authentication and authorization strategies modeled at a higher level of abstraction than that provided by programming languages (such as C, C++, and Java) or scripting languages (such as Perl and Python).

(d) **Provision** the application by assembling and deploying the selected application and middleware components end-to-end using the configuration metadata synthesized by the MDA tools.

The initial focus of MDA technologies were largely on enterprise applications. More recently, MDA technologies have emerged to customize QoS-enabled component middleware for DRE applications, including aerospace (Aeronautics 2003), telecommunications (Networks 2003), and industrial process control (Railways 2003). This chapter describes how MDA technologies are being applied to QoS-enabled CORBA component middleware to create Model Driven Middleware frameworks.

Chapter organization. The remainder of this chapter is organized as follows: Section 1.2 presents an overview of the OMG Model Driven Architecture (MDA) effort; Section 1.3 describes how the Model Driven Middleware paradigm, which is an integration of MDA and QoS-enabled component middleware, resolves key challenges associated with DRE application integration; Section 1.4 provides a case study of applying Model Driven Middleware in the context of real-time avionics mission computing; Section 1.5 compares our work on Model Driven Middleware with related efforts; and Section 1.6 presents concluding remarks.

1.2 Overview of the OMG Model Driven Architecture (MDA)

The OMG has adopted the Model Driven Architecture (MDA) shown in Figure 1.3 to standardize the integration of the modeling, analysis, simulation and synthesis paradigm with different middleware technology platforms. MDA is a development paradigm that applies domain-specific modeling languages systematically to engineer computing systems, ranging from small-scale real-time and embedded systems to large-scale distributed enterprise applications. It is *model driven* because it uses models to direct the course of understanding, design, construction, deployment, operation, maintenance, and modification. MDA is a key step forward in the long road of converting the art of developing software into an engineering process. This section outlines the capabilities and benefits of OMG's MDA.

1.2.1 Capabilities of the MDA

The OMG MDA approach is facilitated by domain-specific modeling environments (van Deursen et al. 2002), including model analysis and model-based program synthesis

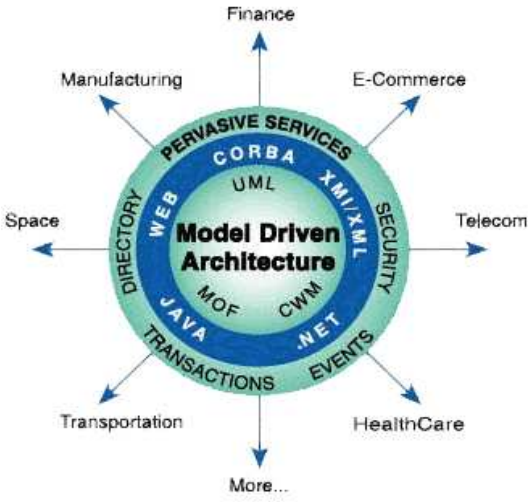

Figure 1.3 **Roles and Relationships in OMG Model Driven Architecture**

tools (Ledeczi et al. 2001). In the MDA paradigm, application developers capture integrated, end-to-end views of entire applications in the form of models, including the interdependencies of components. Rather than focusing on a single custom application, the models may capture the essence of a *class* of applications in a particular domain. MDA also allows domain-specific modeling languages to be formally specified by *metamodels* (Karsai et al. 2003; Sprinkle et al. 2001).

A metamodel defines the abstract and concrete syntax, the static semantics (*i.e.*, well-formedness rules), and semantic mapping of the abstract syntax into a semantic domain for a domain-specific modeling language (DSML), which can be used to capture the essential properties of applications. The (abstract) metamodel of a DSML is translated into a (concrete) domain-specific modeling paradigm, which is a particular approach to creating domain models supported by a custom modeling tool. This paradigm can then be used by domain experts to create the models and thus the applications.

The MDA specification defines the following types of models that streamline platform integration issues and protect investments against the uncertainty of changing platform technology:

- **Computation-independent models (CIMs)** that describe the computation independent viewpoint of an application. CIMs provide a domain-specific model that uses vocabulary understandable to practitioners of a domain and hides non-essential implementation details of the application from the domain experts and systems engineers. The goal of a CIM is to bridge the gap between (1) domain experts who have in-depth knowledge of the subject matter, but who are not

generally experts in software technologies, and (2) developers who understand software design and implementation techniques, but who are often not domain experts. For example, the Unified Modeling Language (UML) can be used to model a GPS guidance system used in avionics mission computing (Sharp and Roll 2003) without exposing low-level implementation artifacts.

- **Platform-independent models (PIMs)** that describe at a high-level how applications will be structured and integrated, without concern for the target middleware/OS platforms or programming languages on which they will be deployed. PIMs provide a formal definition of an application's functionality implemented on some form of a virtual architecture. For example, the PIM for the GPS guidance system could include artifacts such as priority of the executing thread or worst-case execution time for computing the coordinates.
- **Platform-specific models (PSMs)** that are *constrained* formal models that express platform-specific details. The PIM models are mapped into PSMs via *translators*. For example, the GPS guidance system that is specified in the PIM could be mapped and refined to a specific type in the underlying platform, such as a QoS-enabled implementation (Wang et al. 2003b) of the CORBA Component Model (CCM) (Obj 2002a).

The CIM, PIM, and PSM descriptions of applications are formal specifications built using modeling standards, such as UML, that can be used to model application functionality and system interactions. The MDA enables the application requirements captured in the CIMs to be traced to the PIMs/PSMs and vice versa. The MDA also defines a platform-independent metamodeling language, which is also expressed using UML, that allows platform-specific models to be modeled at an even higher level of abstraction.

Figure 1.3 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-metamodeling layer and an object modeling language—closely related to UML—that ties together the metamodels and models. The MOF also provides a repository to store metamodels.

The Common Warehouse Model (CWM) shown in Figure 1.3 provides standard interfaces that can manage many different databases and schemas throughout an enterprise. The CWM interfaces are designed to support management decision making and exchange of business metadata 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 using the Extended Markup Language (XML).

The OMG partitions the architecture of a computing system into the following three levels where MDA-based specifications are applicable:

1. The **Pervasive services** level constitutes a suite of PIM specifications of essential services, such as events, transactions, directory and security, useful for large scale application development.
2. The **Domain facilities** level constitutes a suite of PIM specifications from different domains such as manufacturing, healthcare and life science research within the OMG.

3. The **Applications** level constitutes a suite of PIM specifications created by software vendors for their applications.

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

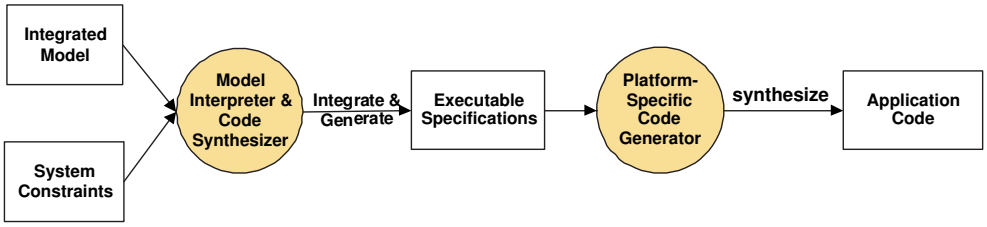

Figure 1.4 **The Model Driven Architecture Computing Process**

As shown in Figure 1.4, MDA uses a set of tools to

- Model the application in a computation-independent and platform-independent style
- Analyze the interdependent features of the system captured in a model and
- Determine the feasibility of supporting different system quality aspects, such as QoS requirements, in the context of the specified constraints.

Another set of tools translate the PIMs into PSMs. As explained in Section 1.2.1, PSMs are executable specifications that capture the platform behavior, constraints, and interactions with the environment. These executable specifications can in turn be used to synthesize various portions of application software.

1.2.2 Benefits of the MDA

When implemented properly, MDA technologies help to:

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

Earlier generations of 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 various complexities associated with developing software for enterprise applications. Their successes have added the MDA paradigm to the familiar programming languages and language processing tool offerings used by previous generations of software developers. Popular examples of MDA or MDA-related tools being used today include the Generic Modeling Environment (GME) (Ledeczi et al. 2001), Ptolemy (Buck et al. 1994), and MDA-based UML/XML tools, such as *Codagen Architect* or *Metanology*.

As described in Section 1.2.1, MDA is a platform-independent technology that aims to resolve the complexities involved with COTS obsolescence and the resulting application transition to newer technologies. The design and implementation of MDA tools for a given middleware technology, however, is itself a challenging problem that is not completely addressed by the standards specifications. The next section addresses these challenges by describing how MDA technology can be effectively combined with QoS-enabled middleware to create *Model Driven Middleware*.

1.3 Overview of Model Driven Middleware

Although rapid strides in QoS-enabled component middleware technology have helped to resolve a number of DRE application development challenges, Section 1.1 highlighted the unresolved challenges faced by DRE application developers. It is in this context that the OMG's Model Driven Architecture can be effectively combined with these QoS-enabled component middleware technologies to resolve these challenges. We coined the term *Model Driven Middleware* to describe integrated suites of MDA tools that can be applied to the design and runtime aspects of QoS-enabled component middleware.

This section first outlines the limitations of prior efforts to use modeling and synthesis techniques for lifecycle management of large-scale applications, including DRE applications. These limitations resulted from the lack of integration between modeling techniques and QoS-enabled middleware technologies. We then describe how to effectively integrate MDA with QoS-enabled middleware.

1.3.1 Limitations of Using Modeling and Middleware in Isolation

Earlier efforts in model driven synthesis of large-scale applications and component middleware technologies have evolved from different perspectives, *i.e.*, modeling tools have largely focused on design issues (such as structural and behavioral relationships and associations), whereas component middleware has largely focused on runtime issues (such as (re)configuration, deployment, and QoS enforcement). Although each of these two paradigms have been successful independently, each also has its limitations, as discussed below:

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

Lack of sophisticated modeling tools. Previous efforts at model-based development and code synthesis attempted by CASE tools generally failed to deliver on their potential for the following reasons (Allen 2002):

- They attempted to generate entire applications, including the middleware infrastructure and the application logic, which often led to inefficient, bloated code that was hard to optimize, validate, evolve, or integrate with existing code.
- Due to the lack of sophisticated domain-specific languages and associated metamodeling 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#.

1.3.2 Combining Model Driven Architecture and QoS-enabled Component Middleware

The limitations with modeling techniques and component middleware outlined above can largely be overcome by integrating OMG MDA and component middleware as follows:

- Combining MDA 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 1.5.
- Combining MDA 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 MDA 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, MDA-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 MDA helps to model the interfaces among various components in terms of standard middleware interfaces, rather than language-specific features or proprietary APIs.

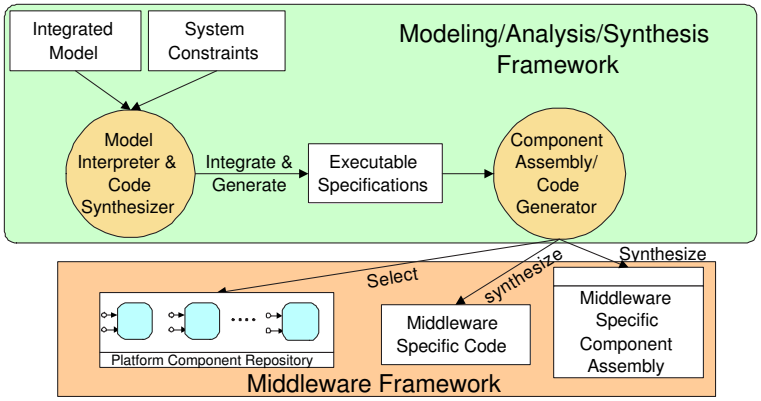

Figure 1.5: **Integrating MDA and Component Middleware to Create Model Driven Middleware**

- 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, such as Cadena (Hatcliff et al. 2003).

Figure 1.6 illustrates seven points at which MDA can be integrated with component middleware architectures and applied to DRE applications. Below, we present examples of each of these integration points in the Model Driven Middleware paradigm:
1. Configuring and deploying application services end-to-end. Developing complex DRE applications requires application developers to handle a variety of configuration, packaging and deployment challenges, such as

- Configuring appropriate libraries of middleware suites tailored to the QoS and footprint requirements of the DRE application
- Locating the appropriate existing services
- Partitioning and distributing application processes among component servers using the same middleware technologies and
- Provisioning the QoS required for each service that comprises an application end-to-end.

It is a daunting task to identify and deploy all these capabilities into an efficient, correct, and scalable end-to-end application configuration. For example, to maintain correctness and efficiency, services may change or migrate when the DRE application requirements change. Careful analysis is therefore required for large-scale DRE systems to effectively partition collaborating services on distributed nodes so the information can be processed efficiently, dependably, and securely. The OMG's Deployment and Configuration specification (Obj 2003a) addresses these concerns and describes the mechanisms by which distributed component-based applications are configured and deployed.

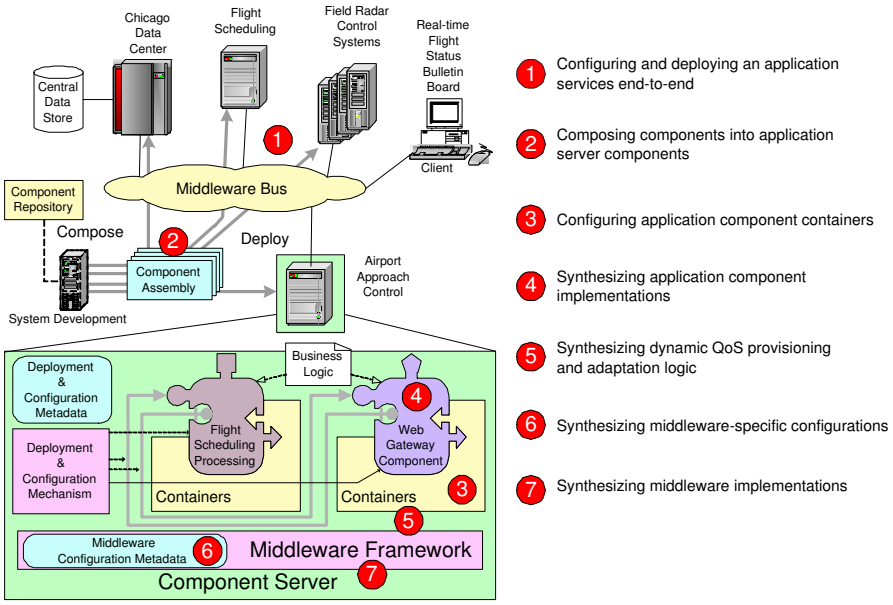

Figure 1.6: **Integrating Model Driven Architecture with Component Middleware**

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

2. Composing components into component servers. Integrating MDA 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 component server.

OMG MDA tools, such as *OptimalJ* from Compuware, provide tools for composing J2EE component servers from visual models.

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

Due to the number of policies and the intricate interactions among them, it is tedious and error-prone for a DRE application developer to *manually* specify and maintain component policies and semantic compatibility with policies of other components. MDA 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 MDA tools can then analyze the models and generate the necessary policies and ensure their consistency.

The Embedded Systems Modeling Language (ESML) (Karsai et al. 2002) developed as part of the DARPA MoBIES program uses MDA technology to model the behavior of, and interactions between, avionics components. Moreover, the ESML model generators synthesize fault management and thread policies in component containers.

4. Synthesizing application component implementations. Developing complex DRE applications 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 systems engineers and/or domain experts, rather than programmed manually 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 (Kiczales et al. 1997) and generator tools (Ledeczi et al. 2001) that can synthesize platform-specific code customized for specific application properties, such as resilience to equipment failure, prioritized scheduling, and bounded worst-case execution under overload conditions.

The Constraint Specification Aspect Weaver (C-SAW) (Gray et al. 2003) and Adaptive Quality Modeling Environment (AQME) (Neema et al. 2002) tools developed as part of the DARPA PCES program use MDA technology to provide a model driven approach for weaving in and synthesizing QoS adaptation logic DRE application components. In particular, AQME is used in conjunction with QuO/Qoskets (Technologies n.d.) to provide adaptive QoS policies for an unmanned aerial vehicle (UAV) real-time video distribution application (R. Schantz and J. Loyall and D. Schmidt and C. Rodrigues and Y. Krishnamurthy and I. Pyarali 2003).

5. Synthesizing dynamic QoS provisioning and adaptation logic. Based on the overall system model and constraints, MDA tools may decide to plug in existing

dynamic QoS provisioning and adaptation modules, using appropriate parameters. When none is readily available, the MDA tools can assist in creating new behaviors by synthesizing appropriate logic, *e.g.*, using QoS-enabled aspect languages (Pal et al. 2000). The generated dynamic QoS behavior can then be used in system simulation dynamically to verify its validity. It can then be composed into the system as described above.

The AQME (Neema et al. 2002) modeling language mentioned at the end of integration point 4 above models the QuO/Qosket middleware by modeling system conditions and service objects. For example, AQME enables modeling the interactions between the sender and receiver of the UAV video streaming applications, as well as parameters that instrument the middleware and application components.

6. Synthesizing middleware-specific configurations. The infrastructure middleware technologies used by component middleware provide a wide range of policies and options to configure and tune their behavior. For example, CORBA Object Request Brokers (ORBs) often provide many options and tuning parameters, such as various types of transports and protocols, various levels of fault tolerance, middleware initialization options, efficiency of (de)marshaling event parameters, efficiency of demultiplexing incoming method calls, threading models and thread priority settings, and buffer sizes, flow control, and buffer overflow handling. Certain combinations of the options provided by the middleware may be semantically incompatible when used to achieve multiple QoS properties.

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

Advanced meta-programming techniques, such as adaptive and reflective middleware (Cross and Schmidt 2002; Fábio M. Costa and Gordon S. Blair 1999; Gordon S. Blair and G. Coulson and P. Robin and M. Papathomas 1998; Kon et al. 2002) and aspect-oriented programming (Kiczales et al. 1997), are being developed to configure middleware options so they can be tailored for particular DRE application use cases.

7. Synthesizing middleware implementations. MDA tools can also be integrated with component middleware by using generators to synthesize custom middleware implementations. This integration is a more aggressive use of modeling and synthesis than integration point 6 described above since it affects middleware *implementations*, rather than just their configurations. For example, application integrators could use these capabilities to generate highly customized implementations of component middleware so that it (1) only includes the features actually needed for a particular application and (2) is carefully fine-tuned to the characteristics of particular programming languages, operating systems, and networks.

The customizable middleware architectural framework *Quarterware* (Campbell et al. 1998) is an example of this type of integration. Quarterware abstracts basic middleware functionality and allows application-specific specializations and extensions.

The framework can generate core facilities of CORBA, Remote Method Invocation (RMI), and Message Passing Interface (MPI). The framework-generated code is optimized for performance, which the authors demonstrate is comparable—and often better—than many commercially available middleware implementations.

1.4 Model Driven Middleware Case Study: Integrating MDA with QoS-enabled Middleware for Avionics Mission Computing

The Model Driven Middleware tool suite we are developing is called CoSMIC (Component Synthesis with Model Integrated Computing) (Gokhale 2003; Gokhale et al. 2002; Lu et al. 2003). Our research on CoSMIC is manifested in the integration of OMG MDA with QoS-enabled component middleware, such as CIAO (Wang et al. 2003b), along the seven points illustrated in Figure 1.6. This section illustrates how the Model Driven Middleware concept manifested in CoSMIC is being operationalized in practice for the avionics mission computing domain.

Within the CoSMIC framework, we are integrating the OMG MDA tools and processes with the CIAO QoS-enabled component middleware platform (Wang et al. 2003b) and applying them in the context of Boeing's Bold Stroke (Sharp 1998; Sharp and Roll 2003) avionics mission computing platform. Bold Stroke is a large-scale DRE platform that is based heavily on QoS-enabled component middleware. The CoSMIC tools we are developing for avionics mission computing are designed to model and analyze both application functionality and end-to-end application QoS requirements. With CIAO's support for QoS-enabled, reusable CCM components it is then possible for CoSMIC to:

- Model the QoS requirements of avionics applications using domain-specific modeling languages we have developed
- Associate the models with different static and dynamic QoS profiles
- Simulate and analyze dynamic behaviors
- Synthesize appropriate middleware configuration parameters including different policies of the CIAO containers
- Synthesize the QoS-enabled application functionality in component assemblies and
- Weave in crosscutting runtime adaptation logic metadata.

Figure 1.7 illustrates the interaction between CoSMIC and CIAO that enables the resolution of the challenges outlined in Section 1.1. Below we describe how we are using Model Driven Middleware in the context of avionics mission computing.

Handling avionics mission computing middleware configuration. The CIAO CCM implementation provides a large number of configuration parameters to fine tune its performance to meet the QoS needs of Bold Stroke avionics applications. CIAO imposes several constraints on what combinations of these options are valid for a given specification of application QoS requirements. To handle these complexities,

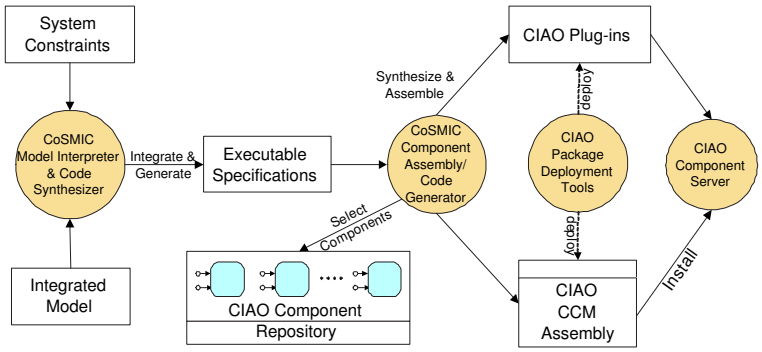

Figure 1.7 **Interactions between CoSMIC and CIAO**

CoSMIC provides a modeling paradigm (Karsai et al. 2003) called the *Options Configuration Modeling Language* (OCML) (Gokhale 2003). CIAO middleware developers can use OCML to model the available configuration parameters and the different constraints involving these options. Likewise, the Bold Stroke avionics mission computing application developers can use the OCML modeling, analysis, and synthesis framework to (1) model the desired QoS requirements and (2) synthesize the appropriate middleware configuration metadata that is then used by CIAO to fine tune its performance.

Figure 1.8 illustrates an example of using the OCML modeling paradigm to model a rule for combining configuration options in CIAO. DRE application developers using

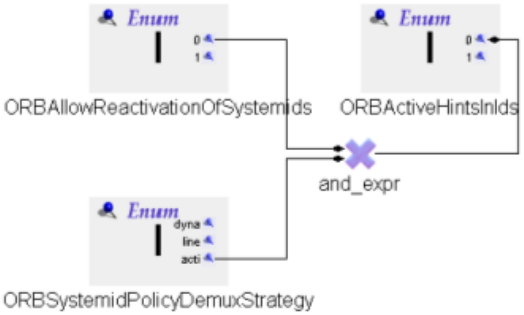

Figure 1.8 **Example using OCML Modeling Paradigm**

the OCML paradigm to choose the CIAO middleware configuration parameters will be constrained by the rules imposed by the suite of OCML models, such as those illustrated in Figure 1.8.

Handling avionics middleware component assembly and deployment. The CoSMIC tool suite provides modeling of DRE systems, their QoS requirements, and

component assembly and deployment. A modeling paradigm called the *Component Assembly and Deployment Modeling Language* (CADML) (Gokhale 2003) has been developed for this purpose. In the context of avionics mission computing, the CADML assembly and deployment models convey information on which Bold Stroke components are collocated on which processor boards of the mission computer. CADML models also convey information on the replication of components used to enhance reliability. CADML is based on a related paradigm called the Embedded Systems Modeling Language (ESML) (Karsai et al. 2002). Whereas ESML enables modeling a proprietary avionics component middleware, CoSMIC CADML enables modeling the standards-based CCM components, and their assembly and deployment described in the OMG D&C specification (Obj 2003a).

Figure 1.9 illustrates an example of using the CADML modeling paradigm to model an assembly of components of an avionics application. This figure illustrates a simple avionics application comprising an assembly of a GPS, an airframe, and a navigational display component. The CoSMIC tool also provides synthesis tools targeted

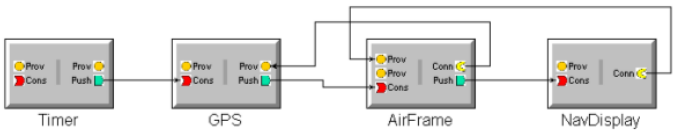

Figure 1.9 **Using CADML to Model Avionics Component Assembly**

at the CIAO QoS-enabled component assembly and deployment. As described in our companion chapter in this book (Wang et al. 2003b), QoS-enabled component middleware, such as 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 synthesis tools use the CADML models to generate the assembly and deployment data.

The CoSMIC component assembly and deployment tools described in this section have been applied successfully for modeling and synthesis of a number of Bold Stroke product scenarios. To address scalability issues for Bold Stroke component assemblies comprising large number of components, *e.g., 50 or more*, CADML allows partitioning the assemblies into manageable logical units that are then themselves assembled. Moreover, using CADML to synthesize the assembly metadata for such large assemblies is a big win over manually handcrafting the same both in terms of the scalability as well as correctness of the synthesized assembly metadata.

1.5 Related Work

This section reviews related work on model driven architectures and describes how modeling, analysis, and generative programming techniques are being used to model and provision QoS capabilities for DRE component middleware and applications.

Model-based software development. Research on Model Driven Middleware extends earlier work on Model-Integrated Computing (MIC) (Gray et al. 2001; Harel and Gery 1996; Lin 1999; Sztipanovits and Karsai 1997) that focused on modeling and synthesizing embedded software. MIC provides a unified software architecture and framework for creating Model-Integrated Program Synthesis (MIPS) environments (Ledeczi et al. 2001). Examples of MIC technology used today include GME (Ledeczi et al. 2001) and Ptolemy (Buck et al. 1994) (used primarily in the real-time and embedded domain) and MDA (Obj 2001a) based on UML (Obj 2001b) and XML (Domain n.d.) (which have been used primarily in the business domain). Our work on CoSMIC combines the GME tool and UML modeling language to model and synthesize QoS-enabled component middleware for use in provisioning DRE applications. In particular, CoSMIC is enhancing GME to produce domain-specific modeling languages and generative tools for DRE applications, as well as developing and validating new UML profiles (such as the UML profile for CORBA (Obj 2002c), the UML profile for quality of service (Obj 2003b), and UML profile for schedulability, performance and time (Obj 2003c)) to support DRE applications.

As part of an ongoing collaboration (R. Schantz and J. Loyall and D. Schmidt and C. Rodrigues and Y. Krishnamurthy and I. Pyarali 2003) between ISIS, University of Utah, and BBN Technologies, work is being done to apply GME techniques to model an effective resource management strategy for CPU resources on the TimeSys Linux real-time OS (TimeSys 2001). Timesys Linux allows applications to specify CPU reservations for an executing thread, which guarantee that the thread will have a certain amount of CPU time, regardless of the priorities of other threads in the system. Applying GME modeling to develop the QoS management strategy simplifies the simulation and validation necessary to assure end-to-end QoS requirements for CPU processing.

The Virginia Embedded System Toolkit (VEST) (Stankovic et al. 2003) is an embedded system composition tool that enables the composition of reliable and configurable systems from COTS component libraries. VEST compositions are driven by a modeling environment that uses the GME tool (Ledeczi et al. 2001). VEST also checks whether certain real-time, memory, power, and cost constraints of DRE applications are satisfied.

The Cadena (Hatcliff et al. 2003) project provides an MDA tool suite with the goal of assessing the effectiveness of applying static analysis, model-checking, and other light-weight formal methods to CCM-based DRE applications. The Cadena tools are implemented as plug-ins to IBM's Eclipse integrated development environment (IDE) (Object Technology International, Inc. 2003). This architecture provides an IDE for CCM-based DRE systems that ranges from editing of component definitions and connections information to editing and debugging of auto-generated code templates.

Commercial successes in model-based software development include the Rational Rose (Matthew Drahzal 1999) suite of tools used primarily in enterprise applications. Rose is a model driven development tool suite that is designed to increase the productivity and quality of software developers. Its modeling paradigm is based on the Unified Modeling Language (UML). Rose tools can be used in different application domains including business and enterprise/IT applications, software products and

systems, and embedded systems and devices. In the context of DRE applications, Rose has been applied successfully in the avionics mission computing domain (Sharp 1998).

Other commercial successes include the Matlab Simulink and Stateflow tools that are used primarily in engineering applications. Simulink is an interactive tool for modeling, simulating, and analyzing dynamic, multidomain systems. It provides a modeling paradigm that covers a wide range of domain areas, including control systems, digital signal processors (DSPs), and telecommunication systems. Simulink is capable of simulating the modeled system's behavior, evaluating its performance, and refining the design. Stateflow is an interactive design tool for modeling and simulating event-driven systems. Stateflow is integrated tightly with Simulink and Matlab to support designing embedded systems that contain supervisory logic. Simulink uses graphical modeling and animated simulation to bridge the traditional gap between system specification and design.

Program transformation technologies. Program transformation is used in many areas of software engineering, including compiler construction, software visualization, documentation generation, and automatic software renovation. The approach basically involves changing one program to another. Program transformation environments provide an integrated set of tools for specifying and performing semantic-preserving mappings from a source program to a new target program.

Program transformations are typically specified as rules that involve pattern matching on an abstract syntax tree (AST). The application of numerous transformation rules evolves an AST to the target representation. A transformation system is much broader in scope than a traditional generator for a domain-specific language. In fact, a generator can be thought of as an instance of a program transformation system with specific hard-coded transformations. There are advantages and disadvantages to implementing a generator from within a program transformation system. A major advantage is evident in the pre-existence of parsers for numerous languages (Baxter 2001). The internal machinery of the transformation system may also provide better optimizations on the target code than could be done with a stand-alone generator.

Generative Programming (GP) (Czarnecki and Eisenecker 2000) is a type of program transformation concerned with designing and implementing software modules that can be combined to generate specialized and highly optimized systems fulfilling specific application requirements. The goals are to (1) decrease the conceptual gap between program code and domain concepts (known as achieving high intentionality), (2) achieve high reusability and adaptability, (3) simplify managing many variants of a component, and (4) increase efficiency (both in space and execution time).

GenVoca (Batory et al. 1994) is a generative programming tool that permits hierarchical construction of software through the assembly of interchangeable/reusable components. The GenVoca model is based upon stacked layers of abstraction that can be composed. The components can viewed as a catalog of problem solutions that are represented as pluggable components, which then can be used to build applications in the catalog domain.

Yet another type of program transformation is aspect-oriented software development (AOSD). AOSD is a new technology designed to more explicitly separate concerns in software development. The AOSD techniques make it possible to mod-

ularize crosscutting aspects of complex DRE systems. An aspect is a piece of code or any higher level construct, such as implementation artifacts captured in a MDA PSM, that describes a recurring property of a program that crosscuts the software application *i.e.*, aspects capture crosscutting concerns). Examples of programming language support for AOSD constructs include AspectJ (Kiczales et al. 2001) and AspectC++ (Olaf Spinczyk and Andreas Gal and Wolfgang Schröder-Preikschat 2002).

1.6 Concluding Remarks

Large-scale distributed real-time and embedded (DRE) applications are increasingly being developed using QoS-enabled component middleware (Wang et al. 2003b). QoS-enabled component middleware provides policies and mechanisms for provisioning and enforcing large-scale DRE application QoS requirements. The middleware itself, however, does not resolve the challenges of choosing, configuring, and assembling the appropriate set of syntactically and semantically compatible QoS-enabled DRE middleware components tailored to the application's QoS requirements. Moreover, a particular middleware API does not resolve all the challenges posed by obsolescence of infrastructure technologies and its impact on long-term DRE system lifecycle costs.

The OMG's Model Driven Architecture (MDA) is emerging as an effective paradigm for addressing the challenges described above. The MDA is a software development paradigm that applies domain-specific modeling languages systematically to engineer computing systems. This chapter provides an overview of the emerging paradigm of *Model Driven Middleware*, which applies MDA techniques and tools to help configure and deploy QoS-enabled component middleware and DRE applications and large-scale systems of systems.

To illustrate recent progress on Model Driven Middleware, we describe a case study of applying the CoSMIC tool suite in the domain of avionics mission computing. CoSMIC is designed to simplify the integration of DRE applications that consist of QoS-enabled component middleware, such as the CIAO QoS enhancements to the CORBA Component Model (CCM) (Wang et al. 2003b). CoSMIC provides platform-dependent metamodels that describe middleware and container configurations, as well as platform-independent metamodels to describe DRE application QoS requirements. These metamodels can be used to provision static and dynamic resources in CIAO. By extending CIAO to support component deployment metadata for QoS policies, such as real-time priorities, DRE applications can be composed from existing components while applying various QoS policies. This capability not only reduces the cost of developing DRE applications, it also makes it easier to analyze the consistency of QoS policies applied throughout a system using MDA tools.

All the material presented in this book chapter is based on the CoSMIC Model Driven Middleware tools available for download at `www.dre.vanderbilt.edu/cosmic`. The associated component middleware CIAO can be downloaded in open-source format from `www.dre.vanderbilt.edu/Download.html`.

Bibliography

Aeronautics LM 2003 Lockheed Martin (MDA Success Story) `http://www.omg.org/mda/mda_files/LockheedMartin.pdf`.

Allen P 2002 Model Driven Architecture. *Component Development Strategies*.

Batory D, Singhal V, Thomas J, Dasari S, Geraci B and Sirkin M 1994 The GenVoca Model of Software-System Generators. *IEEE Software* **11**(5), 89–94.

Baxter I 2001 *DMA: A Tool for Automating Software Quality Enhancement*. Semantic Designs (www.semdesigns.com).

Buck JT, Ha S, Lee EA and Messerschmitt DG 1994 Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulation, Special Issue on Simulation Software Development Component Development Strategies*.

Campbell R, Singhai A and Sane A 1998 Quarterware for Middleware *Proceedings of ICDCS 98* IEEE.

Cross JK and Schmidt DC 2002 Applying the Quality Connector Pattern to Optimize Distributed Real-time and Embedded Middleware In *Patterns and Skeletons for Distributed and Parallel Computing* (ed. Rabhi F and Gorlatch S) Springer Verlag.

Czarnecki K and Eisenecker U 2000 *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston.

de Miguel MA 2002 QoS-Aware Component Frameworks *The* 10^{th} *International Workshop on Quality of Service (IWQoS 2002)*, Miami Beach, Florida.

Domain WA n.d. Extensible Markup Language (XML) `http://www.w3c.org/XML`.

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

Gokhale A 2003 Component Synthesis using Model Integrated Computing `www.dre.vanderbilt.edu/cosmic`.

Gokhale A, Natarajan B, Schmidt DC, Nechypurenko A, Gray J, Wang N, Neema S, Bapty T and Parsons J 2002 CoSMIC: An MDA Generative Tool for Distributed Real-time and Embdedded Component Middleware and Applications *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture* ACM, Seattle, WA.

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

Gray J, Bapty T and Neema S 2001 Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM* pp. 87–93.

Gray J, Sztipanovits J, Bapty T, Neema S, Gokhale A and Schmidt DC 2003 Two-level Aspect Weaving to Support Evolution of Model-Based Software In *Aspect-Oriented Software Development* (ed. Filman R, Elrad T, Aksit M and Clarke S) Addison-Wesley Reading, Massachusetts.

Harel D and Gery E 1996 Executable Object Modeling with Statecharts *Proceedings of the 18th International Conference on Software Engineering*, pp. 246–257. IEEE Computer Society Press.

Harrison TH, Levine DL and Schmidt DC 1997 The Design and Performance of a Real-time CORBA Event Service *Proceedings of OOPSLA '97*, pp. 184–199 ACM, Atlanta, GA.

Hatcliff J, Deng W, Dwyer M, Jung G and Prasad V 2003 Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems *Proceedings of the International Conference on Software Engineering*, Portland, OR.

Joseph K. Cross and Patrick Lardieri 2001 Proactive and Reactive Resource Reallocation in DoD DRE Systems *Proceedings of the OOPSLA 2001 Workshop "Towards Patterns and Pattern Languages for OO Distributed Real-time and Embedded Systems"*.

Karsai G, Neema S, Bakay A, Ledeczi A, Shi F and Gokhale A 2002 A Model-based Front-end to ACE/TAO: The Embedded System Modeling Language *Proceedings of the Second Annual TAO Workshop*, Arlington, VA.

Karsai G, Sztipanovits J, Ledeczi A and Bapty T 2003 Model-Integrated Development of Embedded Software. *Proceedings of the IEEE* **91**(1), 145–164.

Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J and Griswold WG 2001 An overview of AspectJ. *Lecture Notes in Computer Science* **2072**, 327–355.

Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes CV, Loingtier JM and Irwin J 1997 Aspect-Oriented Programming *Proceedings of the 11th European Conference on Object-Oriented Programming*.

Kon F, Costa F, Blair G and Campbell RH 2002 The Case for Reflective Middleware. *Communications of the ACM* **45**(6), 33–38.

Kopetz H 1997 *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, Massachusetts.

Ledeczi A, Bakay A, Maroti M, Volgysei P, Nordstrom G, Sprinkle J and Karsai G 2001 Composing Domain-Specific Design Environments. *IEEE Computer*.

Lin M 1999 Synthesis of Control Software in a Layered Architecture from Hybrid Automata *HSCC*, pp. 152–164.

Liu C and Layland J 1973 Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM* **20**(1), 46–61.

Locke CD 1992 Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives. *The Journal of Real-Time Systems* **4**, 37–53.

Lu T, Turkaye E, Gokhale A and Schmidt DC 2003 CoSMIC: An MDA Tool suite for Application Deployment and Configuration *Proceedings of the OOPSLA 2003 Workshop on Generative Techniques in the Context of Model Driven Architecture* ACM, Anaheim, CA.

Matthew Drahzal 1999 *Rose RealTime – A New Standard for RealTime Modeling: White Paper* Rose Architect Summer Issue 1999 edn Rational (IBM).

Neema S, Bapty T, Gray J and Gokhale A 2002 Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, Pittsburgh, PA.

Networks LG 2003 Optical Fiber Metropolitan Network `http://www.omg.org/mda/mda_files/LookingGlassN.pdf`.

Obj 2001a *Model Driven Architecture (MDA)* OMG Document ormsc/2001-07-01 edn.

Obj 2001b *Unified Modeling Language (UML) v1.4* OMG Document formal/2001-09-67 edn.

Obj 2002a *CORBA Components* OMG Document formal/2002-06-65 edn.

Obj 2002b *Real-time CORBA Specification* OMG Document formal/02-08-02 edn.

Obj 2002c *UML Profile for CORBA* OMG Document formal/02-04-01 edn.

Obj 2003a *Deployment and Configuration Adopted Submission* OMG Document ptc/03-07-08 edn.

Obj 2003b *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Joint Revised Submission* OMG Document realtime/03-05-02 edn.

Obj 2003c *UML Profile for Schedulability* Final Draft OMG Document ptc/03-03-02 edn.

Object Technology International, Inc. 2003 *Eclipse Platform Technical Overview: White Paper* Updated for 2.1, Original publication July 2001 edn Object Technology International, Inc.

Ogata K 1997 *Modern Control Engineering*. Prentice Hall, Englewood Cliffs, NJ.

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

Pal P, Loyall J, Schantz R, Zinky J, Shapiro R and Megquier J 2000 Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)* IEEE/IFIP, Newport Beach, CA.

R. Schantz and J. Loyall and D. Schmidt and C. Rodrigues and Y. Krishnamurthy and I. Pyarali 2003 Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware *Proceedings of Middleware 2003, 4th International Conference on Distributed Systems Platforms* IFIP/ACM/USENIX, Rio de Janeiro, Brazil.

Railways A 2003 Success Story OBB `http://www.omg.org/mda/mda_files/SuccessStory_OeBB.pdf/`.

Rajkumar R, Juvva K, Molano A and Oikawa S 1998 Resource Kernel: A Resource-Centric Approach to Real-Time Systems *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking* ACM.

Ritter T, Born M, Unterschütz T and Weis T 2003 A QoS Metamodel and its Realization in a CORBA Component Infrastructure *Proceedings of the* 36^{th} *Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003* HICSS, Honolulu, HW.

Schmidt DC, Schantz R, Masters M, Cross J, Sharp D and DiPalma L 2001 Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems. *CrossTalk*.

Sharp DC 1998 Reducing Avionics Software Cost Through Component Based Product Line Development *Proceedings of the 10th Annual Software Technology Conference*.

Sharp DC and Roll WC 2003 Modle-Based Integration of Reusable Component-Based Avionics System *Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003*.

Sprinkle JM, Karsai G, Ledeczi A and Nordstrom GG 2001 The New Metamodeling Generation *IEEE Engineering of Computer Based Systems* IEEE, Washington, DC.

Stankovic JA, Zhu R, Poornalingam R, Lu C, Yu Z, Humphrey M and Ellis B 2003 VEST: An Aspect-based Composition Tool for Real-time Systems *Proceedings of the IEEE Real-time Applications Symposium* IEEE, Washington, DC.

Sztipanovits J and Karsai G 1997 Model-Integrated Computing. *IEEE Computer* **30**(4), 110–112.

Technologies B n.d. Quality Objects (QuO) www.dist-systems.bbn.com/papers.

TimeSys 2001 TimeSys Linux/RT 3.0 `www.timesys.com`.

van Deursen A, Klint P and Visser J 2002 Domain-Specific Languages `http://homepages.cwi.nl/~jvisser/papers/dslbib/index.html`.

Wang N, Schmidt DC, Gokhale A, Gill CD, Natarajan B, Rodrigues C, Loyall JP and Schantz RE 2003a Total Quality of Service Provisioning in Middleware and Applications. *The Journal of Microprocessors and Microsystems* **27**(2), 45–54.

Wang N, Schmidt DC, Gokhale A, Rodrigues C, Natarajan B, Loyall JP, Schantz RE and Gill CD 2003b QoS-enabled Middleware In *Middleware for Communications* (ed. Mahmoud Q) Wiley and Sons New York.