

Applying Model-Integrated Computing to Middleware and Application Components

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

February 17, 2002

This paper has been submitted to a special issue of Communications of the ACM on Enterprise Components, Services, and Business Rules, edited by Ali Arsanjani.

Keywords: CORBA Component Model, Model-Driven Architectures, Architectural and Optimization Patterns

1 Introduction

Enterprise application is a term applied to a large class of applications that comprise enterprise resource planning (ERP), shop floor automation, supply chain management (SCM), e-business and customer relationship management (CRM) systems.

Examples of enterprise applications include airline reservation systems, bank asset management systems, and business-to-business supply chain management systems. These types of applications constitute the majority of information technology (IT) investment and staffing, with annual expenditures expected to exceed \$7.3bn worldwide by 2004.

Enterprise applications were historically developed using custom built, in-house applications built customizing proprietary systems like *HighExPlus*, *BancsConnect* and *EX* that are being used from mainframes to PC's. Due to deregulation, time-to-market pressures, and stiff global competition for human and economic resources, however, enterprise applications are increasingly being developed using services and protocols defined by commercial-off-the-shelf (COTS) middleware integration platforms, such as the Common Object Request Broker Architecture (CORBA) [1], Java 2 Enterprise Edition (J2EE) [2], and emerging web services middleware, such as .NET [3] based on XML [4] and SOAP [5]. COTS

middleware encapsulates specific services or sets of services to provide reusable building blocks that can be composed to develop enterprise applications rapidly and robustly. In particular, COTS middleware offers enterprise application developers the following reusable capabilities:

- *Horizontal infrastructure services*, such as object 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.

Despite advances in COTS middleware quality and ubiquity, however, developers of enterprise applications face the following important—yet unresolved—challenges:

- **Proliferation of middleware technologies.** Large-scale, long-lived enterprise applications require the underlying middleware technology platform be available on heterogeneous platforms and languages, interface with legacy code in different languages, and interoperate with multiple technologies from different suppliers. However, COTS middleware technologies, such as J2EE and the emerging .NET web services, do not yet provide a complete end-to-end solution to 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. However, conventional implementations of middleware web services cannot enforce complex QoS requirements of enterprise applications effectively since they were developed for less stringent requirements.

- **Addressing accidental complexities in assembling components.** To reduce application lifecycle costs and time-to-market, developers are required to assemble and deploy enterprise applications from COTS middleware components. However, selecting the right set of compatible components that address all the application requirements is a daunting task leading developers to write customized code. This task not only increases the application lifecycle costs, but also is extremely tedious and error-prone which increases the accidental complexities.

A promising way to address the three challenges described above is to apply *model-integrated computing* (MIC) technologies [?, 6]. MIC is a paradigm for developing application functionality and QoS requirements at higher levels of abstraction than is possible with programming languages like Java, C++, or C#. MIC requires developers to model an integrated, end-to-end view of the entire system including all the interdependencies. These models capture the essence of a class of applications rather than focusing on a single, custom application. MIC uses one set of tools that analyzes the interdependent features of the system captured in the model and determines the feasibility in terms of different QoS requirements. Another set of interpreter tools translate models into executable specifications which in turn can be used to synthesize the software.

In the context of enterprise applications, sophisticated MIC generator tools and aspect weavers can be applied to analyze the models and synthesize platform-specific code that is customized for specific middleware and application properties. Important properties for enterprise applications include isolation levels of a transaction, recovery strategies to handle various runtime failures, and authentication and authorization strategies. A portable, component-based middleware framework that supports configurable and adaptive QoS management is essential to the success of model-integrated computing.

This paper provides three contributions to the study of model-integrated computing for enterprise applications:

- We illustrate how the MIC paradigm can be applied to simplify the development of enterprise applications and reusable component services.
- We describe how pattern-oriented middleware enables modeling and synthesis tools to rapidly develop, assemble, and deploy middleware and component services tai-

lored for the needs of enterprise systems with multiple simultaneous QoS requirements.

- We discuss how emerging standards, such as the OMG Model Driven Architecture (MDA) [7] based on UML [8] and XML [4] and the CORBA Component Model (CCM) [9], can be used to enhance MIC technologies.

The remainder of this paper is organized as follows: Section 2 describes how model-integrated computing and middleware can be combined to resolve key challenges associated with enterprise application development; Section 3 illustrates how the MIC paradigm is being standardized via the OMG's MDA and CCM and also outlines our approach for synthesizing CCM-based application functionality from higher-level models; Section 4 describes how patterns help resolve the accidental complexities intrinsic to the task of synthesizing semantically-compatible components that address application functionality and QoS requirements; and Section 5 provides concluding remarks and future directions.

2

This section describes how model-integrated computing and middleware can be combined to resolve key challenges associated with enterprise application development.

2.1 Overview of Middleware

Middleware is systems software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware [10]. Its primary role is to functionally bridge the gap between application programs and the lower-level hardware and software infrastructure in order to coordinate how parts of applications are connected and how they interoperate. Middleware also simplifies the integration of components developed by multiple technology suppliers.

When implemented properly, middleware can help to:

- Shield software developers from low-level, tedious, and error-prone platform details, such as socket-level network programming.
- Amortize software lifecycle costs by leveraging previous development expertise and capturing implementations of key patterns in reusable frameworks, rather than rebuilding them manually for each use.
- Provide a consistent set of higher-level network-oriented abstractions that are much closer to application requirements in order to simplify the development of distributed and embedded systems.
- Provide a wide array of developer-oriented services, such as logging and security that have proven necessary to operate effectively in a networked environment.

Over the past two decades, various technologies, such as the Distributed Computing Environment (DCE) [11], have been devised to alleviate many complexities associated with developing software for enterprise applications. Their successes have added a new category of systems software to the familiar operating system, programming language, networking, and database offerings of the previous generation. Some of the most successful of these technologies have centered on distributed object computing (DOC) middleware, such as CORBA [1], J2EE [2], and the emerging web services middleware, such as .NET [3] based on XML [4] and SOAP [5]. DOC is an advanced, mature, and field-tested middleware paradigm that supports flexible and adaptive behavior. DOC middleware architectures are composed of relatively autonomous software objects that can be distributed or collocated throughout a wide range of networks and interconnects. Clients invoke operations on target objects to perform interactions and invoke functionality needed to achieve application goals. Through these interactions, a wide variety of middleware-based services are made available off-the-shelf to simplify application development. Aggregations of these simple, middleware-mediated interactions form the basis of large-scale enterprise application deployments.

2.2 Overview of Model-Integrated Computing

Computer based systems are large, complex, software-hardware systems, where much of the complexity is due to interactions between the software and its environment. Model-integrated Computing (MIC) is defined as the principled application of domain-specific modeling languages to the engineering of computer based systems. MIC provides rich, domain-specific modeling environments including model analysis and model-based program synthesis tools.

MIC offers the following benefits:

- it frees developers from dependencies on any particular middleware technology's APIs.
- even if existing middleware technologies and their associated APIs are obsoleted by newer ones, the models can be used for a much longer time.

Combining MIC with middleware is important because it doesn't require the modeling tools to generate all the code. Instead, large portions of applications are simply reused from existing middleware components. This reusable middleware handles many QoS-critical aspects, such as concurrency, distribution, transactions, security, dependability, etc.

2.3 Resolving Enterprise Application Challenges with Model-Integrated Computing and Middleware

As described in Section 1, developing and deploying enterprise applications and services using today's COTS middleware technologies requires developers to address the following challenges:

- Proliferation of middleware technologies
- Satisfying multiple quality of service requirements simultaneously
- Eliminating accidental complexities arising out of manual assembly of components

A promising way to address the challenges of developing enterprise applications is to develop intelligent tools that can assemble the right set of components based on models. These models serve as higher-level specifications of application functionality and its QoS requirements.

Below we outline how the model-integrated computing and middleware can be combined to address these challenges more effectively.

Challenge 1: Proliferation of Middleware Technologies

Problem. There are an increasing number of middleware technologies, such as CORBA, J2EE, and .NET web services based on XML/SOAP outlined above. Although each has its strengths and weaknesses, no technology provides a comprehensive *one-fits-all* solution. For example, middleware technologies, such as Sun's J2EE and the Microsoft's emerging .NET web services, may be challenged to provide a complete end-to-end solution to build enterprise applications due to their dependency on implementation language, such as Java, or platforms such as Windows, respectively.

Business organizations also often have a considerable investment in legacy applications that do not use today's middleware technologies. In particular, large-scale enterprise applications often consist of components based on multiple software technologies developed over long periods of time. Problems arise when these components must interoperate or be integrated together into a single executable since it is hard to assemble semantically compatible and interoperable components based on multiple middleware platforms.

With an increasing trend towards availability of newer middleware technologies, it is imperative for enterprises to achieve good return on investment over a period of time. It is also important to interoperate seamlessly with emerging technologies without affecting their business rules and customers.

Solution. We are working on techniques based on the Model Drive Architecture (MDA) [7], which is a MIC standard proposed by the Object Management Group (OMG), The MDA

provides a standard way to address the middleware proliferation challenges outlined above. The MDA provides standard modeling languages such as the Unified Modeling Language (UML) [8]. Languages like UML can be used to model application functionality and system interactions. They can also be used as meta-modeling languages where platform specific models can themselves be modeled at an even higher-level, platform independent model.

However, the MDA does not yet adequately address a broad spectrum of distributed enterprise application QoS issues, in particular the integration of dependability, security, scalability, and predictability.

Challenge 2: Satisfying Multiple Quality of Service Requirements Simultaneously

Problem. A growing number of enterprise applications demand varying degrees and forms of quality of service (QoS) support from their middleware. For example, enterprise applications such as ERP, shop floor automation, supply chain management, e-business and CRM systems, require efficient, predictable, scalable, security, and fault tolerance guarantees. Owing to the complex nature of enterprise QoS requirements, it is not feasible to develop a single-vendor, end-to-end solution that addresses all these challenges. Instead, highly configurable, flexible, and optimized commercial off-the-shelf (COTS) components from several different providers based on standard distributed object computing (DOC) middleware are required to rapidly assemble and deploy these systems.

Solution. We are using the MDA modeling tools to model at a higher level of abstraction not only enterprise application functionality but also its end-to-end QoS requirements. These models can then be analyzed by intelligent tools and the required functionality can be mapped into platform-specific deployment descriptors, such as XML configuration files, that contain all the QoS-related information. The descriptors are then used by the platform-specific *component servers* or *aspect weavers* [12] that can weave in the right set of components to deploy the application [6]. This process can be applied until the desired functionality is achieved.

Challenge 3: Lack of Adequate End-user Programming Skills

Problem. End-users seldom possess the skills to either program components or to assemble and deploy the applications. Customized tools with intuitive user interfaces are therefore required for end-users to mix and match components to obtain the desired application functionality.

The right implementation choices, and even the right granularity of components, required to bring enterprise applications to market is often not understood until several iterations of a

system undergo field trials. To maintain a competitive edge, however, it is imperative that the time-to-deployment of an application is minimized. Component vendors must therefore be able to develop and deploy the appropriate component configurations to their customers rapidly, while minimizing component lifecycle costs.

Solution. A portable component-based middleware framework that supports configurable and adaptive QoS management is essential to achieve this approach.

Currently, we are working on code-generation tools that synthesize code for OMG's CORBA Component Model (CCM). CORBA is a distributed object computing middleware standard defined by the OMG that allows clients to invoke operations on remote objects without concern for where the object resides or what language the object is written in [13]. In addition, CORBA shields applications from non-portable details related to the OS/hardware platform they run on and the communication protocols and networks used to interconnect distributed objects. These features make CORBA ideally suited to provide the core communication infrastructure for distributed applications. The CORBA component model (CCM) provides application programmers with a standard way to implement, manage, configure, and deploy components that implement and integrate CORBA services. The CCM standard not only enables greater software reuse for servers, it also provides greater flexibility for dynamic configuration of CORBA applications.

Challenge 4: Assembling semantically compatible QoS-enabled components

Problem. QoS-enabled DOC component middleware, such as the CCM, comprises a library of reusable, QoS-enabled components that can be composed to assemble and deploy applications. However, a naive approach to assembling these components leads to incompatible, non-interoperable components getting assembled thereby increasing the accidental complexities. Moreover, manual assembly of compatible components is tedious and error-prone which adversely affects application lifecycle costs and time-to-market.

Solution. We are addressing this challenge by developing MDA translator tools that can analyze application functionality and QoS requirements based on the input models. The tools can then synthesize pattern-oriented, semantically compatible standard middleware from the library of QoS-enabled, reusable CCM components. The decision on which patterns make most sense will be made by the tools based on the input models.

3 Using Model Driven Architecture to Compose QoS-enabled Component-based Application Functionality

This section describes how the OMG Model Driven Architecture (MDA) can synthesize the required enterprise application functionality specified as models by assembling semantically compatible QoS-enabled, reusable components provided by the CORBA Component Model (CCM). First, we briefly describe the MDA and the CCM. Next, we outline our approach to synthesizing CCM-based application functionality from models.

3.1 Overview of MDA and CCM

This section provides a brief overview of the Model Driven Architecture (MDA) and the CORBA Component Model (CCM).

3.2 Model Driven Architecture

The MDA proposed by the Object Management Group (OMG) provides a standard way to address the middleware proliferation challenges explained in Section 1. The MDA builds upon years of research on model-integrated computing [?, 6] to provide platform-independent models (PIMs) and platform-specific models (PSMs) that streamline platform integration issues and protect investments against the uncertainty of changing platform technology. PIM and PSM descriptions of applications are formal specifications built using modeling standards, such as the Unified Modeling Language (UML) [8]. The PIM models are mapped into PSMs via translators. PSMs can be implemented using standard or proprietary middleware platforms, such as CORBA or .NET.

Although the OMG MDA standard has adopted the UML-based PIM and PSM for CORBA, they do not yet adequately address a broad spectrum of distributed application QoS issues, in particular the integration of dependability, scalability, security, and predictability that is very crucial for the success of enterprise applications.

3.2.1 CORBA Component Model

One of the common goals shared by both MDA and CCM is to decouple application-specific functionality and logic from the accidental complexities inherent in the infrastructure. This enables application programmers to concentrate on the application-specific functionality alone.

Conventional ways of developing application software, however, often yield code that is not robust enough because objects are often tightly coupled with each others. Moreover,

many non-functional aspects, such as persistent data store, security, and housekeeping of run time environment, that weave through multiple layers result in even more tight coupling between application objects and with the infrastructure and its associated housekeeping tasks.

These tight couplings lead to brittle abstractions that are hard to reuse.

Software frameworks try to address this issue by abstracting out common application logic into frameworks. However, this solution is not complete since application objects are still allowed to interact directly with each other, which ultimately leads to tight coupling among application objects. Therefore, some housekeeping code is still required within the applications to manage the framework. Moreover, these software frameworks incur yet another dependency on the applications by tightly coupling them to the framework they are developed upon. Therefore, it is non-trivial to reuse application objects and port them to a different framework.

The Object Management Architecture (OMA) defined in the CORBA 2.x specifications [1] provides a uniform framework for building portable distributed systems by providing a flexible infrastructure framework. The CORBA 2.x specification, however, does not sufficiently address the aforementioned issues and system developers are required to address them in an *ad hoc* way. For example, object interconnections in a system composed of interdependent objects still need to be implemented by the developer. Moreover, a developer still needs to perform housekeeping work, such as initializing the broker and setting up policies, to get the system operational.

Like other component-based technologies, *e.g.* EJB and DCOM, the CORBA Component Model (CCM) addresses these concerns by creating a physical boundary around components with well defined interfaces, and executing components in generic “application servers.”

A component is the unit of reuse in a CCM system. System designers *compose* systems by specifying the component interconnections using meta-data. As the actual component interconnections are not established until components are deployed in a component server, there is no tight coupling among components. Below, we list the major features offered by the CCM and benefits.

Figure 1 shows an overview of the run-time architecture of the CCM model.

3.3 Synthesizing Component-based Application Functionality

The CCM provides the mechanisms for composing a system from reusing existing components. The MDA, on the other hand, can take advantage of the deployment facility that CCM provides to achieve this. One virtue of using MDA is to uti-

Section 4 describes how patterns can help MDA tools to generate semantically correct CCM deployment descriptors thereby eliminating the accidental complexities in manually hand-crafting these.

4 Applying Patterns for Synthesis of Middleware Components

As noted in Section 3.3, manual hand-crafting of semantically compatible, QoS-enabled CCM components is tedious and error-prone. The solution we proposed was to use the model driven approach to synthesizing application code.

However, the task of selecting and assembling semantically compatible and provably correct reusable components is also a difficult task for MDA tools. Patterns help address this challenge.

In the following section we discuss the common patterns that are useful to implement such MDA tools. Breaking up the component framework into logical formulations of patterns helps the composition problem. Fixing responsibilities for every participant in the pattern and its interaction behavior with other patterns helps to view systems as smaller sub-systems of patterns.

4.1 Configuration Patterns

Context: Meta-level architectures such as the one we described that enable developers to compose application should allow configuration of different properties and services. The types of properties that need to be configured includes the QoS properties, type of communication model, such as connection oriented or push-pull, type of event handling, and type of demultiplexing mechanisms. The application could also configure services that would get triggered when certain types of events occur in the system.

Problem: Providing all the combinations of properties at link time to an application is obtrusive for the following reasons (1) flexibility of composing applications is limited i.e. adding or removing a property that needs to be supported involves compiling all of the framework code and the application code, and (2) application would unnecessarily incur costs of footprint and memory for all the combination of properties that they do not use.

Solution: The solution is to use the *Configuration patterns* [14]. The *Component Configurator* [15] allows the application to link and unlink properties provided using the *Strategy* [16] pattern dynamically at run-time instead of statically linking in the properties with applications. This can be used to

Figure 1: Overview of the CCM Run-time Architecture

lize complex modeling tools that can check for certain properties of the implementation, e.g., check the correctness of an algorithm, or ensure that a series of constraints are enforced, etc. Moreover, with support for QoS-enabled, reusable CCM components, it is possible to model the QoS requirements of applications using MDA-provided tools such as UML and synthesize the QoS-enabled application functionality.

The CCM also defines several helpful patterns and abstractions that can be reused, rather than regenerated by an MDA tool everytime. For example, instead of generating entire CCM components, the MDA tools can extend the XML deployment descriptors that define the QoS requirements that a component or a deployment needs. These descriptors serve as input to application server (component server) that in turn decipher the information and determine how to provide the necessary QoS support.

Below we outline an iterative process that can be used to assemble and deploy QoS-enabled enterprise applications in a timely manner, while also minimizing lifecycle costs.

1. Use the MDA to model a system by mixing and matching existing off-the-shelf components, and partitioning or defining the functionality of new components. This will also include defining what QoS properties of the components can (or need to) be adjusted. The result of this step will clearly define the contract a component supports and will enable us to take advantage of MDA to verify the system correctness.
2. Based on the component definition from the previous step, the component implementation can be refined using MDA, with the MDA ensuring the correctness of refined component implementation.
3. Deploy the resulting system for testing and tuning. Here, we can again use MDA to tweak the QoS requirements of the components and change the system configurations by fine tuning the deployment descriptors.

separate the different properties as different components and load them when the application needs it. *Interceptor* pattern allows adding and triggering services when certain events occur in the system.

4.2 Event Handling Patterns

Context: Distributed systems based on the “request-response” architectures usually have a client system that opens up a connection to the server offering a service and makes a request to it. The server in turn accepts a connection, processes the request, and sends back a response on the same connection.

In the meta-level architectures that we describe, the applications might require communicating over different transports using different messaging protocols like GIOP, SOAP, HTTP-NG or RMI.

Problems: The problem is multi-dimensional, in the sense that it spans from choosing the right transport and right messaging protocol that needs to be used to initiate a request, to the message specific processing that needs to be performed to send back a response on the same transport.

Solution: The solution is to use the *Acceptor-Connector* pattern that addresses the division of concerns [17] of connection initialization and processing that needs to be performed on the messages received over the connection. Previous work on ORB middleware [18] has shown a possible architecture using the Component Configurator pattern with the Acceptor-Connector pattern to address the problem of communicating over different transports and protocols.

In addition to the patterns in the contexts explained above, patterns like *Microkernel* [19] and *Reflection* are useful for developing adaptable systems.

5 Concluding Remarks

There has been a proliferation of middleware technologies that address various requirements of enterprise applications. These types of applications must often be 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 QoS properties simultaneously.

This paper describes a model-driven approach to addressing these challenges. It illustrates the benefits of model integrated computing for developing large-scale enterprise applications focusing on the MDA standard. It also describes how pattern-oriented middleware enables modeling and synthesis

tools to rapidly develop, assemble, and deploy DOC middleware based, large-scale, enterprise systems with multiple, simultaneous QoS requirements.

We are developing a framework to provide the modeling and synthesis tools to develop, assemble, and deploy middleware large-scale, enterprise systems with multiple, simultaneous QoS requirements.

References

- [1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.6 edition, Dec. 2001.
- [2] Sun Microsystems, “Java™ 2 Platform Enterprise Edition,” <http://java.sun.com/j2ee/index.html>, 2001.
- [3] Microsoft, “.NET Web Services Platform,” <http://www.microsoft.com/net>.
- [4] W3C Architecture Domain, “Extensible Markup Language (XML),” <http://www.w3c.org/XML>.
- [5] W3C, “Simple Object Access Protocol (SOAP) 1.1,” <http://www.w3c.org/TR/SOAP>, May 2000.
- [6] Akos Ledeczki, Arpad Bakay, Miklos Maroti, Peter Volgysei, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai, “Composing domain-specific design environments,” *IEEE Computer*, pp. 44–51, November 2001.
- [7] Object Management Group, *Model Driven Architecture (MDA)*, OMG Document ormsc/2001-07-01 edition, July 2001.
- [8] Object Management Group, *Unified Modeling Language (UML) v1.4*, OMG Document formal/2001-09-67 edition, September 2001.
- [9] BEA Systems, et al., *CORBA Component Model Joint Revised Submission*, Object Management Group, OMG Document orbos/99-07-01 edition, July 1999.
- [10] Richard E. Schantz and Douglas C. Schmidt, “Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications,” in *Encyclopedia of Software Engineering*, John Marciniak and George Telecki, Eds. Wiley & Sons, New York, 2001.
- [11] Ward Rosenberry, David Kenney, and Gerry Fischer, *Understanding DCE*, O’Reilly and Associates, Inc., 1992.
- [12] The AspectJ Organization, “Aspect-Oriented Programming for Java,” www.aspectj.org, 2001.
- [13] Steve Vinoski, “CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments,” *IEEE Communications Magazine*, vol. 14, no. 2, February 1997.
- [14] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*, Wiley & Sons, New York, 2000.
- [15] Prashant Jain and Douglas C. Schmidt, “Service Configurator: A Pattern for Dynamic Configuration and Reconfiguration of Communication Services,” in *The 3rd Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07)*, February 1997.
- [16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts, 1995.
- [17] Douglas C. Schmidt, “Acceptor and Connector: Design Patterns for Actively and Passively Initializing Network Services,” in *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP ’95*, Aarhus, Denmark, August 1995.
- [18] Fred Kuhns, Carlos O’Ryan, Douglas C. Schmidt, and Jeff Parsons, “The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware,” in *Proceedings of the IFIP 6th International Workshop on Protocols For High-Speed Networks (P/HSN ’99)*, Salem, MA, August 1999, IFIP.
- [19] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture – A System of Patterns*, Wiley and Sons, New York, 1996.