DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
REPORT A-2000-3

Performance Modeling Framework for CORBA Based Distributed Systems

Pekka Kähkipuro

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium XII, University Main Building, on August 25th, 2000, at 12 o'clock noon.

UNIVERSITY OF HELSINKI
FINLAND

Contact information

Postal address:
Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 1911

Telefax: +358 9 191 44441

ISSN 1238-8645
ISBN 951-45-9496-7
Computing Reviews (1998) Classification: C.2.4, C.4, D.2.2
Helsinki 2000
Helsinki University Printing House

Performance Modeling Framework for CORBA Based Distributed Systems

Pekka Kähkipuro

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
Pekka.Kahkipuro@cs.Helsinki.FI

PhD Thesis, Series of Publications A, Report A-2000-3
Helsinki, May 2000, 151 + 16 pages
ISSN 1238-8645, ISBN 951-45-9496-7

Abstract

The CORBA platform is increasingly popular in distributed computing due to its ability to hide complex implementation issues from application developers. However, performance consequences of the underlying techniques often remain visible to software developers. Moreover, new performance concerns may emerge because of additional layering and indirection. *Performance modeling* allows developers to understand and predict the performance of CORBA based systems.

In this work, we propose a performance modeling framework for supporting the development of CORBA based distributed systems. The main elements of the framework are (1) a notation for expressing the performance models, (2) a set of modeling techniques for representing common solutions in CORBA based systems, (3) an algorithm for solving the resulting models for a number of relevant metrics, (4) a tool for automating some of the tasks required by the framework, and (5) a methodology for creating and modifying the models.

Our modeling notation is based on the Unified Modeling Language (UML). To include performance related information into UML diagrams, we define several extensions that conform to the standard UML extension mechanisms. Special techniques are proposed for supporting layered modeling so that different aspects of the system can be described with separate and interrelated UML diagrams.

To solve UML based performance models, we transform the models into queuing networks with simultaneous resource possessions. The networks are further decomposed into a set of auxiliary product-form queuing networks, and iteration is used for finding an approximate solution that respects the mutual dependencies between the auxiliary networks. This algorithm combines existing approximation techniques in a novel way so that commonly occurring situations in object-oriented systems, such as callbacks and recursive calls, can be dealt with.

Our concise methodology defines a layered structure for modeling CORBA based distributed systems. In addition, it describes activities for creating and refining the models. We also present a prototype tool and discuss a case study where the results of a performance model are compared to metrics obtained from an actual system. The case study confirms that a relatively complex CORBA based system can be modeled with the framework without significant difficulties.

Computing Reviews (1998) Categories and Subject Descriptors:

C.2.4 Computer-communication networks: Distributed Systems
C.4 Performance of systems – Modeling techniques
D.2.2 Software engineering: Design Tools and Techniques

General terms:

Design, Performance

Additional Key Words and Phrases:

CORBA, mean value analysis, middleware, queuing network models, software performance engineering, Unified Modeling Language

Acknowledgements

I wish to thank my advisor Prof. Martti Tienari for his guidance and encouragement during the years that I have been working at the University of Helsinki. It would have been practically impossible to realize this work without his support. Prof. Kimmo Raatikainen has also given me excellent professional guidance. He has given me valuable advice on both the subject matter of my work and also on the the process of producing the results. I would also like to thank Prof. Murray Woodside for his valuable and highly professional comments that significantly improved the quality of my work. It makes no sense to leave anything half-finished. Finally, a special thanks is due to Prof. Timo Alanko. More than once, he has guided me to the right track in my choice of tools and techniques for this work. However, there is no match for the support that I have received from my wife Marjatta. She has patiently watched me implementing my endless performance tests, and she has tolerated and supported me during the long months that I have spent writing this work.

Helsinki, August 2000

Pekka Kähkipuro

Contents

Chapter 1

Introduction

Distributed computing is increasingly popular in the development of commercial information systems. This trend is unlikely to change due to the increasing demand for distributed solutions and also because of the continuous progress in PC-based hardware solutions and communications technology. However, distributed systems also entail difficulties that cannot be ignored by system developers and end users. Some of them arise from the complexity implied by the distribution of system components. Additional difficulties are caused by the heterogeneity that is hard to avoid in any reasonably sized system.

The CORBA platform from the Object Management Group [OMG99d] has been created to overcome difficulties encountered in heterogeneous distributed systems. *Distribution transparencies* have a key role in the CORBA platform, since they hide many problematic aspects of distribution and heterogeneity from software developers and end users [ISO95, Käh98b]. For example, the location transparency hides the location of system components, and the access transparency masks out differences in accessing various elements in the system. In addition, the CORBA platform defines tools and techniques for making software development easier. For example, the IDL language allows the specification of interfaces in a way that is independent of any programming language. Similar concepts are available in other object-oriented middleware environments, such as the Java platform from Sun Microsystems [Sun99] and the COM platform from Microsoft [Mic98].

While CORBA and other object-oriented middleware platforms may considerably facilitate the construction of distributed systems, a number of drawbacks have also been observed. In particular, the performance of middleware based systems may be significantly inferior to the performance of similar systems that have been implemented with lower-level tools. In many cases, the middleware itself results in a significant overhead by introducing additional layering and indirection. In addition, the use of middleware may lead to software architectures with unsatisfactory performance. In particular, the ease of implementing remote communication with the CORBA platform may mislead software engineers to use remote invocations excessively. As a result, the responsiveness of the software may drop significantly.

To improve the performance of middleware based distributed systems, benchmarking [OMG99a, Gra91] and performance heuristics [Käh98a] can provide useful hints for the development work. They help developers to find promising technologies and architectures but, unfortunately, they do not give any quantitative estimates for the performance of the resulting system. In addition, such techniques provide assistance only during a few steps in the software life cycle. A more comprehensive approach, *software performance engineering* (SPE), has been proposed to overcome these limitations. It refers to a complete methodology and a set of supporting tools for constructing software systems to meet performance objectives [Smi90].

The role of *performance modeling* is essential in SPE, as it allows designs to be validated against performance requirements at all stages of software development. The idea is to build a series of predictive performance models of the system using the available information, and to solve it for the relevant metrics, such as the average response time, throughput, and utilization. However, traditional performance modeling has concentrated on centralized systems and communication issues in client/server systems [Men94, Smi90, Hav98]. Less attention has been paid to a number of issues that are important for CORBA based distributed systems. Such issues include the modeling of software server contention together with hardware contention, the decomposition of large and complex systems into manageable parts or layers, the support for designs at different levels of abstraction, and the support for refining performance models as the software design develops. An essential feature in CORBA based systems is the existence of a middleware layer that hides a number implementation issues from application developers but strongly affects the performance of applications. Traditional performance modeling techniques provide only weak support for middleware based designs. Moreover, there is little support for programming techniques commonly used in CORBA based systems, such as callback mechanisms and recursive calls.

Most traditional performance modeling approaches use specialized modeling notations and assume an in-depth understanding of the underlying modeling techniques. This makes them unattractive for general-purpose software engineering. It is both costly and error prone to convert the results of software design into, say, stochastic Petri nets that depict exactly the same system with the same behavior. This *cognitive gap* between software engineering concepts and performance modeling approaches has been pointed out by several authors [Utt97, Wat97, Sho98].

1.1 Scope of the work

In this work, we describe a *performance modeling framework* for supporting the development and maintenance of CORBA based distributed systems. The framework produces predictive performance models that can be used for obtaining performance related information on the target system at all stages of its development. For this purpose, the framework contains (1) a suitable notation for expressing the performance models, (2) a set of modeling techniques with representative examples, (3) an algorithm for solving the models, (4) a tool for automating some of the tasks required by the framework, and (5) a methodology for creating and refining the models during the system life cycle.

The scope of our framework is further limited by the following underlying technologies:

- The Unified Modeling Language (with minor extensions) is used for representing the performance models,
- Stochastic queuing networks are used as an underlying technical representation for the performance models,
- Mean value analysis based approximations are used for solving the queuing networks,
- Object-oriented analysis and design methodologies are used as a reference environment for our performance modeling methodology,
- Existing SPE methodologies are used for covering those performance related tasks that are not part of performance modeling.

The primary domain for the Unified Modeling Language (UML) [Rat97] is functional modeling of software systems, but the language has enough expressive power for performance modeling as well [Poo99, Dou99, OMG99i]. To represent performance related features that are not covered by the core UML, we use the standard UML extension mechanisms. The main advantage of using UML in our framework is the possibility to create both functional and performance models in parallel with the same design tools and with the same diagrams, and thereby reduce the cognitive gap between these domains. Since there is currently no human-readable textual representation for the UML, we use our own notation, Performance Modeling Language (PML), for representing those UML features that are relevant for performance modeling.

The UML allows models to be written in different ways, and not all of them are suitable for performance modeling. Therefore, we define a set of UML modeling techniques that support the construction of performance models for CORBA based distributed systems. To cope with the complexity of distributed systems, the proposed techniques divide the overall model into a set of understandable UML diagrams. Most of these techniques have been published earlier in [Käh99b].

To provide practical support for software engineering, the framework needs an algorithm for solving the performance models for the relevant performance metrics. We follow a stepwise approach. First, a UML based performance model is transformed into a stochastic queuing network with simultaneous resource possessions. Second, the resulting queuing network is decomposed into auxiliary queuing networks without simultaneous resource possessions. This way, each of the networks can be solved efficiently for the relevant metrics. However, a number of dependencies exist between the auxiliary networks and, hence, they cannot be solved independently. As a last step, we use iteration for solving the set of interdependent queuing networks. An early version of this algorithm was presented in [Käh99a] but it has been extended significantly to support the needs of the framework.

In our implementation of the above algorithm, we use an approximate version of mean value analysis (MVA) for solving the auxiliary queuing networks. A number of alternative techniques could have been used instead. They include other approximate algorithms [Agr85], simulation [Raa89a], and exact algorithms [Con89, Hav89]. However, the use of approximate MVA is well motivated in our work. On one hand, approximate MVA is computationally inexpensive compared to the alternatives. Hence, we can solve large models in a fairly short time, thus making the framework suitable for complex and sizeable systems. On the other hand, the selected MVA approximation has been used extensively and is known to be quite robust. Therefore, it is relatively safe to embed the algorithm into a tool that hides it from software developers. The use of MVA based techniques has also some drawbacks. There is only a limited choice of available scheduling disciplines and service time distributions for system resources. Also, distributions of the obtained metrics are not available. Hence, we rule out in our current implementation of the framework a substantial class of modeling tasks where distributions are considered significant [Raa89a]. Moreover, MVA approximations do not always produce accurate results (see e.g. [Hei84, Agr85]). However, the known limitations of approximate MVA are acceptable in our work, since our focus is on performance engineering issues.

Finally, we specify a concise performance modeling methodology in order to link our framework into the software engineering process. The methodology is intended to be used in parallel with existing UML based analysis and design methodologies (e.g. [Sou98, Jac98]). In addition, we assume that developers are using a suitable SPE approach (e.g. [Smi90, Jai91, Men94]) for those activities that are not directly related to performance models. Our methodology primarily indicates how to structure the performance model and how to obtain enough input data for the model during the software engineering process. It does not, for example, specify how the resulting metrics should be used for guiding the development process. Hence, it can be seen as an extension for UML based analysis and design methodologies and existing SPE approaches.

1.2 Goals

The main goal for the performance modeling framework is to offer a sufficient set of modeling techniques for supporting software performance engineering of CORBA based distributed systems. To reach this goal, four generic requirements must be met.

First, the framework should produce performance models that can be solved automatically in a reasonable amount of time for the relevant performance metrics, such as throughputs, response times, utilizations, and queue lengths. It is permissible to use approximate algorithms for producing the solutions, since the focus is on performance engineering. Very little structural limitations should be imposed for the application design. In particular, the framework should support all those design techniques that are commonly used for CORBA based systems. This way, the framework can be easily integrated with various development tools and methodologies.

Second, we require the framework to support the usual style of UML modeling as proposed in the UML standard [Rat97] and in the literature (e.g. [Eri98, Jac98, Dou99, Rum99]). This way, it is possible to extend existing functional models into performance models without rewriting them.

Third, we require the framework to clearly distinguish between different architectural aspects of CORBA based distributed systems. The framework should allow designers to keep application objects, system infrastructure, hardware resources, and network topology in separate UML diagrams. This way, it is possible to experiment with different design alternatives in some parts of the system without modifying other parts. This requirement is elaborated in Section 2.5 once the CORBA platform has been presented in more detail.

Fourth, the framework should support incremental development style, since this is commonly used with object-oriented analysis and design [Eri98, Jac98, Sou98]. In particular, it should be possible to build abstract and solvable performance models already in the analysis phase when the infrastructure and hardware issues are still unknown. Also, it should be possible to upgrade the first tentative models into more accurate ones without the need to write a completely new set of UML diagrams.

The main contributions of our work can be found from three areas. The first contribution is the proposed collection of UML based modeling techniques for describing CORBA based distributed systems. In earlier performance-related approaches, UML has mainly been used for describing the performance requirements, and the actual performance models have been described with traditional notations. In our approach, UML provides the primary representation for the performance models, and the proposed modeling techniques ensure that the resulting models can be solved for the relevant performance metrics. This approach has two advantages. First, the use of functional modeling notation reduces the cognitive gap between software designers and performance analysts. As a result, the same tools and modeling techniques can be used for functional modeling and performance modeling. Second, the UML has been specifically designed for representing large and complex systems in a flexible way, and this characteristic has been preserved in our framework.

The second contribution is the iterative algorithm for solving queuing networks with simultaneous resource possessions. The proposed algorithm combines existing iterative and approximation techniques in a novel way so that the special requirements of our framework can be satisfied. In particular, the algorihm allows synchronous and asynchronous messages to be sent between arbitrary elements in the system without requiring a layered calling structure for the elements in the model. This feature is useful for modeling systems where callbacks lead to cyclic dependencies between system elements.

The third contribution is the presentation of an overall framework for analyzing and predicting the performance of CORBA based distributed systems. Previous work on the performance of CORBA based systems has concentrated on measuring the response times of round trip calls under varying conditions. In addition, a number of qualitative heuristics have been proposed for improving application performance. Our work, on the other hand, provides a complete methodology and a set of tools for predicting quantitatively the performance of CORBA based systems. The proposed performance modeling framework strongly relies on previous results on performance modeling, but we add the elements that are necessary for modeling CORBA based systems. In particular, we propose modeling techniques for keeping apart the CORBA infrastructure and the application logic so that changes in either domain can be implemented without affecting the other domain.

1.3 Related research

We briefly discuss related research in three areas. First, we present performance modeling techniques that are suitable for representing complex software systems, such as those based on the CORBA platform. Second, we discuss existing work on the performance of CORBA based systems. Finally, we point out some performance modeling approaches that bear similarities to our framework.

Performance models for complex software systems

The support for simultaneous resource possessions is an essential feature for performance modeling techniques that can be applied to complex software systems. Synchronous calls to software servers are an important source for simultaneous resource possessions in CORBA based systems since the caller stays blocked until it obtains a reply from the server. Simultaneous resource possessions can also arise from the co-existence of software and hardware resources in the same model.

Early work in this area includes the *method of surrogates* that uses two product-form queuing networks for modeling different parts of the queuing delay caused by simultaneous resource possessions [Jac82]. Both networks contain a surrogate delay server that represents the component of the queuing delay captured by the other network. An approximate solution is found with an iterative algorithm that solves the models alternatively and propagates the intermediate results between them.

Stochastic Rendezvous Networks (SRNs) have been proposed for modeling systems where software or hardware objects interact with the rendezvous mechanism [Woo95]. The rendezvous mechanism can be used to model different interaction types present in software systems, including those using simultaneous resource possessions. For example, a synchronous CORBA operation request can be modeled as a special case of a rendezvous. The solution for a SRN can be found with an iteration technique that computes a series of intermediate solutions using an MVA approximation and continues until the estimated throughputs converge.

The *method of layers* has been proposed for solving complex systems with one or more layers of software servers [Rol92, Rol95]. Each layer is allowed to call services from the layer immediately below it, thus introducing simultaneous resource possession. To find a solution for the layered queuing network (LQN) model, an iterative algorithm is used for solving the layers until successive response time estimates converge. A second queuing network is used for determining queuing delays in hardware devices, and the results are combined with the software server model to produce performance estimates for the overall system.

A similar layered queuing network model is proposed in [Ram98] for representing client-server systems where communication is carried out with synchronous and asynchronous messages. Unlike the method of layers, this approach requires that the complete flow of messages through the clients and servers be specified. Hence, the model is closer to the actual software and may be easier to construct. An approximate solution is obtained by iterating back and forth between the layers until the results converge.

Performance of CORBA based systems

Existing work on the performance of CORBA concentrates on the implementations of the development platform and the run-time infrastructure. The results of a comparative study between CORBA and low-level mechanisms in an ATM network have been reported in [Gok96]. For simple scalar types in a remote operation, the measurements reveal that CORBA implementations achieve 75% to 80% of the throughput of socket-based C and C++ implementations. For complex data structures, the throughput of CORBA implementations is only about 33% of the lower-level implementations. Further results show that a significant amount of time is spent in presentation layer conversions and data copying [Gok98a]. In one particular test, 42% of the client node's CPU processing time was spent in copying and marshaling the data. An additional source of overhead in several CORBA implementations is the routing of invocations to their handlers at the server side. A detailed study shows several reasons for this overhead. For example, long chains of function calls may occur due to the layered structure of the CORBA implementation. Furthermore, it was found that linear searching and costly string comparisons were used in one of the CORBA infrastructure implementations. For this particular product, almost 72% of the CPU time at the server node was spent for invocation routing and demarshaling.

The above findings led to the implementation of TAO, a high-performance real-time CORBA implementation with several improvements over conventional products [Gok98a, Sch98b]. For example, invocation routing bypasses the logical ORB layers and accesses directly the target methods. Also, presentation layer conversions are minimized by supporting multiple encoding strategies depending on the application's needs. In addition, excessive data copying and long chains of function calls are reduced by using special compiler optimizations that automatically omit unnecessary copying of data between the CORBA infrastructure and the application. These optimizations offer significant performance improvements over traditional implementations. For example, when sending data structures over the network, the latency of the enhanced IIOP protocol implementation is approximately one fourth of the latency imposed by the original protocol implementation obtained from SunSoft.

Performance benchmarking with well-defined workloads is an important area of research. A report produced by the Charles University makes a thorough comparison between three CORBA implementations [MLC98]. The comparison reveals significant performance differences between the products. For simple data types, the longest response time was 5.9 times longer than the shortest response time. For a large array of complex data types, the slowest product was 27 times slower than the fastest one. The study also reveals an important observation that is common to all measured products: the overhead of an invocation overshadows the impact of argument sizes and types, unless passing a very large number of complex arguments. Recently, the Object Management Group has investigated the possibility of defining guidelines for conducting CORBA benchmarks [OMG99a].

Performance modeling frameworks

A classical example of an integrated framework for performance modeling is described in [Smi90] as a part of a complete SPE approach. Two separate notations are proposed for performance models: *execution graphs* represent the software structure and *information processing graphs* model the overall system. A set of modeling techniques is described for commonly occurring situations in software engineering. Also, the requirements for tool support are discussed, and an example tool is presented. Finally, the approach presents a methodology for gathering data in order to produce the performance models. In a later work, a methodology is proposed for transforming object-oriented designs manually into performance models that fit into this framework [Smi97].

A repository-based performance modeling framework is presented and discussed in [Wat97]. The framework uses a common repository for integrating a number of heterogeneous modeling tools and techniques. Standard object-oriented notation is used for representing the application structure, but proprietary notations are defined for specifying workloads and the execution environment. The models are solved with a discrete event simulation tool. The framework's strength is in the possibility to use separately developed tools. However, if general-purpose tools are used – as preferred by software engineers – the support for performance modeling is not necessarily sufficient for all performance engineering purposes. Also, the processing time for making the necessary transformations between the tools is relatively long (i.e. in the order of minutes) even for simple models in the proposed framework implementation. This may reduce the framework's usability for iterative development.

An extensive performance modeling framework is being developed on the basis of layered queuing networks. In [Sho98], an complex example is presented for applying performance modeling to a telecommunication system. The example outlines an approach that aims at making performance engineering more accessible to software developers. The paper also describes requirements for a performance oriented design tool. In a recent paper, Petriu and Wang propose an approach for transforming high-level UML diagrams into layered queuing networks [Pet99]. In this approach, communication-related architectural patterns in the system design are automatically converted into LQN diagrams using a graph rewriting system. The paper describes transformations for most communication patterns occurring in distributed systems. The advantage of this approach is the use of the well-known UML notation for representing an abstract view of the architecture, while the LQN representation allows the model to be solved with existing algorithms.

A closely related issue is the creation of performance models from non-UML notations, such as the Specification and Description Language (SDL). In particular, Steppler proposes to use an SDL based design methodology for producing formal specifications for communication systems, and to use the SPEET tool for evaluating these systems by means of simulation and emulation [Ste98]. While the proposed approach is well suited for communication oriented systems, it may be less usable for complex software systems where communication is only one of the elements that affects the overall performance.

1.4 Outline of the work

The rest of this work is structured as follows. Chapter 2 presents the CORBA platform and discusses some performance aspects of CORBA based applications. Also, it formulates additional technical requirements for the framework. Chapter 3 provides an overview of the framework architecture. On one hand, it defines the main elements of the framework and, on the other hand, it describes the required performance model representations. Chapter 4 starts with an introduction to queuing networks. Then, augmented queuing networks are defined for providing better support for distributed CORBA based systems. Finally, an algorithm is described for solving augmented queuing networks. Chapter 5 is dedicated to UML based performance modeling. It starts with an introduction to UML, and continues with the presentation of UML based performance modeling notation and a set of UML based performance modeling techniques for CORBA based systems. Chapter 6 describes a concise performance modeling methodology that allows the framework to be used with object-oriented analysis and design approaches. Chapter 7 discusses a tool prototype and presents a case study with some practical results. Chapter 8 provides concluding remarks and proposes plans for future work.

Chapter 2

The Common Object Request Broker Architecture

In this chapter, we briefly present the CORBA platform and discuss its future development. In addition, we point out some general performance issues in CORBA based applications, and review several performance improvement techniques at the application level. Finally, we discuss technical requirements for our performance modeling framework imposed by the technical choices in the CORBA platform.

2.1 Introduction to CORBA

The Object Management Group (OMG) is a consortium of more than 800 organizations attempting to create a common technology base for object-oriented distributed systems. The objective of the OMG is to promote portability, reusability, and interoperability of software through the use of object-oriented technologies. The OMG pursues its goals by providing a common architecture and a set of specifications [OMG92].

The OMG started its work by specifying the Object Management Architecture (OMA) to provide an overall framework for its activities and further specifications [OMG92]. The OMA defines a reference model that divides the problem space of distributed systems into four distinct sub-spaces, as illustrated in Figure 1. Two of them, the application objects and the CORBAfacilities, are concerned with application level issues, while the other two, the object request broker and the CORBAservices, concentrate on the basic technology for distributed computing. For system providers, the reference model offers a conceptual framework for identifying their role in the marketplace. For user organi-

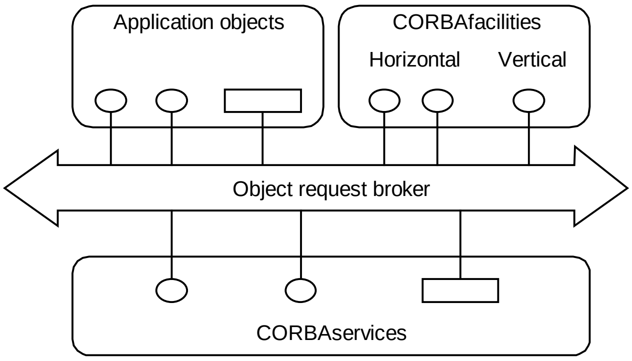


Figure 1. The OMA reference model [OMG92].

zations, the reference model gives a high-level architecture according to which they can partition their systems.

The *object request broker* (ORB) enables communication between different parts of the system, and isolates them for each other so that heterogeneous technologies can be used. The technical details of the ORB are given separately in the Common Object Request Broker Architecture (CORBA). In this work, we concentrate on version 2.3 of the CORBA specification [OMG99d].

The *CORBAservices* define a set of general-purpose building blocks for helping the work of application programmers. On one hand, they increase programmer productivity by offering ready-made solutions for commonly occurring programming needs and, on the other hand, they increase application portability by offering these solutions in the same way in all environments.

The *CORBAfacilities* provide generic frameworks and high-level building blocks for CORBA based systems. Horizontal CORBAfacilities are intended to cover domain-independent needs, such as systems management, internationalization, user interface management, and work process automation. Vertical CORBAfacilities provide high-level building blocks for specific application domains, such as the telecommunications industry.

Application objects in the OMA reference model represent functions and services that are specific to a particular application. The design principles and implementation techniques for application objects and CORBAfacilities are similar, but their scope is different: application objects

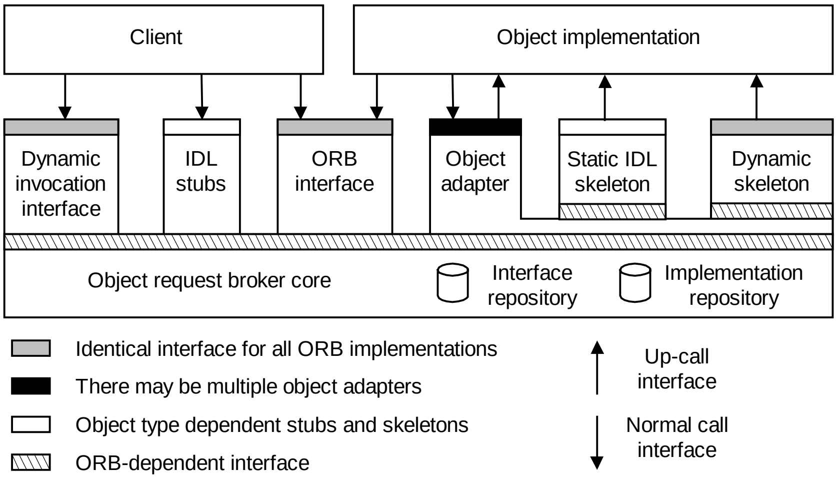

Client
Object implementation
Dynamic invocation interface
IDL stubs
ORB interface
Object adapter
Static IDL skeleton
Dynamic skeleton
Object request broker core
Interface repository
Implementation repository
Identical interface for all ORB implementations
There may be multiple object adapters
Object type dependent stubs and skeletons
ORB-dependent interface
Up-call interface
Normal call interface

Figure 2. The Object Request Broker [OMG99d].

are only intended for a particular application or a system, and they are not described in OMG's specifications.

The technical structure of the ORB and its interfaces are illustrated in Figure 2. A CORBA *client* can be anything that has the ability to request operations on objects. For example, ready-made software packages, end-user applications, and scripts can all be CORBA clients. An *object implementation* is an executable system component that creates one or more CORBA objects and provides the necessary run-time support for performing operations on them. An object implementation contains definitions for creating the objects, and program code for carrying out the operations. The implementation of an operation is called a *method*. Object implementations are the concrete components responsible for realizing the service abstraction that the clients perceive through object interfaces.

IDL stubs allows clients to invoke operations. A stub accepts the client's invocation as a local call, transforms the invocation into a network message, and sends it to the appropriate object implementation. For many compiled programming languages, there is a separate stub for each operation in an interface, and the stubs are linked permanently to the client executables. The implementation details of the stubs depend on the selected technology and the operating environment. The stubs are responsible for encoding the request parameters, dispatching the requests to the object implementations, and decoding the subsequent replies. In addition, the stubs report exceptions to the clients so that they can handle abnormal conditions in a controlled way. Exceptions may be raised by the ORB or by the object implementation.

An *object adapter* allows the ORB core to deliver operation requests to the appropriate object implementations. In addition, it allows object implementations to use the ORB's services. An object adapter supports operations for registering new objects, activating and deactivating existing objects, and accessing object information maintained by the ORB. Currently, the CORBA specification defines only one object adapter, the portable object adapter (POA), for supporting objects that are implemented with ordinary programs and processes. However, additional object adapters may be defined for other kinds of objects, such as those using an object database [Rev96].

An *IDL skeleton* is the server-side counterpart for an IDL stub. It receives operation invocations from the ORB and transforms them into local calls to the object implementation. The detailed structure of the IDL skeleton depends on the object adapter and the object implementation's programming language. For many compiled languages, there is a separate skeleton for each operation in an interface, and the skeletons are linked permanently to the object implementation executables. The skeleton is responsible for hiding all communication details from the programmer who implements the objects. The hidden details include decoding the request parameters, encoding the reply, sending it to the client, and treating abnormal situations that may result from communication failures.

The *dynamic invocation interface* (DII) allows clients to invoke operations that were not known at the time of compiling the client executable. When using the DII, the client does not need an IDL stub. Instead, it makes a series of calls to the DII and thereby gives a detailed description of the operation request. The client must identify the target object, the operation to be invoked, the parameter values, the parameter types, the type of the expected result, etc. When the request has been fully specified, the client sends it to the object implementation with a separate call and examines the results when the invocation has been completed.

The *dynamic skeleton interface* (DSI) allows object implementations to accept requests that were unknown at the time of compiling the application. Hence, there is no need for interface-specific IDL skeletons when using the DSI. If an operation request is received through the DSI, it is converted into a data structure that contains all necessary information for interpreting the request. When the operation has been completed, the results are returned to the DSI that transforms them into a network message and sends them to the client. The client has no way of knowing whether the request has been executed through the DSI or through an IDL skeleton at the server side.

The *object request broker core* is responsible for delivering requests and replies between clients and object implementations. In addition, the ORB core also maintains location and access transparencies for CORBA objects. For this purpose, CORBA objects are never identified directly through their network address. Instead, the ORB core generates and maintains a unique identifier, *object reference,* for every object under its control. In addition, the ORB implements an internal location service that maps object references to the corresponding network addresses. The ORB core also takes care of other technical activities, such as the enforcement of secure communication and the maintenance of implicit status information during transaction processing.

The *implementation* and *interface repositories* maintain information about the object implementations and interfaces that are available in the system. Finally, the *ORB interface* supports general-purpose operations that can be used by the clients and object implementations. For example, there are operations for initializing the ORB and for finding out a set of initial object references that allow the applications to start their operation.

The CORBA specification also defines a number of additional technologies for supporting the use of ORBs in different environments. The *OMG Interface Definition Language* (IDL) allows system developers to specify object interfaces in a way that is independent of any specific programming language or operating environment. A number of *language mappings* indicate how the interfaces specified with the IDL are used in different programming language environments. The mappings define how IDL data types and interfaces are represented, and how the corresponding operations are invoked and implemented. Version 2.3 of the CORBA specification contains language mappings for Ada, C, C++, Cobol, Java, and Smalltalk. CORBA implementations use precompilers to generate IDL stubs and skeletons that conform to the specified language mappings.

The interoperability architecture in the CORBA specification describes how different ORB products can cooperate. At the heart of the architecture, the GIOP and IIOP protocols provide the means for sending requests and replies across technology boundaries. The GIOP defines the transfer syntax for requests and replies, and the IIOP specifies how GIOP messages are transmitted using the TCP/IP protocol. In addition, the *interworking architecture* is specified to provide a mapping between CORBA and the Component Object Model (COM) from Microsoft.

The ORB provides adequate means for implementing communication between system elements, but it provides limited support for application development in other respects. To compensate this insufficiency, the CORBAservices define a comprehensive set of low-level building blocks for system developers [OMG98a]. We briefly mention some of the most important services. The *Naming Service* allows applications to give names to objects and to organize them into hierarchical name spaces. The *Event Service* provides basic support for asynchronous and many-to-many communication. Recently, the *Notification Service* has been defined to enhance the functionality of the Event Service in telecommunications environments [OMG99e]. The *Life Cycle Service* provides mechanisms for creating, moving, copying, and deleting objects. The *Concurrency Control Service* regulates access to objects so that their consistency is not compromised by requests coming from multiple concurrent computations. The *Transactions Service* supports transactions in distributed environments. The *Security Service* supports the development of secure distributed applications by providing a set of general security facilities. The *Trading Object Service* supports the advertising and discovery of services available in the system.

2.2 Future directions for CORBA

The CORBA specification is currently evolving towards the next generation and a number of new elements are being added to it. We briefly discuss three important elements that are likely to influence the performance of CORBA based applications.

The first important addition is the *CORBA Messaging Specification* [OMG99f]. It extends the current call-reply communication model with several asynchronous invocation modes. The reply for an asynchronous invocation can be obtained either by polling or by providing a callback interface that receives the reply as a return call. With the *time independent invocation* model, it is possible that the original client no longer exists when the reply is returned and accepted by some other entity in the system. The specification also defines a generic framework for controlling the quality of service (QoS) of communication. For example, it is possible to control message ordering, deadlines, time-to-live, routing policy, and hop count. The QoS framework can also be used for controlling the normal synchronous communication.

Another important new element is the *Real-Time CORBA Specification* [OMG99g]. It regulates the control of critical resources, such as threads, protocols, and connections, so that systems can be built to meet hard and statistical real-time requirements. In addition to defining an interface for the real-time ORB and a number of interfaces for controlling the essential resources, the specification also describes the *Scheduling Service*. Its purpose is to abstract away many of the low-level details that are needed for enforcing fixed priority real-time scheduling policies. Currently, the specification does not incorporate dynamic scheduling.

Finally, the CORBA Component Model (CCM) specification provides a comprehensive framework for developing, deploying, and using CORBA based software components [OMG99h]. The CCM combines a number of essential features from several CORBAservices and allows them to be used through high-level abstractions. Unlike traditional objects in the CORBA environment, components are allowed to support multiple interfaces. The Component Implementation Definition Language (CIDL) has a key role in the specification, since it allows developers to specify various aspects of security, transaction processing, persistence, and event handling without writing program code. The CCM also defines a programming model for supporting the run-time control of components. Finally, the specification defines an XML based distribution format for supporting the distribution, aggregation, and deployment of components.

2.3 Performance issues in CORBA based applications

The performance of CORBA based applications may be threatened in a number of ways. We briefly review some of the threats that are discussed in the literature. The following topics are discussed:

- Distribution transparencies,
- Marshaling and demarshaling of parameters,
- Invocation routing,
- Network bandwidth and latency,
- The use of network connections,
- Server contention,
- Server activation and deactivation,
- The effect of ORB abstractions on the design work.

Distribution transparencies are an important source of performance problems. They are often successful in hiding the complexity of the underlying techniques, but some of the performance implications cannot be hidden. Moreover, transparencies may imply additional processing that is not easy to predict. For example, there may be additional lookup messages for locating an object. The “black box” approach adopted by many CORBA vendors makes it difficult to analyze the impact of distribution transparencies on performance. Proposals have been made to allow applications to gain control over middleware performance. For example, it has been argued that all distribution transparencies need standardized access and control mechanisms for managing the performance aspects of the underlying transparency implementations [Mar94].

The *marshaling and demarshaling* of network messages are a common source of performance problems [Gok98b]. These activities take place during every CORBA interaction and, consequently, small inefficiencies may build up into serious performance problems. Unfortunately, marshaling and demarshaling is often beyond the control of application programmers. Hence, it may be difficult to bypass or fix inefficient marshaling code of an ORB implementation.

Invocation routing refers to the actions that take place when an operation request arrives at the server node and gets routed to the correct destination. A special daemon process is often used for locating the object implementation and for delivering the request. Daemons add processing overhead and may become performance bottlenecks in highly loaded environments [Sch97, Gok97]. The scheduling of threads and processes at the server node may also cause performance problems [Sch98a].

Network bandwidth can be a limiting factor in some applications, such as those running in wireless mobile environments [OMG98c] and those transferring large amounts of data over the network [Gok98a]. *Network latency* is also a well-known issue, but its effect on the application behavior can be reduced with well known techniques, such as caching and pre-fetching. However, there is an additional problem in CORBA based systems: unpredictability due to distribution transparencies. Depending on the configuration, an invocation may sometimes take several seconds (e.g. to an object behind a congested Internet connection) and sometimes only a few microseconds (e.g. to an object in the same process).

Inefficient use of *communication resources* has been observed to cause performance problems. For example, ORB implementations may close and reopen TCP/IP connections unnecessarily, and use multiple connections when it might be more efficient to multiplex a single connection. The optimal way of using communication resources is application dependent and, therefore, some form of control is needed for application developers. Some ORB implementations allow connections to be controlled with configuration parameters (e.g. [Inp99]).

Contention at software servers is a well-known performance issue. In the CORBA platform, this may be avoided by using a multi-threaded object implementation instead of a single-threaded one. The correct choice depends on several factors, such as the implementation cost, application workload and performance requirements. However, there is an additional concern related to the CORBA platform: contention may occur at internal servers that are not visible for application developers. Even if application-level servers were designed to withstand high workloads, the system may fail to reach its performance goals due to excessive contention at internal servers.

The *activation and deactivation of CORBA servers* may also cause performance problems. To allow applications to optimize their use of hardware and software resources, the CORBA specification supports various activation and deactivation mechanisms and policies. However, it may be difficult to make correct decisions as to when and where a particular server should be activated. Some CORBA implementations make these decisions transparently from applications (e.g. [BEA99]), but it may be difficult to configure such mechanisms so that they correctly predict the behavior of the applications.

Finally, it should be noted that the high-level abstractions provided by CORBA platform might sometimes mislead application developers to ignore the physical reality. Distribution transparencies together with object-orientation can make the implementation of remote interactions and distributed systems straightforward and easy. However, the cost of remote interactions and other potential performance problems do not vanish with the improved programmer productivity.

2.4 Application-level performance heuristics

A number of application-level guidelines have been proposed for improving the performance of CORBA based systems. They are often presented with *design patterns*, a popular technique for documenting software design expertise in a structured way [Gam94, Cop96]. We briefly discuss the following heuristics:

- Increasing the amount of parallel execution,
- Reducing the number system elements,
- Reducing the amount of communication,
- Improving the architecture of CORBA based systems.

When discussing the guidelines, we also point out the underlying technical motivations in the CORBA architecture. A more generic set of application-level heuristics is given in [Smi90].

Increasing parallelism

Synchronous calls are the primary method of invoking operations on CORBA objects. In many cases, however, the waiting time at the client side could be used for other purposes. A number of application-level techniques have been proposed to increase parallelism during operation invocations. A common technique is to replace synchronous operation invocations with fully or partially asynchronous communication [Mow97]. For example, the server can invoke a callback routine on the client when the requested operation has been completed. The CORBA Messaging specification provides advanced tools for implementing such techniques.

Another technique for increasing parallelism is to use threads with synchronous communication. At the server side, the object adapter can launch a new thread for each incoming request. This technique is transparent to the application code apart from the need to synchronize the simultaneous access to the object's internal state. At the client side, a special thread can be spawned for waiting for the reply while other threads continue as usual. A special technique has been proposed for hiding the use of threads in CORBA clients [Hel96]. When the client application invokes an operation, a new thread is launched transparently for waiting for the server's reply. The client's main thread continues executing the application code until it attempts to use the results of the pending invocation. If the server has already delivered the results, the client application code simply accesses them. However, if the server is still executing the request, the client blocks until the results become available.

Reducing the number of system elements

A large number of generated IDL stubs and skeletons increase the overhead of routing invocations to object implementations. Compiler optimization techniques can alleviate the problem [Eid97] but they cannot eliminate it completely. Several possibilities exist for reducing the number of interfaces, operations, and attributes in CORBA based applications. For example, an interface can offer two generic operations, *get* and *set*, for supporting access to any number of attributes [Mow97]. Similarly, it is possible to reduce the number of operations and interfaces by abstracting away differences that exist between them. The *Any* data type can be used as a placeholder for generic parameters that can change their type from one invocation to another. However, the *Any* data type entails the use of meta-information at run-time and this may incur a serious degradation of performance in some cases [Zie96, MLC98].

Reducing the amount of communication

The amount of remote communication is a typical target for optimizations due to the limitations imposed by network bandwidth and latency. There are several techniques for reducing remote communication in CORBA based systems. We briefly discuss three of them. First, the client stubs can be modified to maintain a local cache of remote objects [Mow97, Inp99]. This way, access to the remote object is minimized but the client can still maintain the abstract view provided by the IDL interface. Second, an application-level object can be divided into two half-objects that use an optimized protocol for maintaining the necessary state information in both halves [Mes94]. This way, both half-objects can locally support non-optimized CORBA interfaces, but the overall design can avoid most of the related overhead.

The third technique for reducing the communication overhead is to replicate CORBA objects in the network. This way, clients can invoke operations locally while the replicated servers synchronize in the background. Some ORB implementations provide ready-made solutions for creating multiple implementations for a single object reference [DEC96], but standard CORBA facilities can also be used for implementing object replication [Mow97]. An important use of replication is the implementation of high availability through *virtual synchrony* [Maf97]. This technique uses a group abstraction mechanism for ensuring that the request of a client succeeds if there is at least one operational replica in the group. In process-oriented applications with very little persistent data, virtual synchrony can implement reliable system behavior with much less overhead compared to traditional ACID type transaction processing.

Improving the software architecture

Correct IDL design may strongly influence the performance of the system. In particular, it is usually more efficient to use a single invocation for carrying large amounts of data instead of having multiple invocations with less data [MLC98]. Accordingly, if a long sequence of calls to primitive operations is replaced by a single and more complex invocation, the responsiveness of the system may increase significantly [Sla99].

The CORBAservices have been designed to support general-purpose needs in various application domains and, hence, they may be inefficient for some applications. An application-specific layer can be introduced between the client and a general-purpose CORBAservice for reducing the number of calls and the amount of data to be transferred. For example, the context-relative CORBA naming service can be hidden under a simplified interface that uses a single naming context [Mow97].

2.5 Technical requirements for the framework

We can now formulate additional technical requirements for our performance modeling framework. The following requirements reflect the architectural choices of the CORBA specification and the possibilities that it offers for implementing information systems:

- Support for complex hidden interactions,
- Support for flexible changes in configurations,
- Interface support,
- Support for heterogeneity.

Complex hidden interactions emerge from the implementation of distribution transparencies. A simple application-level interaction may entail interactions between infrastructure objects for locating, activating, and accessing the relevant application objects. Such activities impact the performance of the system but are not visible at the application level. The performance modeling framework should support the representation of such mechanisms without forcing them to be mixed with the application logic. Different CORBA implementations use different mechanisms and, consequently, the framework should not be limited to a single set of mechanisms.

In many CORBA based systems, it is possible to modify the run-time configuration without any changes in the logical application architecture or program code. For example, the locations of application objects can be changed so that remote interactions turn into local ones and *vice versa*. Similar flexibility should be available in the performance modeling framework so that developers can evaluate application designs in different run-time configurations. In particular, this should be possible without any changes in the model at the application level.

Interfaces and the IDL language have a central role in the design of CORBA based systems and, consequently, the framework should allow developers to specify performance models for designs that use the IDL as the main specification language. In particular, there should be modeling constructs for directly representing interfaces and operations as they appear in the IDL.

Finally, the performance modeling framework should support heterogeneity wherever it is allowed in the CORBA platform. For example, a single model should be able contain heterogeneous hardware elements and network connections. In particular, there must be a way to model different CPU speeds and network latencies. In addition, it should be possible to model systems consisting of multiple CORBA environments having different internal implementations. Moreover, the framework should not depend on any programming language specific notation since CORBA itself is programming language independent.

2.6 Summary

In this chapter, we have given an overview of the CORBA 2.3 platform and related CORBAservices. In addition, we have presented important elements in the next generation of the CORBA platform. A number of potential performance problems in CORBA based system have been discussed, and possible application-level techniques have been proposed for reducing their adverse effect. Finally, we have formulated a set of additional technical requirements for the performance modeling framework.

Chapter 3

Performance modeling framework architecture

In this chapter, we present the architecture of our performance modeling framework. On one hand, we identify the main elements of the framework and discuss briefly their mutual relationships. On the other hand, we present the four performance model representations that are used by the framework. This provides an introduction to the technical details of the framework.

3.1 Elements of the framework

The performance modeling framework can be divided into the following five elements:

- A UML based modeling notation,
- A set of performance modeling techniques,
- The method of decomposition for solving the models,
- A performance modeling and analysis tool,
- A performance modeling methodology.

These elements and their relationships are illustrated in Figure 3.

The *performance modeling methodology* links the framework to the software engineering process. It indicates how to obtain performance related information from the requirements specifications, and how the performance modeling techniques can be used at different stages of systems development to produce useful performance models. A layered model structure is also defined so that developers can divide performance models into more manageable sub-models. The use of layers also allows a

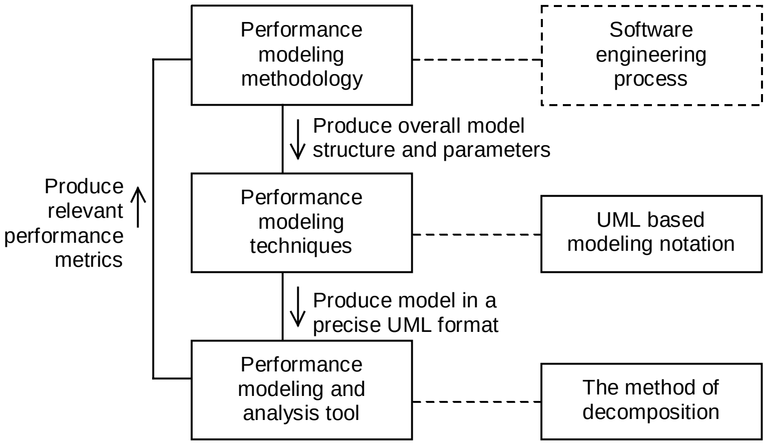

Performance modeling methodology
Software engineering process
Produce overall model structure and parameters
Produce relevant performance metrics
Performance modeling techniques
UML based modeling notation
Produce model in a precise UML format
Performance modeling and analysis tool
The method of decomposition

Figure 3. Elements of the performance modeling framework.

complex model to be built in a stepwise process that proceeds in parallel with software development.

The *UML based performance modeling notation* is a subset of the full UML notation with a few extensions for indicating performance related information in otherwise normal UML diagrams. The proposed extensions comply with the standard UML extension mechanisms. The notation prepares the ground for applying the actual performance modeling techniques.

The proposed *set of performance modeling techniques* provides sufficient means for creating precise performance models of complex distributed systems based on the CORBA platform. In particular, a number of techniques allow application level issues to be separated from the infrastructure and the network. The proposed techniques are close to the normal UML modeling style as presented in the UML standard and literature (e.g. [Rat97, Eri98, Jac98]). Therefore, they allow existing functional UML models to be extended into performance models.

The *performance modeling and analysis tool* automates some of the tasks required by the framework. In particular, it transforms UML based performance models into a solvable format, produces an approximate solution for the performance model, and converts the solution into a set of relevant performance metrics to be used in the performance modeling methodology. This is the current functionality of our prototype tool. However, a full tool implementation could have additional features, such as capabilities for graphical modeling and for the visualization of the metrics obtained for the models.

The method of decomposition (MOD) provides the foundation for the framework since it describes a decomposition technique and an iterative algorithm that allows us to solve the performance models that result from UML based modeling techniques. The key element in the algorithm is the support for simultaneous resource possessions that arise, for example, from synchronous operation invocations. The MOD alone, however, is unsuitable for modeling large distributed systems due to its low-level representation of the modeled system. Hence, the method of decomposition is used as an underlying technique for solving the higher-level UML based models.

3.2 Four performance model representations

The technical aspects of the framework and the operation of the modeling and analysis tool can be described in terms of four performance model representations. The framework defines mappings between the representations, as shown in Figure 4. The idea is to start from the UML representation and proceed downward using the mappings. Once the bottom has been reached, an approximate solution can be found for the model. The mappings also indicate how the obtained metrics can be propagated upwards.

The *UML representation* describes the system with UML diagrams. This representation may contain purely functional elements that are not needed for performance modeling. To reduce the complexity of the diagrams, we assume that the UML representation is divided into separate layers corresponding to different parts of the system, such as the application, the infrastructure, and the network.

The *PML representation* (Performance Modeling Language) provides an accurate textual notation for representing performance related elements in the UML diagrams. The PML representation has the same layered structure as the UML representation, and the mapping from UML to PML is straightforward. The purpose of this representation is to filter out those features that have no significance for performance modeling, such as graphical UML variations and purely functional parts of the UML model. Moreover, the PML representation has an important role in the development of the framework, as it is currently the input format for the prototype implementation of the modeling and analysis tool. However,

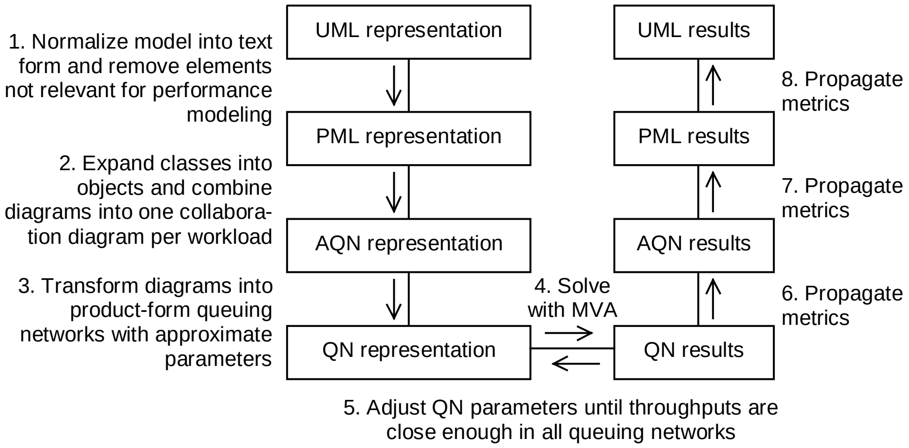

1. Normalize model into text form and remove elements not relevant for performance modeling
UML representation
UML results
8. Propagate metrics
PML representation
PML results
2. Expand classes into objects and combine diagrams into one collaboration diagram per workload
7. Propagate metrics
AQN representation
AQN results
3. Transform diagrams into product-form queuing networks with approximate parameters
4. Solve with MVA
6. Propagate metrics
QN representation
QN results
5. Adjust QN parameters until throughputs are close enough in all queuing networks

Figure 4. Four performance model representations.

the use of PML is not mandatory, and any other human-understandable UML representation could be used instead. For example, we might later opt for the human-usable textual UML notation that is currently being developed by the OMG [OMG99b]. For the purposes of this work, however, the current PML based approach is sufficient.

The *AQN representation* (Augmented Queuing Network) describes the target system as a queuing network that can contain simultaneous resource possessions and, therefore, is not necessarily product-form. This representation allows the use of the method of decomposition for solving the model. The AQN representation is obtained from the PML representation by expanding object classes into object instances that correspond to individual resources in the system. Moreover, the UML collaboration and sequence diagrams describing the behavior of the application and the infrastructure are combined into one or more workload specifications. Each workload specification can be visualized with a collaboration diagram that indicates how the system's resources are used by that particular workload.

The *QN representation* (Queuing Network) consists of product-form queuing networks with mutual dependencies that relate them to the same overall system. The QN representation is obtained from the AQN representation during the initial steps of the method of decomposition. The resulting queuing networks contain a number of unknown parameters and, to solve the networks while respecting the mutual dependencies, iteration has to be used. The transformation from the AQN representation into the

QN representation involves a number of approximations that are needed to make the queuing networks solvable with efficient algorithms.

Similar tiered architectures are common in performance modeling [Agr85, Hav98], but the top representation is often a sophisticated performance modeling notation, such as a variant of stochastic Petri nets, to support advanced modeling techniques. In our case, a non-technical top representation is used for hiding most of the underlying performance modeling issues and for making the models closer to the functional models used by software engineers.

Chapter 4

The method of decomposition

This chapter extends traditional queuing networks to support the modeling of CORBA based distributed systems. We start with a brief introduction to queuing networks, and present a number of algorithms for solving them. We focus on approximate algorithms that can solve large models efficiently. Next, we define augmented queuing networks that contain extensions to cope with the needs of CORBA based distributed systems. Finally, we present the method of decomposition for solving augmented queuing networks. It combines existing decomposition and approximation techniques in a new way for dealing with needs of our framework.

4.1 Introduction to queuing networks

Queuing networks provide a straightforward way to model computer systems at an abstract level so that a number essential performance characteristics can be represented in an easy-to-use manner. The static structure of a system is represented in terms of *resources*[1] that correspond to hardware devices, such as CPUs or disks, and software resources, such as CORBA object implementations. It is assumed that access to each resource is controlled by a queue of requests. A resource is characterized by its scheduling discipline and service time distribution. The dynamic aspects of the system are modeled with *jobs* that correspond to threads of execution accessing system resources. A job can represent, for example, a series of actions requested by an end user or a background task running in the system. *Routing probabilities* indicate how jobs move from one resource to another, and *service demands* indicate the average amount of

[1] Resources in queuing networks are often referred to as *servers* or *service centers*. However, to avoid confusion with CORBA servers, we use the term *resource*.

service per visit, in time units, that jobs require from the resource. Service demands can be obtained, for example, through measurements or by examining the program code for the jobs. Service demands are also a good candidate for calibrating a model to match the measurements from an existing system. See Section 6.6 for more details.

In this work, we consider *multi-class* queuing networks, where jobs are partitioned into several *classes* according to the way they are using resources. Each class consists of independent jobs that have the same pattern of using resources. We distinguish between two kinds of classes: *closed classes* have a fixed number of jobs, and *open classes* have a variable number of jobs that may enter and leave the system. A *closed* queuing network contains closed classes, an *open* queuing network contains open ones, and a *mixed* queuing network has both kinds of classes.

To ensure that a queuing network can be solved efficiently, a number of assumptions must be made so that the requirements for *product-form* queuing networks are met. In a product-form queuing network, the probability of a state of the system can be represented as a product of resource-specific factors that only depend on the type and state of the resource, without any side effects across resources. Product-form queuing networks are also called *separable* since the states of the individual resources can be considered separately.

Baskett, Chandy, Muntz, and Palacios have shown that a relatively large set of multi-class queuing networks (referred to as *BCMP networks*) satisfies the product-form requirements [Bas75]. BCMP networks can have resources with the following four scheduling disciplines:

- First-come-first-served (FCFS),
- Processor sharing (PS), where incoming jobs start receiving service immediately with a rate that is inversely proportional to the number of jobs currently being served,
- Preemptive-resume last-come-first-served (LCFS), and
- Infinite server (IS), where incoming jobs start receiving service immediately at the nominal rate.

Resources in the three first types are collectively called *queuing* resources, and those of the last type are called *delay* resources. For FCFS resources, the service time must be exponentially distributed and the distribution must be the same for all classes of jobs. For other types of resources, each class of jobs can have a different service time distribution, and the distributions must be Coxian. This requirement it is equivalent to requiring that the distributions have rational Laplace transforms [Cox55].

The PS scheduling discipline is a limiting case of the more practical round robin (RR) discipline. In RR scheduling, each job is given a fixed quantum of service time. If the job does not finish execution during the allocated time quantum, it is placed at the end of the queue and the next job is taken into service. When the quantum in the RR discipline approaches to zero, the RR discipline becomes the PS discipline. Variants of the RR scheduling are commonly used in time-sharing operating systems and, hence, PS can be considered as a reasonable approximation for modeling processor scheduling in actual operating systems.

For open classes in BCMP networks, the time between successive arrivals should be exponentially distributed and no bulk arrivals are permitted. Two types of arrival processes are supported. In the first type, the total arrival rate to the network is Poisson and the mean rate is allowed to depend on the total number of jobs in the network. In the second type, there is a separate Poisson arrival stream for every open class in the network, and the arrival rate is allowed to depend on the number of jobs in the class itself. It is also essential that the jobs in the network are independent of each other, and they advance in the queuing network with routing probabilities that do not depend on the state of the network.

The class of product-form networks has been further extended by several authors (e.g. [Kel76, Bar76], see [Con89] for a discussion). Furthermore, Denning and Buzen [Den78] have formulated a set of operational requirements that lead to product-form queuing networks. Unlike the stochastic assumptions for BCMP networks, these requirements state operationally testable conditions for the behavior of the resources. For example, the system is required be flow balanced (i.e. the number of arrivals for a resource must be the same as the number of departures for a given observation period) and the resources must be homogeneous (i.e. the routing of the jobs must be independent of queue lengths, and the service completion time must not depend on the queue length of other resources). For the purposes of this work, however, the BCMP requirements are adequate for two reasons. On one hand, they provide a sufficient foundation for justifying approximations that are needed for modeling CORBA based distributed systems. On the other hand, they have been widely used for performance modeling and the predicted performance metrics are relatively accurate even when the requirements are not completely valid for the modeled system.

4.2 Solutions for product-form queuing networks

We now present exact and approximate algorithms for solving open, closed and mixed multi-class queuing networks that satisfy the product-form requirements. These algorithms are needed later when we describe the details of the method of decomposition. In order to avoid computationally unattractive algorithms, we limit ourselves to average performance measures. Hence, we do not discuss algorithms for solving steady-state probabilities of the networks since this would essentially require solving the underlying continuous-time Markov chains. See [Hav98] and [Con89] for additional algorithms for solving product-form queuing networks.

The first algorithm solves open product-form queuing networks. This algorithm is based on two fundamental results: the arrival theorem for open product-form queuing networks [Sev81] and Little's law [Lit61]. The arrival theorem states that a job arriving at a resource sees the resource's queue in its steady-state distribution. Little's law in turn describes a direct relationship between the average queue length n, the throughput X, and the average residence time R:

$$n = XR.$$

Little's law can be applied to individual classes and to the aggregate of all classes in the model. The above results lead to a straightforward algorithm for the performance metrics of open product-form queuing networks. A derivation can be found, for example, in [Men94]. In the algorithm, the service demands of individual visits to a resource are combined into *total service demand*. It can be obtained from a queuing network description by summing up the individual service demands multiplied by their execution probabilities.

Algorithm 1. Exact solution for open product-form queuing networks. The input parameters are the number of resources K, the number of classes R, the arrival rate λ_r for each class r, and the total average service demand $D_{i,r}$ for each resource i and class r.

Step 1. Compute the utilization $U_{i,r}$ for class r jobs in each resource i, and the total utilization U_i for each resource i:

$$U_{i,r} = \lambda_r D_{i,r} \quad \text{and} \quad U_i = \sum_{r=1}^{R} U_{i,r}.$$

To ensure the stability of the model, it is required that $U_i < 1$ for all queuing resources i.

Step 2. Compute the average queue length $n_{i,r}$ for each resource i with class r jobs, and the average total number of jobs n_i for each resource i:

$$n_{i,r} = \begin{cases} \lambda_r D_{i,r} & \text{if } i \text{ is a delay resource} \\ \dfrac{U_{i,r}}{1-U_i} & \text{if } i \text{ is a queuing resource} \end{cases} \quad \text{and} \quad n_i = \sum_{r=1}^{R} n_{i,r}.$$

Step 3. Compute the average residence time $R_{i,r}$ for class r jobs in each resource i, and the average response time R_r for each class r:

$$R_{i,r} = \begin{cases} D_{i,r} & \text{if } i \text{ is a delay resource} \\ \dfrac{D_{i,r}}{1-U_i} & \text{if } i \text{ is a queuing resource} \end{cases} \quad \text{and} \quad R_r = \sum_{i=1}^{K} R_{i,r}. \qquad (1)$$

□

The interaction between jobs in different classes can be observed in formula (1). For queuing resources, the average residence time of a job is inversely proportional to the idle time that remains when the service demand of all classes has been taken into account.

For closed product-form queuing networks, the arrival theorem is slightly different. When a job arrives at a resource, it sees the resource's queue in a state that corresponds to the steady-state distribution of the same network with one less job in its own class [Sev81]. As a consequence, the exact solution for closed product-form queuing networks can be obtained recursively by using the solutions of networks with one less job in each class. This idea is implemented by the exact mean value analysis (MVA) algorithm [Rei80].

Algorithm 2. Exact MVA for closed product-form queuing networks. The input parameters are the number of resources K, the number of classes R, the population N_r for each class r, and the total average service demand $D_{i,r}$ for each resource i and class r.

Step 1. Let $\overline{N} = (j_1, j_2, \dots, j_R)$ be a vector describing the population of each class. Let $n_i(\overline{N})$ indicate the average total queue length at resource i for population vector $\overline{N}$, and let $n_i(\overline{N} - 1_r)$ be the average total queue length at resource i for population vector $\overline{N}$ with one class r job removed. Initialize $n_i(\overline{0}) = 0$ for all $i = 1, \dots , K$.

Step 2. For $j_1 = 0, \dots , N_1; j_2 = 0, \dots , N_2; \dots ; j_R = 0, \dots N_R$ (j_1 changes most rapidly) perform steps 3 to 5.

Step 3. For all classes r and all resources i compute the average residence time $R_{i,r}(\overline{N})$ for the population vector $\overline{N} = (j_1, j_2, \dots , j_R)$:

$$R_{i,r}(\overline{N}) = \begin{cases} D_{i,r} & \text{if } i \text{ is a delay resource} \\ D_{i,r}[1 + n_i(\overline{N} - 1_r)] & \text{if } i \text{ is a queuing resource.} \end{cases} \tag{2}$$

Step 4. For all classes r compute the throughput $X_r(\overline{N})$ for the population vector $\overline{N}$:

$$X_r(\overline{N}) = \frac{j_r}{\sum_{i=1}^{K} R_{i,r}(\overline{N})}.$$

Step 5. For all resources i compute the average queue length $n_i(\overline{N})$ for the current population vector $\overline{N}$:

$$n_i(\overline{N}) = \sum_{r=1}^{R} X_r(\overline{N}) R_{i,r}(\overline{N}).$$

□

The trouble with the exact MVA algorithm is its computational cost due to the recursive use of average queue lengths in formula (2). Hence, the solution for a multi-class model with a population vector $(N_1, N_2, \dots , N_R)$ requires the solution of $\Pi_{r=1}^{R}(N_r + 1)$ separate models, yielding time complexity $O(R \times K \times \Pi_{r=1}^{R}(N_r + 1))$. In real systems where performance issues are relevant, the actual number of classes and jobs is often relatively high, and the cost of using exact MVA may become prohibitive.

Schweitzer has proposed an approximation for breaking the recursion in the exact MVA algorithm (see e.g. [Jai91] and [Men94] for discussions). The approximation is based on the assumption that the number of class r jobs at each resource increases proportionally to the total number of class r customers in the model. Using the above notations, we have

$$\frac{n_{i,r}(\overline{N} - 1_r)}{n_{i,r}(\overline{N})} = \frac{N_r - 1}{N_r}$$

and therefore

$$n_{i,r}(\overline{N}-1_r)=\frac{N_r-1}{N_r}n_{i,r}(\overline{N}). \tag{3}$$

This result can be applied directly to the recursive expression in formula (2). The approximate MVA algorithm starts with a set of initial estimates for queue lengths $n_{i,r}(\overline{N})$ and calculates successive estimates by substituting the recursion in formula (2) with the result in equation (3). Iteration stops when successive estimates for queue lengths are sufficiently close. The initial queue length estimates are obtained by assuming that class r jobs are equally distributed among those resources i that have a non-zero class r service demand $D_{i,r}$.

Algorithm 3. Approximate MVA for closed product-form queuing networks. The input parameters are the number of resources K, the number of classes R, the population N_r for each class r, the total average service demand $D_{i,r}$ for each resource i and class r, and the error tolerance ε for successive queue length estimates

Step 1. For all classes r and all resources i initialize the queue length estimates $e_{i,r}(\overline{N})$ for the population vector $\overline{N} = (N_1, N_2, \dots, N_R)$:

$$e_{i,r}(\overline{N})=\begin{cases} N_r/K_r & \text{if } D_{i,r}>0 \\ 0 & \text{if } D_{i,r}=0, \end{cases}$$

where K_r indicates the number of those resources i for which the service demand $D_{i,r} > 0$.

Step 2. For all classes r and all resources i propagate the estimated average queue lengths to be the current average queue lengths:

$$n_{i,r}(\overline{N})=e_{i,r}(\overline{N}).$$

Step 3. For all classes r and all resources i compute the average queue lengths with one less job in each class:

$$n_{i,r}(\overline{N}-1_t)=\begin{cases} n_{i,r}(\overline{N}) & \text{if } t\neq r \\ \dfrac{N_r-1}{N_r}n_{i,r}(\overline{N}) & \text{if } t=r \end{cases}$$

for all $t = 1, \dots, R$.

Step 4. For all classes r and all resources i compute the average residence time for the population vector $\overline{N}$:

$$R_{i,r}(\overline{N}) = \begin{cases} D_{i,r} & \text{if } i \text{ is a delay resource} \\ D_{i,r}\left[1+\sum_{t=1}^{R} n_{i,t}(\overline{N}-1_r)\right] & \text{if } i \text{ is a queuing resource.} \end{cases}$$

Step 5. For all classes r compute the throughput for the population vector $\overline{N}$:

$$X_r(\overline{N}) = \frac{N_r}{\sum_{i=1}^{K} R_{i,r}(\overline{N})}.$$

Step 6. For all classes r and all resources i compute the new queue length estimates for the population vector $\overline{N}$:

$$e_{i,r}(\overline{N}) = X_r(\overline{N})R_{i,r}(\overline{N}).$$

Step 7. If the maximum relative error for successive queue length estimates satisfies the condition

$$\max_{i,r}\left|\frac{e_{i,r}(\overline{N}) - n_{i,r}(\overline{N})}{e_{i,r}(\overline{N})}\right| < \varepsilon$$

then terminate the algorithm. Otherwise, continue from step 2.

□

The time complexity for each iteration is $O(K \times R)$. Experience has shown that the computation required by approximate MVA is significantly lower than that of exact MVA for complex models. The convergence of the algorithm has been proven by Agrawal [Agr85] and typical errors have been reported to be less than 20% [Hei84]. The approximation proposed by Schweitzer has been further improved by the Linearizer algorithm [Cha82]. However, increased accuracy is achieved at the expense of added cost: the time complexity for each iteration is $O(K \times R^2)$ [Sil90]. In this work, we use the Schweitzer approximation but the framework could also be used with other variants of the MVA algorithm.

We now combine Algorithms 1 and 3 to produce an approximate solution for mixed product-form queuing networks [Men94]. A mixed model is divided into two submodels: the open submodel is formed by the open classes, and the closed submodel by the closed ones. The algorithm starts by solving the resource utilizations in the open submodel. The service demands in the closed submodel are multiplied with an elongation factor to reflect the contention in open classes. The elongation factor is inversely proportional to the idle time remaining from open classes as indicated by equation (4). This is a special case of the approximate *load concealment transformation* discussed in [Agr85]. The closed submodel with elongated service demands is solved with the approximate MVA algorithm. Finally, the residence times of the open submodel are adjusted to reflect the contention of jobs in closed classes. This is done by adding the average queue length of the closed submodel to the number of jobs seen by arriving jobs in the open submodel.

Algorithm 4. Approximate solution for mixed product-form queuing networks. The input parameters are the number of resources K, the open classes numbered from 1 to O (the open submodel), the closed classes numbered from $O + 1$ to C (the closed submodel), the arrival rate λ_r for each open class r, the population N_r for each closed class r, and the total average service demand $D_{i,r}$ for each resource i and class r.

Step 1. Solve the open submodel with Algorithm 1 to get utilizations $U_{i,r} = \lambda_r D_{i,r}$ for all resources i and all open classes $r = 1, \ldots, O$.

Step 2. For all resources i compute the open submodel utilization:

$$U_{i,open} = \sum_{r=1}^{O} U_{i,r}.$$

Step 3. For all resources i and all closed classes $r = O + 1, \ldots, C$ compute the adjusted service demand:

$$D'_{i,r} = \frac{D_{i,r}}{1 - U_{i,open}}. \tag{4}$$

Step 4. Using the adjusted service demands, solve the closed submodel with Algorithm 3. Obtain the following results for all resources i and all closed classes $r = O + 1, \ldots, C$: average residence time $R_{i,r}$, average queue length $n_{i,r}$, and throughput X_r.

Step 5. For all resources i compute the average queue length of closed jobs:

$$n_{i,closed} = \sum_{r=O+1}^{C} n_{i,r}.$$

Step 6. For all resources i and open classes $r = 1, \ldots, O$ compute the average residence time:

$$R_{i,r} = \frac{D_{i,r}(1 + n_{i,closed})}{1 - U_{i,open}}.$$

□

4.3 Queuing networks and distributed objects

Queuing networks provide an established tool for analyzing the performance of centralized information systems, but they are also suitable for modeling object-oriented distributed systems. Objects in CORBA based systems and resources in queuing networks have at least two important similarities. First, both provide abstractions for hiding the details of internal structure and implementation techniques. For objects, the abstraction is given by the set of supported interfaces and operations. For queuing network resources, the abstraction is given by the service demands. Second, both have scheduling disciplines that determine how incoming operation requests or jobs are served. This is obvious for resources in queuing networks, but is also easily observable for objects. For example, a single-threaded object implementation in a typical CORBA environment enforces the FCFS discipline for incoming operation requests, and a multi-threaded implementation reflects often directly the CPU scheduling discipline of the underlying operating system.

Unfortunately, there are also valid arguments against using queuing networks for modeling CORBA based distributed systems. The following two problems are particularly challenging:

- Simultaneous resource possessions is not supported,
- Queuing networks of CORBA based systems are often too complex for software engineering purposes.

In traditional queuing networks, it is assumed that a job can only use a single resource at a time, and simultaneous resource possession is not possible. However, simultaneous resource possession occurs frequently in CORBA based distributed systems, since the basic communication primitive is a synchronous call that blocks the caller until it gets the reply. Furthermore, it is commonplace to use nested invocations, so that several objects are blocked unless multi-threading is used in the object implementations. In addition, most software and hardware resources in object-oriented systems are represented in terms of objects. If a client wishes to access such resources, it must first acquire the possession of the corresponding object and simultaneous resource possession may result.

The second problem, excessive complexity, is a consequence of the intricate structure of CORBA based distributed systems. In a reasonably accurate queuing network model, resources would exist for representing the application (e.g. CORBA object implementations), the infrastructure (e.g. ORB daemons), the hardware (e.g. CPUs and disks), and the network (e.g. network latency). Even a simple invocation at the application level would access a large number of them. To specify the queuing network would require the tedious task of analyzing the full execution path of jobs for obtaining the service demands for all system resources. If the system is modified during an iterative design process, the whole procedure must be repeated to obtain the changed service demands before the model can be solved again for performance metrics.

The first problem is addressed in Sections 4.4, 4.5, and 4.6 where we extend queuing networks to support simultaneous resource possession. The second problem is studied in Chapter 5 where we propose modeling techniques for dealing with complex CORBA based systems.

4.4 Augmented queuing networks

We now define *augmented queuing networks* (AQNs) for supporting the performance modeling of CORBA based distributed systems. When compared to product-form BCMP networks, AQNs have the two important extensions. On one hand, a job can possess more than one resource at a time and, on the other hand, each access to a resource can be specified and solved separately. The first extension is an essential requirement for CORBA based systems. The second extension is needed to produce an adequate implementation for the first one, since a more fine-grained workload specification is required for analyzing the effect of simultaneous resource possessions. It essentially means that service demands must be specified for individual resource accesses instead of giving the service demand totals on a class by class basis as in traditional queuing network models. This extension is particularly useful from the software engi-

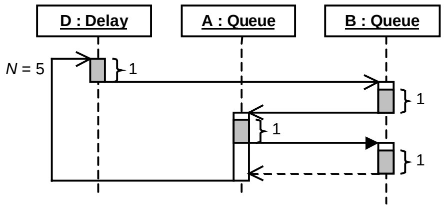


Figure 5. An AQN with three direct uses and one synchronous call.

neering point of view since it brings performance models and the obtained results closer to the functional descriptions of software behavior.

To support simultaneous resource possession in AQNs, we define two ways for jobs to access resources (see e.g. [Jac82]):

- Direct use,
- Synchronous call.

A direct use refers to the usual way of accessing resources in queuing networks, i.e. a job can access exactly one resource at a time. Direct uses are like asynchronous one-way messages to indicate the transfer of execution between resources. *A synchronous call*, on the other hand, refers to a nested flow of control. A job first gains possession of a primary resource and access it for a given service time (which may also be zero). After having accessed the primary resource, the job keeps the primary resource to itself and accesses a secondary resource. When the service time for secondary resource is over, both resources are released. Synchronous calls can be nested so that the job may proceed to another secondary resource without first releasing the already reserved secondary resource. There is no limit to the depth of nesting.

We use simple UML sequence diagrams to represent AQNs (see also [Poo99]). Figure 5 shows a model with three resources and a single class of five jobs. Resources are represented as UML objects. Delay resources are instances of the *Delay* object class[2] and queuing resources are instances of the *Queue* class. Time proceeds from top to bottom in the diagram, and a sequence of arrows indicates how the resources are used by the jobs. For closed models, we write down explicitly the class population and sometimes we also draw an arrow from the last resource access

[2] If there is a danger of confusion, we use the term *object class* to denote UML classes and the term *workload class* or *job class* to denote classes in queuing networks.

back to the top of the diagram. For open models, the arrival rate for the job class is given. An enlargement in a resource's lifeline indicates that a job is accessing the resource. A gray area in the enlargement indicates that the job is in service. A white area before the gray area indicates that the job is queuing for the resource, and a white area after the gray area indicates that the job is waiting for a reply from an embedded synchronous call. Numerical values for service demands (the gray area) and residence times (the gray and white areas together) may also be given in the diagram. To comply with the UML notation, we use an open arrowhead to indicate a direct use, a filled arrowhead to indicate a synchronous call, and a dashed arrow to indicate the return from a synchronous call. The primary resource's activation period is extended to cover the time it stays blocked during the execution of the synchronous call. The simple example in Figure 5 contains three direct uses and one synchronous call.

We allow synchronous calls to have several characteristics that increase their usability in performance modeling. First, they can be chained to arbitrary length. In such a chain, the last resource cannot be accessed until a job has acquired possession of all resources in the chain before it. Second, synchronous calls can introduce cycles in the dependency graph of the resources. For example, a model may contain two CORBA objects that are allowed to make synchronous calls to each other. Such configurations may arise in object-oriented systems through the use of callback mechanisms. Third, recursion is allowed. In other words, a job that is currently accessing a resource is allowed to make a synchronous call to the same resource during this access. Recursion is interpreted as an elongation of the original call. Indirect recursion is also supported, i.e. a job may call the resource indirectly in a nested call through some other resource. Fourth, it is possible to access the same resource both through direct use and synchronous calls. For example, an application can access the same CORBA object with synchronous invocations and with asynchronous messages that allow the caller to proceed execution while the object implementation is executing the operation.

Before proceeding to the algorithm for solving AQNs, we briefly discuss two simplifying transformations. They both preserve the performance characteristics of the models while making them easier to handle:

- If there is a *nested direct use* of a resource *i* during a synchronous call, this direct use can be replaced by an additional synchronous call to *i* after the first one.

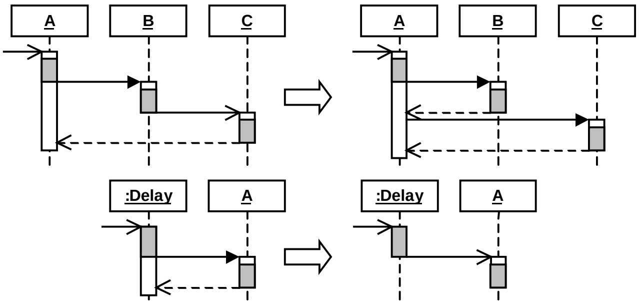


Figure 6. Simplifying transformations for AQNs.

- If a delay resource is making a synchronous call, this call can be changed to a direct use of the called resource.

These transformations are illustrated in Figure 6. The first one results from the definitions for direct uses and synchronous calls. The second one reflects the fact that delay resources never impose queuing for jobs. Hence, we may later assume that there are no direct uses within synchronous calls and that delay resources never make synchronous calls. However, we still allow such constructs to appear in UML sequence diagrams for added convenience to the designer.

4.5 Introduction to the method of decomposition

We now give an informal introduction to the method of decomposition (MOD) that can be used for solving AQNs. Decomposition is a known technique for solving queuing networks (see e.g. [Agr85, Men94, Con89]), and our approach combines existing techniques in a novel way to satisfy the requirements of our framework. Full details of the algorithm are presented in Section 4.6.

Decomposing an AQN

In general, an AQN cannot be represented as a product-form queuing network. The source of the problem is the congestion in the secondary resource during a synchronous call, since it affects the queue lengths of both the blocked resource and the secondary resource itself. Consequently, the states of these two resources cannot be considered separately. To solve the problem, we decompose the AQN into multiple queuing networks so that the primary and secondary resources of a synchronous call are always in different networks. As a result, we get a set of product-form multi-class queuing networks that can be solved efficiently. This approach is related to the technique proposed in [Jac82]. Their approach is suited for simple cases where the result can be represented with two product-form queuing networks, while our approach can be applied to complex AQNs requiring a large number of auxiliary product-form networks for sufficiently accurate modeling.

We use the term *secondary resource* to denote any resource that receives at least one synchronous call in the AQN. All other resources are called *primary resources*. We now transform the AQN into a *primary network* by removing all secondary resources and all accesses to them, and by adding a *surrogate delay resource, Surr*. The open and closed classes of the original AQN are preserved in the primary network.

To model the effect of accessing secondary resources without actually having them in the primary network, we make the following two changes to the service demands in the primary network:

- For each direct use a of a primary resource, the service demand D_a is increased by the sum of the residence times obtained for synchronous calls that are executed during the direct use a.
- For each class r, the service demand $D_{Surr,r}$ of the surrogate device is defined to be the sum of the residence times for all direct uses of secondary resources in class r.

The first transformation elongates the primary resource's service demand by the time it takes to complete all nested synchronous calls to secondary resources. The second transformation models direct uses of secondary resources with an equivalent delay in the primary network. As a result, all accesses to secondary resources are reflected in the service demands for the primary network.

Unfortunately, the primary network cannot be solved unless we know the residence times for accessing the secondary resources. Therefore, we create a set of *secondary networks*. For each secondary resource i, we create a secondary network $Sec(i)$ that has two resources: the secondary resource i itself and an *auxiliary delay resource, Aux,* for modeling the time the jobs are spending elsewhere in the system.

To model contention for secondary resources, classes of jobs are specified for the secondary networks. Each class r in the original AQN generates at most two secondary classes for a secondary network *Sec*(i). The following two rules apply:

- If there are synchronous calls from class r to the secondary resource i, a closed secondary class s is created, and its population N_s is the maximum number of synchronous calls that can reach the secondary resource in parallel (i.e. without passing through common queuing resources or synchronous calls). For example, if there is a single synchronous call from class r to resource i, then $N_s = 1$.
- If class r has direct uses of the secondary resource i, a secondary class s is created. If class r is open, so is s and the arrival rates are equal. If class r is closed, so is s and the populations are equal.

The first rule reflects the basic characteristic of a synchronous call that allows at most one client to queue for the secondary resource. Other clients are waiting in the primary resource's queue and do not appear in the secondary network unless they can reach the resource through some other chain of access. The second rule treats direct uses of secondary resources as if the resources were part of the primary network and the whole population was allowed to queue for them.

We now describe the workloads for the secondary classes. All direct uses and synchronous calls to a secondary resource i in the original AQN are transformed into direct uses of i in the network *Sec*(i), and the service demands are kept the same. However, if the original access contains nested synchronous calls to some other secondary resources, the service demand of the original accesses is increased by the sum of the residence times obtained for these nested synchronous calls. As a result, there are also dependencies between secondary classes.

To ensure that closed secondary classes correctly represent the contention in the secondary resources, the service demands of the auxiliary delay devices must be adjusted appropriately. The idea is to keep the throughputs of secondary classes equal to the throughputs of the calling classes in the primary queuing network. This way, primary and secondary networks can be considered to represent different parts of the same overall system. If we know the residence time R_i of the secondary resource i, the population N_s of the secondary class s, the throughput X_r of the corresponding primary class r, we can apply Little's law to solve the service demand for the auxiliary delay resource *Aux*:

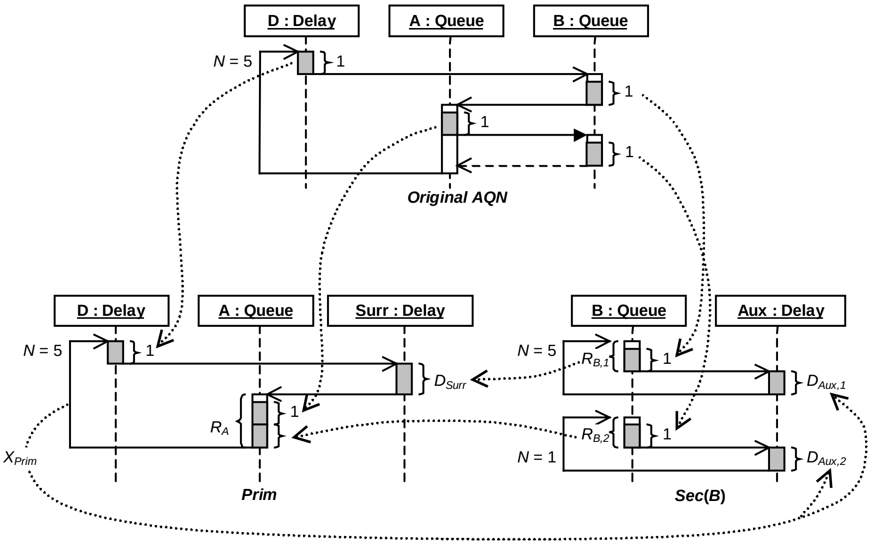


Figure 7. Transformation of the AQN in Figure 5 into primary and secondary networks.

$$D_{Aux} = R_{Aux} = \frac{N_s}{X_r} - R_i. \tag{5}$$

There is no need to specify the service demand for the auxiliary resource in open secondary classes, since the amount of contention for the secondary resource is determined by the arrival rate.

Consider the example in Figure 5. The only secondary resource is *B* and, hence, the original AQN is decomposed into two networks, *Prim* and *Sec*(*B*). The secondary network *Sec*(*B*) has two classes, the first one represents direct uses of *B* and the second one models synchronous calls to *B*. This is illustrated in Figure 7. The dotted arrows descending from the original AQN show how the original service demands are moved to the primary and secondary networks. The horizontal dotted arrows indicate the two types of dependencies that exist between the networks. On one hand, the service demands of the primary network depend on the residence times $R_{B,1}$ and $R_{B,2}$ obtained from the secondary network. On the other hand, the service demands $D_{Aux,1}$ and $D_{Aux,2}$ of the auxiliary resource in the secondary network are obtained from the throughput X_{Prim} of the primary network by using equation (5).

Solving primary and secondary networks simultaneously

After decomposition, the resulting primary and secondary networks cannot be solved independently since input parameters for some networks require the existence of a solution for some other networks, and circles can exist in the dependence diagram. Therefore, we solve the queuing networks simultaneously with iteration. This approach is a special case of the methodology proposed by [Agr85]. During each repetition, we adjust the input parameters of the networks with the outcome of the previous repetition until the throughputs of secondary classes are sufficiently close to the throughputs of the corresponding primary classes. As a result, we get a solution that gives performance metrics for all networks and respects the mutual dependencies that exist between primary and secondary networks. The main steps of the algorithm are:

1. Decompose the AQN into primary and secondary networks.
2. Initialize the input parameters of the primary and secondary networks by assuming no contention for secondary resources.
3. Solve the primary and secondary queuing networks. Obtain average residence times, throughputs, and average queue lengths for all resources in all networks.
4. Distribute the residence times obtained for resources to the individual accesses that contributed to the service demands of those resources.
5. Compute the maximum relative error for secondary network throughputs when compared to the throughputs of the corresponding primary classes. If the error is less than a given constant ε, terminate algorithm.
6. Compute service demands for resources in primary and secondary classes. Determine the service demand $D_{i,r}$ for a resource i in a primary or secondary class r by adding together the service demands of all accesses from class r to resource i.
7. Compute service demands for surrogate resources in primary classes. Determine the service demand $D_{Surr,r}$ by adding together the service demands of all direct uses of secondary resources in class r.
8. Adjust the service demand of auxiliary delay devices in secondary classes so that all closed secondary classes have the same throughput with their corresponding primary class.
9. Jump to step 3.

Full details for each step are given in Algorithm 5 in Section 4.6. We now illustrate the steps by solving the example model in Figure 7. We use the notations of Figure 7 and assume that $\varepsilon = 0.0001$. The initialization step 2 produces the following service demand approximations:

$$D_A = 2,\ D_{Surr} = 1,\ D_{Aux,1} = 0,\ D_{Aux,2} = 0.$$

After solving the primary and secondary networks, the following results are obtained:

$$R_A = 8.03077,\ R_D = 1,\ X_{Prim} = 0.498466,$$
$$R_{B,1} = 6,\ X_{Sec(B),1} = 0.833333,$$
$$R_{B,2} = 6,\ X_{Sec(B),2} = 0.166667.$$

Step 4 is trivial since all obtained residence times are associated with exactly one resource. The maximum relative error

$$Error = \max\left[\left|\frac{0.833333 - 0.498466}{0.498466}\right|, \left|\frac{0.166667 - 0.498466}{0.498466}\right|\right] = 0.671795$$

is greater than ε so we continue the algorithm. In steps 6, 7, and 8, the obtained metrics are used to compute new values for the missing parameters:

$$D_A = 7,\ D_{Surr} = 6,\ D_{Aux,1} = 4.03077,\ D_{Aux,2} = 0.$$

Formula (5) produces a negative result for $D_{Aux,2}$ due to the substantial difference that exists between the throughput estimates for the primary and secondary networks, so we use zero instead. Similar excessive corrections occur during the first iterations of the algorithm but they disappear when the estimates get more accurate. When the adjusted network is solved again in step 3, a new set of results is obtained:

$$R_A = 28.1077,\ R_D = 1,\ X_{Prim} = 0.142419,$$
$$R_{B,1} = 3.77396,\ X_{Aux,1} = 0.640637,$$
$$R_{B,2} = 2.7827,\ X_{Aux,2} = 0.359363.$$

The process continues until the maximum relative error for throughputs $X_{Aux,1}$ and $X_{Aux,2}$ less than ε. In this case, 18 iterations are needed. The final set of results is the following:

$$R_A = 9.80524,\ R_D = 1,\ X_{Prim} = 0.384312,$$
$$R_{B,1} = 2.20551,\ X_{Aux,1} = 0.384311,$$
$$R_{B,2} = 1.58382,\ X_{Aux,2} = 0.384293.$$

Hence, the average response time for a job is $R_D + R_{B,1} + R_A = 13.0108$.

Dividing residence times between multiple calls

When there are multiple accesses $a_1, \ldots, a_n$ from class r jobs to a resource i, the service demands $D_1, \ldots, D_n$ of these accesses are added together in step 6 to produce the total service demand $D_{i,r}$. This is a well-known technique for obtaining queuing network parameters from software specifications [Smi90]. In our case, however, the obtained total residence time $R_{i,r}$ must also be divided back to the contributing accesses in step 4. This information is used in step 6 when new service demand estimates are calculated for those accesses that have nested synchronous calls.

Since all contributing accesses are queuing for the same resource, it is reasonable to assume that the average waiting time is approximately the same for all accesses of resource i from class r (see [Hav98] for a discussion on this approximation). We use the notation $W_{i,r}$ to represent this average waiting time. The average residence time for access a_k is now given by

$$R_k = W_{i,r} + D_k$$

for all $k = 1, \ldots, n$. When we sum up these equations and solve $W_{i,r}$ we obtain

$$W_{i,r} = \frac{1}{n}\left[\sum_{k=1}^{n} R_k - \sum_{k=1}^{n} D_k\right] = \frac{1}{n}\left(R_{i,r} - D_{i,r}\right)$$

and therefore

$$R_k = \frac{R_{i,r} - D_{i,r}}{n} + D_k \tag{6}$$

for all $k = 1, \ldots, n$. This formula is used in step 4 of the algorithm to divide the total residence time to the contributing accesses. If i is a delay resource, we have $R_{i,r} = D_{i,r}$ and the formula can be reduced to the exepected form: $R_k = D_k$ for all $k = 1, \ldots, n$.

Recursion

Recursion refers to a synchronous call to a resource when the same resource is already being accessed by the same job. Recursion is commonly used in object-oriented systems as it allows objects to use the same services that they are offering for their clients. Since our target is in object-oriented systems, the performance aspects of recursive calls need to be studied in more detail.

Recursion can be interpreted as an elongation of the ongoing access during which the recursive call takes place. Since the target resource is already being accessed by the job, we may assume that there is no additional waiting time before the recursive call starts receiving service. This assumption holds for many object-oriented systems, since invocations to the object itself can be routed directly to the target method to increase efficiency. As a result, the normal invocation scheduling is bypassed and no waiting occurs. This observation implies a change to the results so far since equation (6) no longer holds for recursive accesses. Instead, the residence time for a recursive access is always equal to the service demand. Waiting can only occur before the non-recursive access that precedes the recursion. Hence, the divider *n* in equation (6) should be the number of non-recursive accesses from class *r* to resource *i*.

Recursion has also an effect on the total service demand that is normally obtained by summing up individual service demands in steps 6 and 7 of the algorithm. Let a_0 be a non-recursive access from class *r* to a queuing resource *i* representing an object implementation, and let D_0 be the service demand for a_0. Let $a_1, \ldots, a_n$ be recursive calls that take place during a_0 with service demands $D_1, \ldots, D_\mathrm{n}$. Denote the combined access with symbol a_{comb} and the combined service demand with D_{comb}. We may assume FCFS scheduling for resource *i*, since it is the most straightforward way to implement objects that impose queuing for incoming requests. In addition, we follow the BCMP requirements and assume that the distributions for the service demands $D_1, \ldots, D_\mathrm{n}$ are negative exponential. As a result, the combined service demand

$$D_{comb} = \sum_{k=0}^{n} D_k$$

is hypo-exponentially distributed and we have a disagreement with the BCMP requirement that only allow negative exponential distributions with FCFS scheduling. Intuitively speaking, the variability of the combined access a_{comb} is smaller compared to an equally long access that is not composed of multiple parts. The decreased variability reduces contention for the resource and, consequently, the resulting average residence times are shorter. If we solve the primary and secondary queuing networks without any corrections, the obtained residence time estimates would be too high.

To take into account the reduced contention due to recursive calls, we replace the combined service demand D_{comb} with a slightly smaller value. Essentially, we are trying to find an exponentially distributed service demand with average D'_{comb} that produces the same residence time R_{comb} as the combined hypo-exponentially distributed service demand. We make an assumption that the arrival process for class r jobs in resource i is Poisson with arrival rate $\lambda_{i,r}$. The first and second moments for the hypo-exponentially distributed service demand are

$$\alpha_1 = \sum_{k=0}^{n} D_k \qquad \text{and} \qquad \alpha_2 = \alpha_1^2 + \sum_{k=0}^{n} D_k^2 .$$

We can now apply the Pollaczek-Khinchin formula for M|G|1 queues to solve the residence time:

$$R_{comb} = \alpha_1 + \frac{\lambda_{i,r}\alpha_2}{2(1-\lambda_{i,r}\alpha_1)}. \tag{7}$$

On the other hand, the standard result for the residence time of M|M|1 queues implies that

$$R_{comb} = \frac{D'_{comb}}{1-\lambda_{i,r}D'_{comb}}. \tag{8}$$

Hence, we can solve the adjusted service demand:

$$D'_{comb} = \frac{R_{comb}}{1+\lambda_{i,r}R_{comb}}.$$

If there is no recursion (i.e. $D_k = 0$ for all $k = 1, \dots, n$) the above equations can be reduced to the expected result: $D'_{comb} = D_{comb}$.

The above technique for adjusting the service demand of recursive calls makes it possible to use the efficient and robust algorithms presented in Section 4.2. Alternatively, we could have selected an algorithm that supports general service time distributions at FCFS resources (e.g. [Cha75, Shu77]). However, these algorithms are also approximations and they lead to solutions that are computationally more expensive and less robust. See [Agr85] for a discussion. The assumption that the arrival process is Poisson is essential for obtaining equations (7) and (8). While this assumption is not valid for BCMP networks in general, Shum and Buzen also used a similar assumption in their algorithm that supports general service time distributions at FCFS resources [Shu77]. The final motivation for the proposed technique arises from experience. We have observed significant increases in accuracy for open classes. See Section 4.7 for an example.

Convergence

The algorithm in its basic form does not converge in all situations. In particular, if there are several chains of nested synchronous calls that access the same set of resources in different orders, the number of dependencies between the primary and secondary increases and the proposed formulas for adjusting service demands may yield excessive corrections. As a result, the algorithm starts oscillating around a solution but does not converge towards it. It should be noted, however, that this is not a common phenomenon. According to our experience, convergence problems occur typically when the implementation of the model is prone to have deadlocks. AQNs that represent realistic systems typically have no convergence problems.

To improve convergence in problematic situations, we do not use the computed service demands directly. Instead, we use a weighted average of the computed value and the value obtained from the previous iteration. Let ω be an attenuation factor that denotes the weight of the previous value. The computed service demands are adjusted using the following formula:

$$D_{adjustd} = (1-\omega)D_{computed} + \omega D_{previous},$$

where $D_{adjusted}$ denotes the value that is used in the algorithm, $D_{computed}$ denotes the value that is obtained directly from computation, and $D_{previous}$ indicates the value of $D_{adjusted}$ during the previous iteration.

We have observed that the value $\omega = 0.3$ is sufficient for ensuring convergence in most problematic models. To be on the safe side, though, our prototype tool defaults to $\omega = 0.5$. The drawback of using this technique is slower convergence for those models that have no convergence problems. However, our experimentation shows that the price is not too high. For example, the example in Figure 5 requires 18 iterations without adjustment, and 23 iterations with the attenuation factor $\omega = 0.5$. For simple models that would normally converge in one or two iterations, the value $\omega = 0.5$ usually produces around 15 iterations.

High utilization in open classes

An additional convergence problem may occur when the utilization of a queuing resource is close to 100% in an open class. Depending on the structure of the model, the algorithm may produce too large estimates for the service demands, and the intermediate utilization estimate of the resource may incorrectly become greater than 1. The result is an unstable network and the algorithm terminates prematurely.

A solution to the above problem can be found by examining the algorithm in detail. The idea of the algorithm is to propagate information between primary and secondary networks in two different ways. On one hand, the residence times of secondary resources are propagated upwards in the hierarchy of synchronous calls until they reach the primary network. On the other hand, the secondary networks are synchronized with the primary network by using the throughputs of the primary classes for adjusting the service demands of auxiliary resources.

Suppose now that we have a model with unstable primary or secondary networks. From the rules for decomposing the AQN, we know that an open class is either a primary class or a secondary class that models direct uses of secondary resources. This implies that open classes can only exist at the top two levels in the hierarchy of synchronous calls. Therefore, the algorithm can propagate residence times of secondary resources successfully upwards except for the last step. In addition, we know that the throughput of a stable open class is equal to its arrival rate. Consequently, we can synchronize secondary networks with the assumed throughputs of open primary classes even if the open primary network cannot be solved. If the primary network has closed classes, we can use the last valid throughput estimate as an approximation and synchronize secondary classes with that.

The above observations lead to some trivial changes for the algorithm. First, if an unstable primary or secondary network is observed during iteration, the algorithm continues as usual except that the unstable networks (and all other networks that depend on them) are left unsolved. Second, the service demands of auxiliary resources are computed using the last *valid throughput estimate* of the corresponding closed primary class or the *arrival rate* of the corresponding open primary class. If an

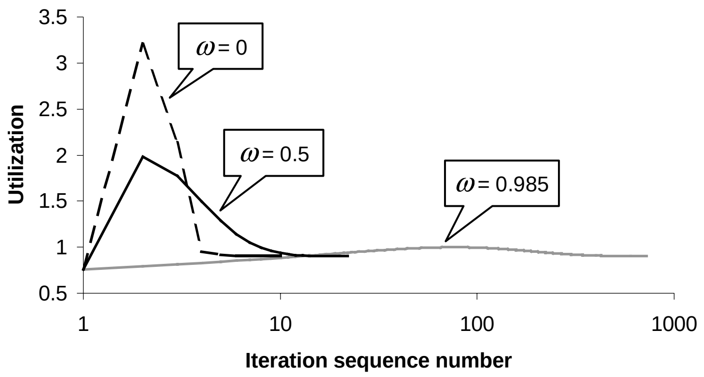


Figure 8. Convergence in a mixed AQN with high resource utilization.

unstable network becomes stable after one or more iterations, we revert to the original algorithm. Otherwise the algorithm continues with the unstable networks as indicated. Experience has shown that the revised algorithm converges even when the utilization of some queuing resources remains greater than 100%. Therefore, the termination condition can be the same for the revised algorithm, but a post-termination check must be added to ensure that the utilization of all queuing resources is less than 1.

Figure 8 illustrates convergence in a mixed AQN with high resource utilization (90%). The graph shows the utilizations of a queuing resource during iteration for three different values of the attenuation factor ω. Already the second iteration produces incorrect utilizations for $\omega = 0$ and $\omega = 0.5$. However, the revised algorithm quickly recovers from this irregular situation and converges in 10 iterations for $\omega = 0$ and in 22 iterations for $\omega = 0.5$. It is possible to verify the obtained results by increasing the attenuation factor high enough so that the utilization remains less than 1 during the complete algorithm. In this case, the value $\omega = 0.985$ is sufficiently high to have this effect. The algorithm converges to the same solution but now it requires 726 iterations. Notice the logarithmic *x*-axis for the iteration sequence numbers.

4.6 The method of decomposition

The various techniques presented in Section 4.5 are now combined into a complete algorithm.

Algorithm 5. The method of decomposition for solving AQNs. The input for the algorithm is an augmented queuing network *M* defined by the following parameters:

K	Set of resources,
R	Set of classes (open or closed),
N_r	Population for each closed class *r*,
λ_r	Arrival rate for each open class *r*,
A	Set of accesses to resources,
Children(*a*)	Set of first-level synchronous calls during access *a*,
D_a	Service demand for an access *a*,
ε	Error tolerance,
ω	Weight of previous result during iteration.

An access $a \in A$ is characterized by the following properties: it is associated to a class $r \in R$, it has a target resource $i \in K$, it may be recursive or non-recursive, and it may be a direct use or a synchronous call. For an access *a*, define the set *Subtree*(*a*) to contain all synchronous calls that are executed during *a* either directly or indirectly. Hence, *Subtree*(*a*) contains all calls in *Children*(*a*) and those calls that are made during these calls, and so on.

Step 1. Define the set of secondary resources to contain all targets of synchronous calls. Transform *M* into primary queuing network *Prim* by removing all secondary resources. The original set of classes *R* remains the same for *Prim* except that accesses to secondary resources are removed. In addition, add a surrogate delay resource *Surr*.

For each secondary resource *i*, create a secondary queuing network *Sec*(*i*). The network has two resources: *i* itself and an auxiliary delay resource *Aux*. Two kinds of classes are created for secondary networks:

- For each class *r* in the original model *M* making synchronous calls to resource *i*, create a closed secondary class $s \in$ *Sec*(*i*). The population of *s* is determined by the number of separate invocation sequences that can reach resource *i* without passing through a common queuing resource or a common synchronous call. If class *r* is closed, the population of class *s* limited by that of class *r*: $N_s \leq N_r$.
- For each class *r* in the original model *M* making direct use of resource *i*, create a secondary class $s \in$ *Sec*(*i*). If class *r* is open with

arrival rate λ_r, then class s is also open and $\lambda_s = \lambda_r$. If class r is closed with population N_r, then class s is also closed and $N_s = N_r$.

For a secondary class s, the notation *Root*(s) indicates the primary class r that induced the secondary class through the above rules. For a primary class r, we define *Root*(r) = r. For each secondary class s, the symbol *Dev*(s) indicates the unique secondary device in the corresponding secondary network. Define the following two sets: *Access*(i, r) contains all accesses from class r to resource i excluding direct uses of secondary resources, and *Direct*(r) contains all direct uses of secondary resources in a primary class r. In addition, define the following subsets: *AccessR*(i, r) contains all recursive accesses in *Access*(i, r), *AccessN*(i, r) contains all non-recursive accesses in *Access*(i, r), *DirectR*(r) contains all recursive uses in *Direct*(r), and *DirectN*(r) contains all non-recursive uses in *Direct*(r).

Step 2. Initialize the service demands in primary and secondary networks. For each resource i and class r,

$$D_{i,r} = \sum_{a \in Access(i,r)} \left[D_a + \sum_{b \in Subtree(a)} D_b \right].$$

Initialize the service demand of the surrogate delay resource in each primary class r:

$$D_{Surr,r} = \sum_{a \in Direct(r)} \left[D_a + \sum_{b \in Subtree(a)} D_b \right].$$

Initialize the service demand of the auxiliary delay resource in each secondary class r:

$$D_{Aux,r} = 0.$$

Step 3. Solve the primary and secondary networks with algorithm 4 to obtain the following metrics for each resource i and class r: average residence time $R_{i,r}$, average queue length $n_{i,r}$, and throughput X_r. If one or more networks contain unstable open classes or undefined service demands, leave these networks unsolved and skip also step 4 for them. In this case, ensure that all primary classes r have a throughput associated with them. If r is open, use the arrival rate λ_r, and if r is closed, use the last valid throughput that was obtained during a previous iteration.

Step 4. For each resource *i* and class *r*, use the obtained residence time $R_{i,r}$ to compute the residence times for individual accesses $a \in Access(i, r)$:

$$R_a = \begin{cases} \dfrac{R_{i,r} - D_{i,r}}{|AccessN(i,r)|} + D_a & \text{for all } a \in AccessN(i,r) \\ D_a & \text{for all } a \in AccessR(i,r). \end{cases}$$

For each primary class *r*, use the obtained residence time $R_{Surr,r}$ of the surrogate resource to compute the residence times for direct uses of secondary resources $a \in Direct(r)$:

$$R_a = \begin{cases} \dfrac{R_{Surr,r} - D_{Surr,r}}{|DirectN(r)|} + D_a & \text{for all } a \in DirectN(r) \\ D_a & \text{for all } a \in DirectR(r). \end{cases}$$

Step 5. Compute the maximum relative error in the throughputs of all closed secondary classes *r*:

$$Error = \max_r \left| \frac{X_r - X_{Root(r)}}{X_{Root(r)}} \right|.$$

If $Error \le \varepsilon$, terminate the algorithm, otherwise continue from step 6. After termination, check the stability of the model by computing the utilization of all queuing resources *i*:

$$U_i = \begin{cases} \sum_{r \in R} X_r D_{i,r} & \text{if } i \text{ is a primary resource} \\ \sum_{r \in Sec(i)} X_r D_{i,r} & \text{if } i \text{ is a secondary resource.} \end{cases}$$

If $U_i < 1$ for all queuing resources *i* the model is stable, otherwise not.

Step 6. Compute service demands for resources in primary and secondary networks. For all resources *i* and classes *r* such that there are no recursive calls from *r* to *i*,

$$D_{i,r} = (1-\omega) \sum_{a \in Access(i,r)} \left[D_a + \sum_{b \in Children(a)} R_b \right] + \omega D_{i,r}^{prev}.$$

The symbol $D_{i,r}^{prev}$ indicates the value of $D_{i,r}$ during the previous iteration. If some of the terms on the right-hand side are not available due to unsolved networks, do not compute the result. If there are nested recursive calls (i.e. *AccessR*(*r*, *i*) ≠ ∅), compute the first and second moment for the service demand of the combined call:

$$\begin{aligned}\alpha_1 &= \sum_{a\in AccessR(i,r)}\left(D_a+\sum_{b\in Children(a)}R_b\right)+\sum_{a\in AccessN(i,r)}\left(D_a+\sum_{b\in Children(a)}R_b\right)\\ \alpha_2 &= \alpha_1^2+\sum_{a\in AccessR(i,r)}\left(D_a+\sum_{b\in Children(a)}R_b\right)^2+\left[\sum_{a\in AccessN(i,r)}\left(D_a+\sum_{b\in Children(a)}R_b\right)\right]^2\end{aligned} \quad (9)$$

Compute the average residence time for the combined call:

$$R=\alpha_1+\frac{X_{Root(r)}\alpha_2}{2(1-X_{Root(r)}\alpha_1)}. \quad (10)$$

The service demand estimate is now given by

$$D_{i,r}=(1-\omega)\frac{R}{1+X_{Root(r)}R}+\omega D_{i,r}^{prev}. \quad (11)$$

Step 7. For each primary class *r* that has no recursive direct uses, compute the service demand for the surrogate resource

$$D_{Surr,r}=(1-\omega)\sum_{a\in Direct(r)}\left[D_a+\sum_{b\in Children(a)}R_b\right]+\omega D_{Surr,r}^{prev}.$$

The symbol $D_{Surr,r}^{prev}$ indicates the value of $D_{Surr,r}$ during the previous iteration. If some of the terms on the right-hand side are not available due to unsolved networks, do not compute the result. If there are recursive direct uses (i.e. *DirectR*(*r*) ≠ ∅), a smaller service demand estimate is used. This service demand is obtained as in formulas (9), (10), and (11) except that *AccessN*(*i*, *r*) is replaced by *DirectN*(*r*), *AccessR*(*i*, *r*) is replaced by *DirectR*(*r*), and $D_{i,r}$ is replaced by $D_{Surr,r}$.

Step 8. For each secondary class *r*, compute the service demand for the auxiliary resource

$$D_{Aux,r}=(1-\omega)\left[\frac{N_r}{X_{Root(r)}}-R_{Dev(r),r}\right]+\omega D_{Aux,r}^{prev}.$$

The symbol $D_{Aux,r}^{prev}$ indicates the value of $D_{Aux,r}$ during the previous iteration. If $D_{Aux,r}$ gets a negative value as a result of this step, set $D_{Aux,r} = 0$.

Step 9. Jump to step 3.

□

4.7 Examples

We now illustrate the method of decomposition with five examples. In addition, we examine the accuracy of the algorithm by presenting simulated results for the proposed models. These examples represent commonly occurring design alternatives in distributed CORBA based systems. However, they do not cover all possible approaches that can be dealt with the method of decomposition. It is a topic for further study to produce an exhaustive analysis of the results in all relevant cases. See [Els98] for an example of an automated approach that might be used as a starting point for such a study.

The accuracy of our results is given as a percentage deviation from simulated values. Simulation results were obtained using the *method of batch means*, where the initial transient interval of a long simulation run is removed and the run is divided into several batches [Jai91]. All simulated values have a 95% confidence interval within ± 5% of the reported result.

The first example is a closed AQN with three queuing servers and two classes of jobs. In the first class, all three servers are accessed in a sequence of nested synchronous calls. In the second class, only the second and third servers are accessed. Once the jobs have completed the calls,

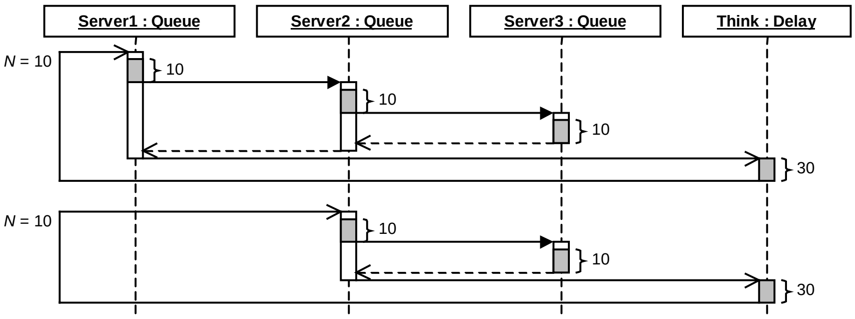


Figure 9. A closed AQN with two classes of jobs.

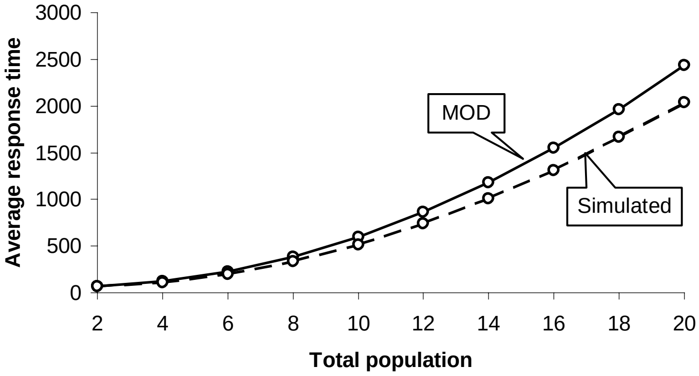


Figure 10. Average response times for the jobs in the first class of the model in Figure 9.

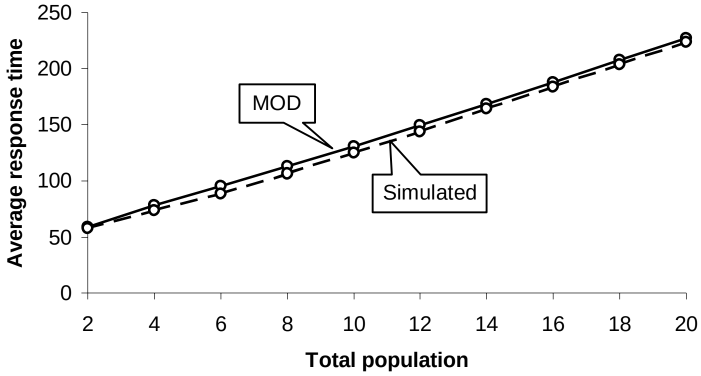


Figure 11. Average response times for the jobs in the second class of the model in Figure 9.

they pause for 30 time units and start over. An additional delay resource represents the pause.

The model is illustrated in Figure 9. Figure 10 shows the average response times for the jobs in the first class, and Figure 11 shows the average response times for the jobs in the second class. In these figures, the total population varies between 2 and 20, and it is divided evenly between the two classes. We define the response time to be the full cycle time including both the accesses to the model's resources and the think

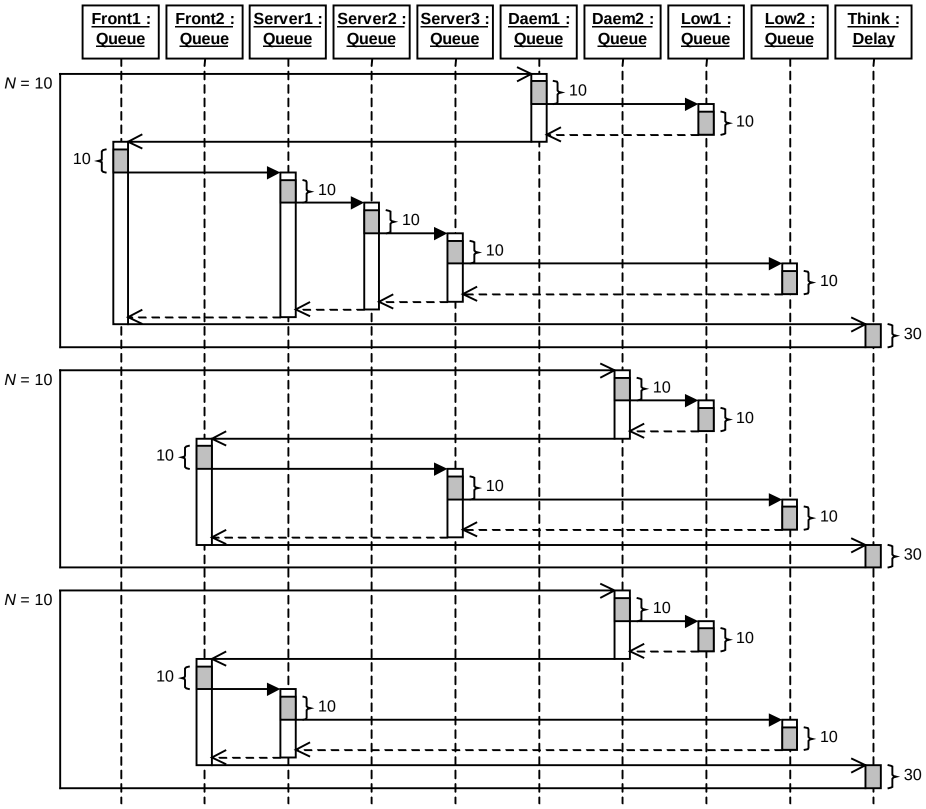


Figure 12. A closed AQN with three classes of jobs.

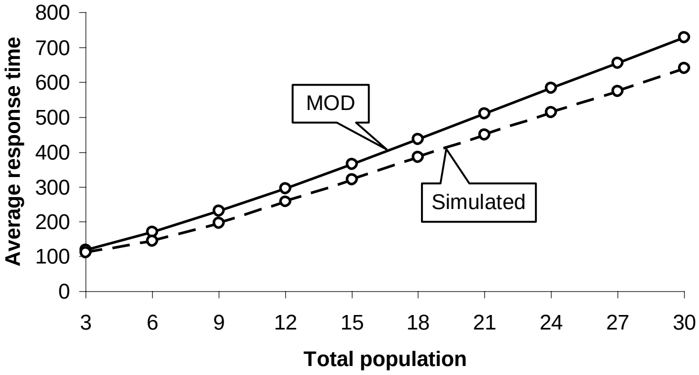


Figure 13. Average response times for the jobs in the first class of the model in Figure 12.

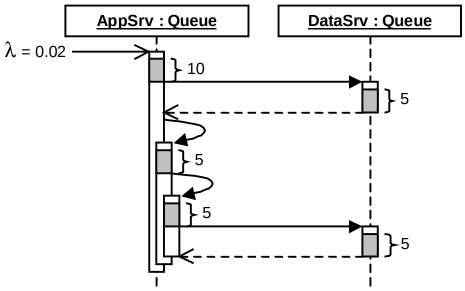


Figure 14. An open AQN with recursion

time represented by an additional access to a delay resource. For the jobs in the first class, the greatest difference between the computed response time and the simulated value is 19.3%. For the second class, the maximum error is 7.7%.

The second example illustrates a more complex system with two daemon processes, five server processes and two low-level services. In all three classes of the system, jobs first access either one of the daemons and then proceed to the server processes where sequences of nested synchronous calls take place. Jobs in different classes use different calling paths for accessing the same server processes and low-level services. The model is illustrated in Figure 12, and the average response times for the jobs in the first (upper) class are illustrated in Figure 13. Again, the total population is divided evenly between the three workloads. The greatest difference between the computed response time and the simulated value is 17.8% in Figure 13. For the other two classes with the same total populations, the maximum differences are 8.1% and 15.3%.

The third example is a simple open AQN with recursion. The model has two resources: an application server *AppSrv* and a data server *DataSrv*. Jobs arrive at the application server and make synchronous calls to the data server. After returning to the application server, the jobs make two recursive calls to the application server's own services and call the data server once more. The model is illustrated in Figure 14.

Figure 15 shows average response times for different arrival rates. It also shows response time estimates without the service demand adjustment for recursive calls (see Section 4.5 for more details). When the system is far from saturation, the relative error from the simulated value is under 10%, but when the system approaches saturation the relative error also increases (e.g. the error is 14% when $\lambda = 0.03$ and the utilization

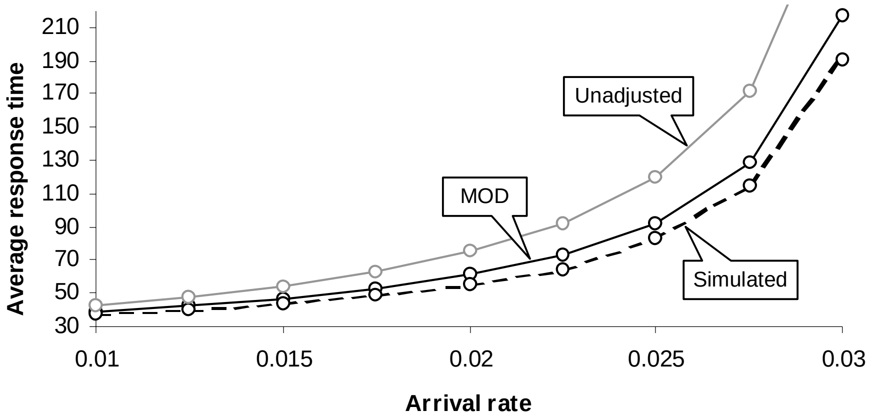


Figure 15. Average response times for the model in Figure 14.

of the application server is 86%). It is typical for open classes that the relative error increases with high utilization and, consequently, the estimates produced by the MOD algorithm are less reliable when the system is close to saturation. This is not surprising as open queuing networks are known to be less robust than closed models when the resource utilization is high [Agr85].

The fourth example illustrates recursion in a closed AQN with two classes of jobs and three resources. In the first class, jobs start by accessing the *AppSrv* and *DataSrv* resources through synchronous calls from *FrontSrv*. Then, the jobs invoke the *AppSrv* resource in a sequence of recursive accesses in a way that is similar to the previous example. Jobs in the second class increase the load of the system by accessing all three resources without recursion. The model is illustrated in Figure 16.

Figure 17 shows the average response times for the jobs in the first class with the total population ranging from 2 to 20. The population is divided evenly between the two classes. The figure also shows response time estimates without the service demand adjustment for recursive calls. The relative error for the estimated value is 25% when the total population is two but the error is significangly less for higher populations (e.g. 2% for a population of 20 jobs). It should be noted that the effect of the service demand adjustment is much smaller compared to the previous open AQN example. This is a common phenomenon for closed AQNs with recursion.

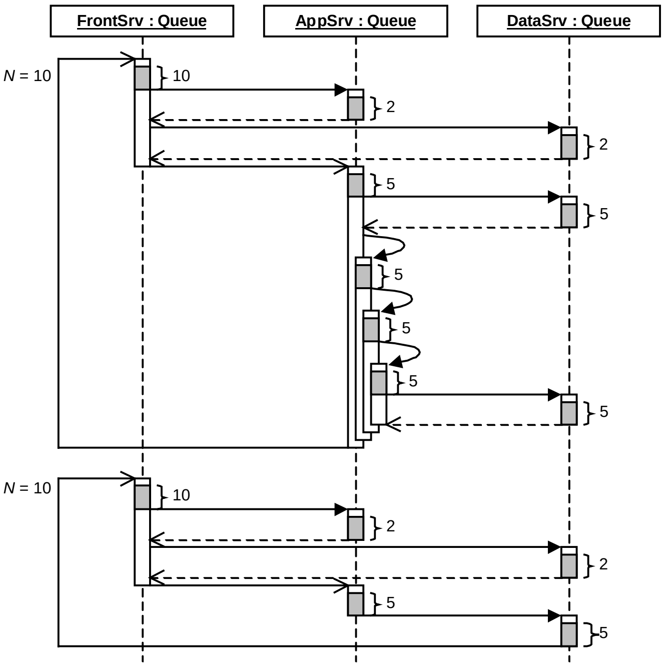


Figure 16. A closed AQN with recursion.

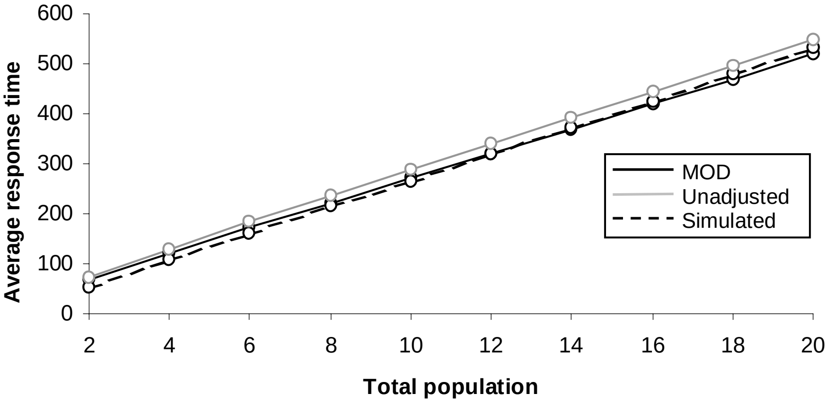


Figure 17. Average response times for the jobs in the first class in Figure 16.

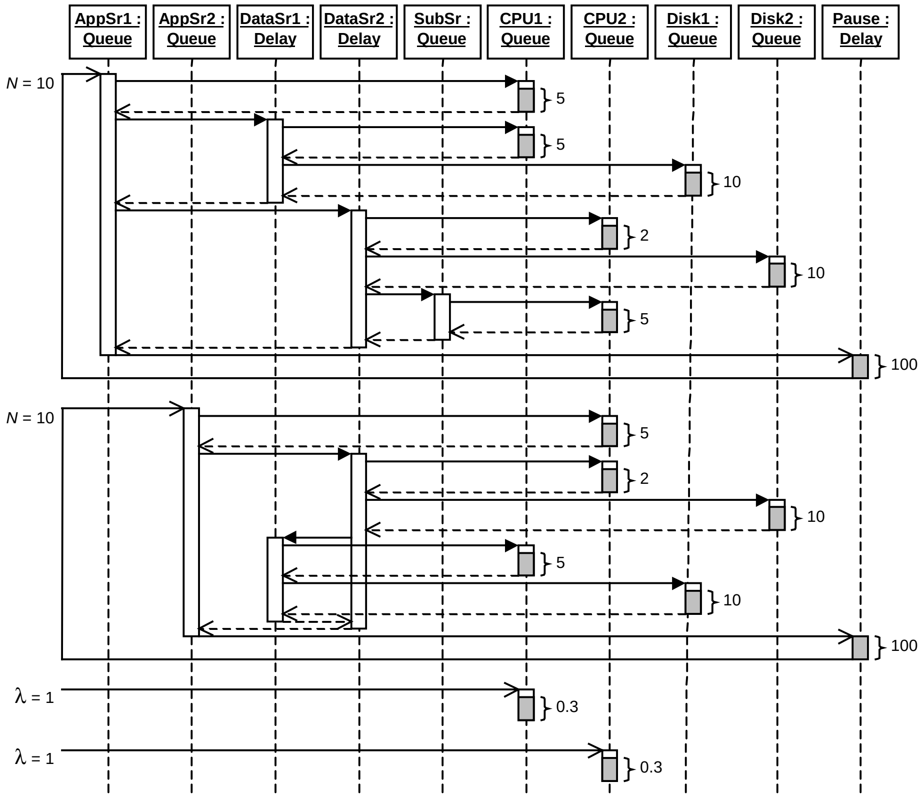


Figure 18. A mixed AQN with software and hardware resources.

The fifth example is a mixed AQN that contains both software and hardware resources. The *AppSr1*, *AppSr2*, *SubSr*, *DataSr1*, and *DataSr2* resources represent application and database processes. The *CPU1*, *CPU2*, *Disk1*, and *Disk2* resources represent the hardware. All accesses to software resources are mapped directly to hardware resources through synchronous calls (i.e. the time to access the primary resource is zero). This way, jobs must first gain access to a software resource before they can start using the necessary CPU and disk resources to accomplish their task. The model is illustrated in Figure 18.

In the presented model, *DataSr1* and *DataSr2* represent multi-threaded software servers that spawn a new thread for every incoming request. As a result, they do not impose queuing for their clients and the use of a delay resource is appropriate. Multi-threaded software servers utilize hardware resources just like single-threaded servers, and this us-

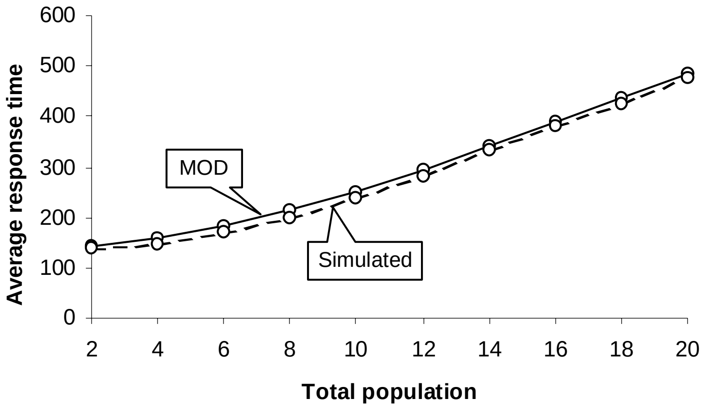


Figure 19. Average response times for the jobs in the first class in Figure 18.

age is represented by synchronous calls to the appropriate disk and CPU devices. When applying the method of decomposition, these synchronous calls are actually transformed into delays for the multi-threaded resource, and the length of the delay is determined by the amount of contention for the relevant hardware devices.

The model has four classes of jobs. The first two classes are closed and they represent the workloads imposed by clients. The last two classes are open, and they model the background load of the CPUs corresponding to a utilization level of 30%. Operating systems typically provide tools for measuring CPU utilization. Figure 19 illustrates the average response times for jobs in the first class with different total populations. Again, the total population is divided evenly between the first and the second class. In Figure 19, the greatest difference between the computed and simulated response times is 7.3%.

4.8 Discussion

We briefly summarize the method of decomposition from two viewpoints. First, we discuss the approximations used in different parts of the algorithm. Second, we consider the time complexity of the algorithm.

Summary of approximations

In step 3 of the method of decomposition, mixed product-form queuing networks are solved with Algorithm 4 that uses three approximations. First, the BCMP requirements are violated by allowing different service demands for FCFS resources in different classes. This approximation is originally proposed by Bard to extend the applicability of the MVA algorithm [Bar79]. Second, the Schweitzer approximation is used to avoid recursion when solving the closed submodel of mixed product-form networks. We might improve the accuracy of this approximation by using a more advanced MVA variant, such as the Linearizer algorithm. Third, the impact of open classes is taken into account through the load concealment transformation. The effect of open classes is represented by elongating the service demands of queuing resources before the closed submodel is solved. See [Agr85] for a discussion on the use of load concealment. The last two approximations might be avoided by using an exact method for solving the underlying product-form queuing networks (e.g. one of the algorithms presented in [Con89]). However, the cost of this approach becomes prohibitive for large models.

Several approximations are also used when the original AQN is divided into primary and secondary networks. The first one of these approximations eliminates simultaneous resource possession by increasing service demands of those accesses that have embedded synchronous calls. The increase in service demand equals to the time it takes to execute the embedded calls. This is a special case of the *state aggregation transformation* where the idea is to combine a set of strongly interacting states with a single state. A common way to use this transformation is to combine multiple resources into a single flow-equivalent resource [Men94, Jai91, Hav98]. In our case, however, state aggregation is applied on an access-by-access basis. As a result, we can remove all problematic states, i.e. those where a single job imposes queuing at several resources, but we can still keep track of individual resources and obtain performance metrics for them. See [Agr85] for a general discussion on state aggregation.

The second approximation required by the decomposition is the use of delay resources to represent various delays in the model. In the primary network, a delay resource is added to represent the delay imposed by direct uses of secondary resources. In the secondary networks, delay resources are added to represent the combined delay imposed by all other resources in the system. Iteration is needed to find approximate values for these delays. Jacobson and Lazowska use a similar approximation in their algorithm for solving queuing networks with simultaneous resource possessions [Jac82]. Unlike the MOD algorithm, however, their approach is limited to two layers of resources. A similar technique, *the method of complementary delays*, can also be used for modeling programs with internal concurrency [Hei83].

The third approximation takes place in step 6 of the algorithm where the total service demand for a resource is formed by adding together the service demands of all accesses to that resource. For FIFO servers in BCMP networks, this technique is valid only if the accesses have exponential service time distribution and the mean is the same for all of them. Unfortunately, FIFO is typically the only available scheduling discipline for software servers, but accesses have usually unequal service demands. This is a strong limitation that should be taken into account when estimating the use of the method of decomposition. If the different service demands do not follow the BCMP requirements, the algorithm still works, but the results may not be reliable.

The fourth approximation required by the decomposition is the technique for distributing the obtained residence times to the individual accesses that contributed to the service demand (step 4 in the MOD algorithm). This approximation follows loosely the methodology proposed by Haverkort for solving polling models [Hav98]. The idea is to consider the individual accesses of the resource as a set of separate queues that are polled by the resource.

The fifth approximation during the decomposition is the technique for reducing the service demand of recursive calls during step 6 of the method of decomposition. In our approach, we reduce the service demand of the resource to model the effect of recursion. However, this also reduces the resource's utilization and may yield too optimistic estimates for its saturation behavior. A similar approach is also used by Shum and Buzen to support non-exponential service demand distributions for FCFS resources [Shu77]. However, they propose to change the kernel of the MVA algorithm while our proposal keeps the MVA algorithm untouched and modifies the input data to obtain a similar effect. It might be possible to increase the accuracy of our algorithm by adding full support for the techniques proposed by Shum and Buzen, although this may reduce the robustness of the algorithm [Agr85].

Our framework does not currently support priorities. However, a number of known approximations, such as the shadow server approach from Sevcik [Sev77], could be used for adding support for priorities.

Agrawal provides an extensive discussion on MVA based approximations for treating priorities [Agr85]. However, it was not felt necessary to include these techniques into the MOD algorithm, since there is very little support for priorities in the current implementations for the CORBA platform. This is expected to change with future implementations of the real-time CORBA specification, and our performance modeling framework can be extended correspondingly.

Time complexity

We now consider the time complexity of each iteration of the MOD algorithm. Let $|R|$ be the number of classes in the AQN, $|K|$ the number of resources in the AQN, m the average number of accesses made by a class of jobs to a single resource, and n the average number of synchronous calls made by an access in the system. The values for m and n depend on the structure of the system being modeled. The complexity of step 3 requires that we know the time complexity of the MVA or a similar technique. If we use the Schweitzer approximate MVA and impose a limit to the number of iterations, the time complexity of solving the primary queuing network is $O(|K||R|)$ and the time complexity of solving the secondary queuing networks is $O(2|K||R|) = O(|K||R|)$. Hence the time complexity of step 3 is $O(|K||R|)$. Since the time complexities of steps 4 through 8 are $O(m|K||R|)$, $O(|R|)$, $O(mn|R||K|)$, $O(mn|R|)$, and $O(|R|)$, we may conclude that the time complexity of one iteration of the method of decomposition is $O(mn|R||K|)$.

While there is no predefined limit for the number of iterations, experimentation has shown that most models require less than 30. Also, the size of the model has virtually no effect on the required number of iterations. Hence, we may conclude that time complexity is not an obstacle for using the method of decomposition for solving complex models.

4.9 Summary

In this chapter, we have defined augmented queuing networks that can be used for specifying performance models of CORBA based distributed systems. The support for synchronous calls is particularly important since it leads to straightforward modeling of communication in such systems.

We have also presented an algorithm for solving AQNs for a number of relevant performance metrics. The idea in the algorithm is to decompose an AQN into a set of product-form queuing networks. The parameters of these networks are not completely defined after decomposition, but a number of mutual dependencies can be pointed out between them. An iterative technique is used to determine approximate values for the missing parameters. During each iteration, we solve the set of auxiliary networks with efficient MVA based algorithms and the obtained results are used to change the parameters for the next iteration.

The algorithm supports recursion and circular calling dependencies between resources. Both are commonly used in object-oriented systems. In addition, special attention has been paid to the convergence of the algorithm in order to make it suitable for practical software engineering. The algorithm uses approximations that are based on known techniques and have been used in various contexts. Hence, the method of decomposition can be considered as a novel combination of existing techniques for solving queuing networks representing CORBA based distributed systems.

Chapter 5

UML based performance modeling

In this chapter, we describe a UML based performance modeling notation and define a collection of modeling techniques for specifying high-level performance models of CORBA based distributed systems. The proposed techniques support the modeling of relatively complex systems by dividing the overall model into a set of simple and manageable UML diagrams.

We start with a brief introduction to the UML and focus on those UML features that are used in our framework. Then, we gradually build up our collection of modeling techniques for CORBA based distributed systems. Along with the relevant UML diagrams, we also present the corresponding performance modeling language (PML) constructs. An abstract grammar for the PML is given in Appendix A, and the detailed syntax and semantics are discussed in [Käh99c]. The PML is a textual notation for those elements in UML diagrams that are relevant for performance modeling. In this work, the main purpose of PML is to facilitate the automatic analysis of performance models while the actual modeling is done with the UML. Finally, we describe how UML and PML representations can be transformed into solvable AQN representations.

5.1 The Unified Modeling Language

We have chosen the UML notation for our framework because of its increasing popularity in the object community and because it is particularly well suited for large and complex systems. In this work, version 1.1 of the UML is used. Comprehensive discussions on the UML are given, for example, in [Rat97, Eri98, Rum99].

The Unified Modeling Language is a notation for specifying, visualizing, constructing, and documenting the artifacts of software systems. In addition, the UML can be used for business modeling and describing other non-software systems [Rat97]. It builds upon ideas and techniques that were developed in earlier object-oriented methodologies. In particular, the OMT [Rum91], Booch [Boo94], and OOSE [Jac92] methodologies had an important influence on the UML. The first draft version of the UML was released in 1995 by Rational Software Corporation. Version 1.1 of the specification was published in 1997 and later adopted by the OMG. Further development of the UML specification is now an integral part of OMG's activities.

The primary goal of the UML is to provide an easy-to-use visual modeling language for developing and exchanging various kinds of models. Unlike earlier notations that were usually integral parts of development methodologies, UML is trying to be independent of the development processes, development tools, and programming languages. To reach these goals and to ensure semantic consistency across different modeling environments, two important objectives have been set. On one hand, there is a need to define a formal basis for the modeling language and, on the other hand, standard extensibility and specialization mechanisms are needed to ensure compatible extensions of the core UML in different environments. Finally, the UML attempts to provide support for various high-level development practices and concepts in the domain of object technologies. For example, special modeling techniques are proposed for supporting frameworks, patterns, and components.

The UML specification has two essential parts. *The UML Semantics* defines the abstract syntax and the exact semantics for the core modeling concepts. *The UML Notation Guide* defines the corresponding graphical notations. The notation guide also provides examples for using UML. An important extension to the UML is the *Object Constraint Language* (OCL). It is used for specifying well-formedness rules in the UML semantics document, but it can also be used for describing application-specific constraints and expressions, such as preconditions and postconditions for operations. The core UML has been extended towards different application domains by using standard extension features. Version 1.1 of the specification contains extensions for business modeling and for Rational Software's own Objectory software engineering methodology. Additional extensions are being specified by the OMG.

Diagram types

There are nine diagram types in the UML. Each diagram type describes a part of the system from a certain point of view. A complete model usually contains several interrelated UML diagrams that together provide a complete picture of the target system. The following diagram types are available:

- Use case diagrams,
- Class diagrams,
- Object diagrams,
- Statechart diagrams,
- Sequence diagrams,
- Collaboration diagrams,
- Activity diagrams,
- Component diagrams,
- Deployment diagrams.

A *use case diagram* shows the relationships between external actors and uses cases describing the system. Use cases are short descriptions of the system's functionality as seen by end users and other external actors when they are interacting with the system. Use cases are a widespread technique for specifying system requirements in object-oriented analysis and design methodologies [Jac92, Jac98, Sou98]. Use case diagrams provide a tool for organizing the use cases and their mutual relationships but the actual use case descriptions are not part of the diagram.

A *class diagram* shows the static structure of the system in terms of classes and their relationships. A class diagram may also contain other modeling elements, such as interfaces, packages and individual objects. Class diagrams have an essential role in object-oriented analysis and design, as they provide the main tool for describing the logical organization of the system being modeled. Figure 20 contains an example of a simple class diagram. It contains a generic class representing a directory entry, and two detailed classes for files and directories. Both are subclasses of the generic directory entry class.

An *object diagram* shows a snapshot of the system's static structure in terms of objects and links between them. However, since class diagrams are allowed to contain objects, the UML specification does not make a clear distinction between class and object diagrams.

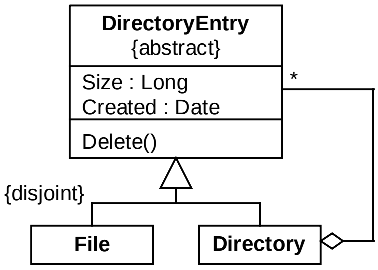


Figure 20. A class diagram example.

A *statechart diagram* shows the states of an object in a particular class. It also indicates the state transitions that can be triggered by various events, such as messages that are sent by other objects in the system. Statechart diagrams are often drawn only for those classes that have clearly distinguishable states and change their behavior depending on their state.

A *sequence diagram* shows an interaction between objects as a sequence of messages. The time dimension makes the diagrams particularly easy to read. Sequence diagrams may also contain classes instead of objects. In such cases, however, the diagram should indicate all alternative interactions that may take place at run-time.

A *collaboration diagram* is also used for presenting interactions between objects but now the context is an object diagram indicating links between the participating objects. Hence, collaboration diagrams do not have a time dimension and, accordingly, the order of messages must be indicated with sequence numbers. Collaboration diagrams can also contain classes instead of objects. Sequence and collaboration diagrams often provide alternative ways to express the same information. Figure 21 illustrates a simple interaction with sequence and collaboration diagrams.

An *activity diagram* is a special kind of state diagram in which most states represent the execution of an operation and most transitions are triggered by the completion of these operations. The scope of an activity diagram may vary from a set of use cases to the implementation of a single operation. Activity diagrams are often used in situations where most events are generated internally by a procedural flow of actions, while statechart diagrams are more appropriate in the presence of externally generated asynchronous events.

A *component diagram* shows the compile-time and run-time dependencies between software components, such as source code components,

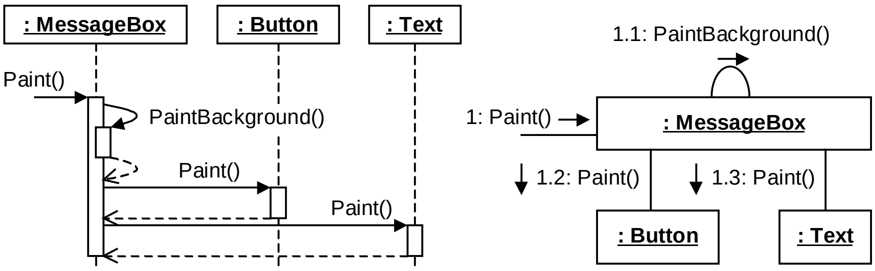


Figure 21. A sequence diagram (left) and a collaboration diagram (right) can be used to model the same interaction.

binary components, and executable components. Component diagrams provide a link from the logical design to the actual implementation modules that make up the system configuration. If there are changes in the logical design, the component diagrams can be used to track down the necessary changes to the individual components.

A *deployment diagram* shows the run-time configuration of a system. This may include, for example, nodes, processes, run-time software components, and objects. Deployment diagrams are used to describe the physical architecture of the system as opposed to its logical structure or behavior. Figure 22 illustrates a deployment diagram with two nodes and a component in each node. The *AppServer* component in the server node is offering the *AppService* interface that the user interface component is using from the client node.

We use five UML diagram types in our performance modeling framework. First, class diagrams are used to describe the static structure of the performance model. This includes the model's resources and the operations that provide access to them. Second, collaboration diagrams are used for describing the model's workloads through sequences of operations to the model's resources. Each workload is characterized by a separate collaboration diagram. Third, sequence diagrams provide an al-

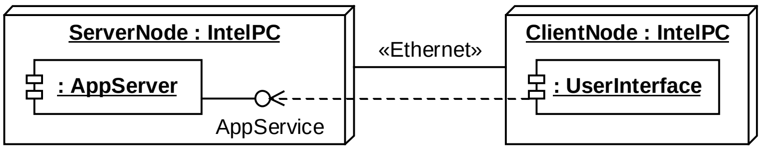


Figure 22. A deployment diagram example.

ternative way to represent the workloads. Both diagram types can be used interchangeably in our framework. Unless otherwise noted, the term collaboration diagram is used to indicate that either diagram type can be used. Fourth, deployment diagrams are used to describe the actual run-time configuration of the model. Finally, object diagrams are used to model the run-time configuration when there is not enough information for deployment diagrams. This may happen, for example, at an early stage of the development cycle.

Basic modeling concepts

The UML defines general-purpose modeling concepts that can be used in all diagrams. These concepts include

- Packages,
- Dependencies,
- Notes.

A *package* is a grouping of model elements. Packages are intended for organizing complex models and they can contain all kinds of modeling elements, such as diagrams and other packages. A *dependency* indicates a semantic relationship between two or more model elements, such as a refinement or a usage relationship. Dependencies are indicated with dashed arrows. A *note* is a graphical symbol that contains textual information, such as comments, constraints, or implementation examples. Figure 23 illustrates the use of packages, dependencies, and notes.

A large number of UML elements are dedicated to the static structure of the model. Important static modeling concepts include

- Classes,
- Associations,
- Composition,
- Generalization,
- Objects,
- Interfaces.

A *class* is a description for a set of objects with similar structure, behavior, and relationships. The notation for a class is a rectangle with three compartments. The top compartment contains the class name and other general properties of the class. The optional middle and bottom com-

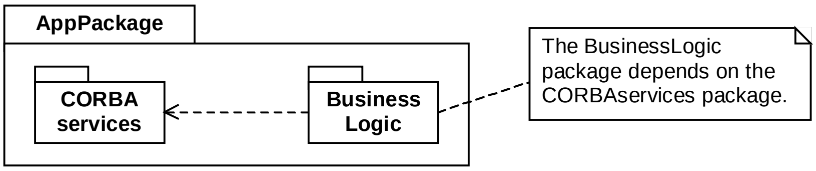

AppPackage
CORBA services
Business Logic
The BusinessLogic package depends on the CORBAservices package.

Figure 23. A diagram with packages, dependencies, and a note.

partments contain the *attributes* and *operations* of the class. There is a rich variety of options for presenting them at different levels of detail. The example in Figure 20 contains three classes: *DirectoryEntry*, *File*, and *Directory*.

An *association* indicates relationships between two or more classes. A recursive association may be defined from a class to itself. An association end may indicate its *multiplicity*, i.e. the range of allowed cardinalities, with a list of integers or integer intervals. An asterisk (*) indicates an unlimited upper bound in an interval. An association end may also have an arrowhead for indicating unidirectional navigability, and a hollow diamond for indicating aggregation. The example in Figure 20 uses aggregation to indicate that a directory may contain any number of directory entries.

Composition is a special form of aggregation that implies strong ownership and coincident lifetimes. Composition can be shown by drawing an association with a filled diamond. An alternative way to indicate composition is to draw class symbols inside the container class, or to include the contained classes in its list of attributes. Figure 24 illustrates three ways to indicate composition in a class diagram.

Generalization is a relationship between a more general element and a more specific element that is consistent with the first element and adds

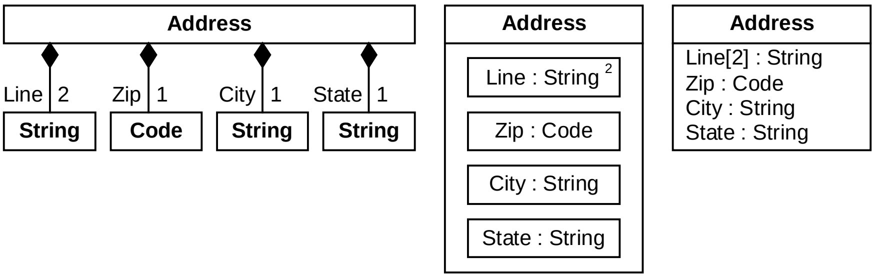


Figure 24. Three ways to indicate composition.

new information. Inheritance relationship between classes is a common case of generalization. The notation for generalization is a solid line from the more specific element to the more general element and a hollow triangle at the end of the line. Figure 20 illustrates generalization: the *DirectoryEntry* class is a generalization of the *File* and *Directory* classes.

An *object* is an instance of a class with an identity and a set of attribute values. The notation for an object is a rectangle with two compartments. The top compartment contains an optional object name followed by a colon and a class name. All three elements are underlined. The optional bottom compartment shows a list of attribute names, types, and values. Objects can be composed like classes. A *link* indicates a relationship between two or more objects, and it is an instance of an association between the classes that correspond to these objects.

An *interface* specifies externally visible operations of a class, component, or other similar entity, without specifying its internal structure. An interface can be considered as a class that only has a list of abstract operations. The notation for an interface is that of a class with the addition of the text «interface» at the name compartment. The notation for interface realization is a small circle representing the interface and a solid line to the elements that support it. If a class or other modeling element is using an interface, a dependency arrow can be drawn from this element to the interface. This is illustrated in Figure 22 where the *AppServer* component realizes the *AppService* interface.

The modeling of dynamic behavior requires additional UML elements. We only discuss those elements that are needed in sequence and collaboration diagrams. Both diagrams use *messages* to model the communication between objects. Common message types include

- Simple messages,
- Synchronous calls,
- Asynchronous messages.

A *simple message* represents flat flow of control and it is denoted with an open arrowhead (→). A *synchronous call* represents nested flow of control, such as a procedural call, and it is denoted with a filled arrowhead (➝). The return from the call may be indicated with a dashed line and an open arrowhead (⇠). An *asynchronous message* represents explicit asynchronous communication within procedural execution and it is denoted with a half-arrowhead (⇀). Figure 21 illustrates the use of synchronous calls.

Hierarchical sequence numbering can be used to indicate the order of messages. Messages that are sent at a particular level of procedural calls are prefixed with the sequence expression of their procedural level. In Figure 21, for example, messages 1.1, 1.2, and 1.3 are sent sequentially during the activation of message 1. Concurrent execution is indicated with labels. For instance, messages 1.1*a* and 1.1*b* would be sent concurrently within the activation of message 1.1. A sequence expression may be preceded by a *guard condition* in square brackets. The corresponding message is sent only if the condition evaluates to true. In a similar way, iteration can be indicated with an asterisk followed by an iteration expression. The UML does not mandate any particular syntax for the guard conditions and iteration expressions. The following example illustrates both concepts:

```
[x > 0] 1.1 *[i = 1..5]: Update(i)
```

Sequence expressions are often omitted from sequence diagrams, since the time axes already indicates the execution order.

Extension techniques

Extension techniques have a key role in the UML as they offer a controlled way to broaden the scope of the language while keeping the UML core relatively small. The following extension techniques are available:

- Stereotypes,
- Properties,
- Constraints.

A *stereotype* defines a new modeling element that is based on an existing one. The appearance and form of the stereotype is the same as that of the original element but the intent is different. In many cases, the aim of a stereotype is to indicate a specific usage. Stereotypes are represented in the same way as the base elements except that the name of the stereotype is placed above the element name within angle brackets (« »). It is also possible to define graphical icons for representing stereotypes. The UML specification itself defines various stereotypes to be used in different diagrams. For example, dependency has a number stereotypes, such as «call», «copy», and «instance», that indicate the nature of the dependencies that may exist between modeling elements.

User defined properties can be specified for any UML modeling elements using tagged values. A *tagged value* has the general form

keyword = value

where *keyword* indicates the name of the property and *value* is a string indicating the corresponding value. The value can be omitted for properties of the Boolean type. In such cases, the value is assumed to be *true*. The properties of an element are specified in a list of tagged values that is placed inside a pair of braces. For example, the tagged value *abstract* in Figure 20 indicates that *DirectoryEntry* is an abstract class, i.e. it cannot have instances of its own.

A *constraint* is a semantic condition that is required to be true at all times. Constraint texts are placed within a pair of braces and a dashed line or arrow can be used to indicate the scope and direction of the constraint. For example, the constraint *disjoint* in Figure 20 specifies that the inheritance relationship is disjoint, i.e. the *File* and *Directory* subclasses cannot have common instances. The object constraint languages can be used to describe complex constraints.

In our framework, we are using properties as our primary extension technique. The main reason for selecting this approach is its user-friendliness: adding properties does not change the structure of the UML diagram and, consequently, properties can be embedded into existing UML designs with little effort. In addition, most UML modeling tools already have strong support for properties. The expressive power of properties is somewhat limited and, therefore, it has been necessary to define one additional stereotype for specifying the characteristics of a network connection. See Section 5.6 for more details.

5.2 Resource representation

We now proceed to the modeling techniques that are specific to our framework. The primary goal is to use UML for representing performance models of CORBA based distributed systems. We start with the modeling of resources.

Software and hardware resources are represented with UML classes. To distinguish these *resource classes* from other classes, we mark them with the *queue* or *delay* properties. The *queue* property denotes a queuing resource, such as a single-threaded software server or a physical device.

The *delay* property denotes a delay resource, such as think time, network delay, or a multi-threaded software server that does not impose queuing for its clients.

A request to an operation in a resource class indicates an access to that resource. In addition, a request to an operation in a non-resource class that is contained within some resource class is also considered as an access to that resource. Service demands for these operations can be defined explicitly with user-named properties. An operation may have a single property for the total service demand (e.g. $d = 10$), or it may have multiple properties to indicate the individual elements of the service demand. For example, there may be separate properties for the disk, CPU, and network adapter. Service demands are given in time units. For software resources, service demand indicates the measured or estimated time to execute the operation using whatever hardware resources necessary. For hardware resources, service demand indicates the actual time during which the corresponding device performs some work.

The model designer can freely select the symbols for service demands. If a symbol is not used in any triggering condition (see Section 5.4) and if it is not explicitly bound to a lower-level resource (see Section 5.5), the service demand represented by the symbol is directly associated with the resource to which the property is attached. In our examples, we com-

```
class CDatabase {
  property queue;
  Read() {cpu=10,disk=50};
  Status() {cpu=5};
  Write() {cpu=10,disk=80};
};
```

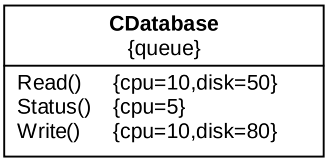


Figure 25. A queuing resource with three operations.

```
interface IDatabase {
  Read() {cpu=10,disk=50};
  Status() {cpu=5};
  Write() {cpu=10,disk=80};
};

class CDatabase {
  property queue;
  realizes IDatabase;
};
```

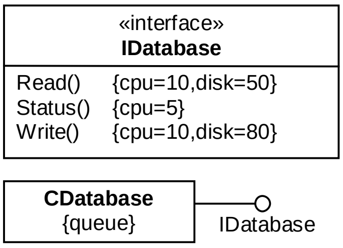


Figure 26. An example interface.

```
class Queue {
  property queue;
};
class Delay {
  property delay;
};
```

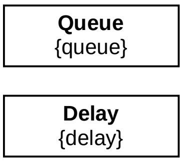


Figure 27. Definitions for the Queue and Delay classes.

monly use the following symbols: *d* denotes the measured or estimated service demand that is usually bound directly to the associated resource, *cpu* denotes service demand to be bound to the CPU resource, *disk* denotes service demand to be bound to the hard disk, and *adapter* denotes the service demand to be bound to the network adapter.

Accurate service demands are difficult to obtain without measuring real systems. Therefore, estimates are often used at early stages of development to produce approximate performance models. Service demands are not mandatory in class diagrams since they can be also defined in the workload descriptions. Figure 25 illustrates a resource class *CDatabase* with three operations. The service demands of the operations are divided into CPU and disk components.

Service demands can also be specified for interfaces. Figure 26 shows how the *CDatabase* resource can be specified with an explicit interface. Resource classes can be used like any other UML classes and mixed freely with non-resource classes. To simplify the instantiation of resources, we assume the existence of two predefined resource classes, *Queue* and *Delay*, that do not have any operations or attributes. Their definitions are given in Figure 27.

5.3 Workload representation

Performance workloads are modeled with collaboration diagrams (*workload diagrams*). For closed workloads, the number of jobs is indicated with the *population* property. For open workloads, the arrival rate of jobs is indicated with the *arrivalrate* property. To comply with the requirements for BCMP networks, we only allow Poisson arrival processes in open workloads. Workload diagrams resemble those used for the AQN representation but there are two differences. First, workload diagrams usually contain classes instead of objects. This way, the defined workloads are generic, and the indicated service demands are divided between

```
collaboration MainLoad {
  property population=100;
  1: CFront.Start() {cpu=10};
  1.1: CDatabase.Read();
  1.2: CDatabase.Write();
  2: User() {d=500};
};

actor User { property delay; };
class CFront { property queue; };
```

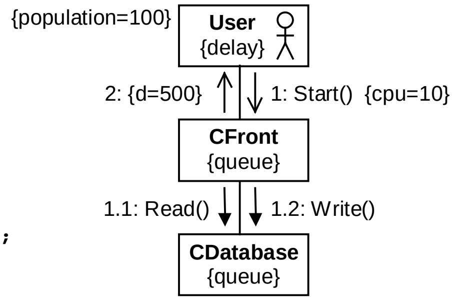


Figure 28. An example workload diagram.

the instances of the target class. Second, only application level resources need to be shown since infrastructure and network resources are activated with triggering properties (see Section 5.4). Figure 28 illustrates a simple workload diagram where 100 users are accessing a database through a front-end application.

Service demands need not be specified explicitly in workload diagrams if they are given in class or interface specifications. Service demands in workload diagrams override those given elsewhere. It is possible to use anonymous invocations that only indicate the target resource but do not name the operation. They always require explicit service demand. The example in Figure 28 models the think time of the end user with an anonymous access to the *User* delay resource.

The *thinktime* property can be used to represent think time or any similar delay when there is no need to explicitly present the corresponding delay resource (e.g. the end user). For instance, the example in Figure 28 can be specified in a more concise form by omitting the *User* class together with the anonymous invocation to it, and by adding the *thinktime* property to the diagram.

Any class can be marked with the *singleuser* property. This Boolean property affects the class itself and all resources that it contains. An access to a *singleuser* resource is considered to have no contention. As a result, its service demand is simply added to the value of the *thinktime* property. Effectively, resources that are marked with the *singleuser* property are excluded from the underlying AQN model except for the increase in think time. This is typically done for user workstations and other similar resources that are dedicated to a single job and cannot impose queuing. In some cases, however, it may be preferable to simplify the model by leaving out such resources and by increasing the *thinktime* property explicitly.

For conditional messages in collaboration diagrams, we specify a performance oriented shorthand notation. Instead of giving a guard condition, we write down explicitly the execution probability for the message. This way, the message

```
[x > 0] 1: Operation() {probability = 0.7}
```

can be shortened to

```
[0.7] 1: Operation()
```

in performance models. The same notation is also used for iteration. For example, the expression

```
1.1: *[i = 1..10]: Operation()
```

can be replaced with

```
[10] 1.1: Operation()
```

This shorthand notation can be used in sequence and collaboration diagrams, and it is the only way to express conditional execution and iteration in PML. It may sometimes be difficult to estimate the correct execution probabilities and iteration counts since both can be data dependent. Such data dependencies are further discussed in [Smi90] together with a number of examples for obtaining acceptable estimates.

5.4 Triggering properties

Triggering properties represent side effects of the application behavior in the infrastructure and in the network. These effects are normally excluded from application-level workload diagrams to keep them readable. Examples include network delays and context switching delays during inter-process communication. Any UML class may specify up to nine triggering properties, as listed in Table 1.

The value of a triggering property is a reference to a collaboration diagram (*triggering diagram*) describing the actions that take place when the triggering condition is satisfied. We use UML package names to indicate these references, but it is possible to use other referencing mecha-

Table 1. Triggering properties.

Property	*Triggering condition*
requestin	Request from an outside class
replyin	Reply sent by an outside class
msgin	Any incoming message (i.e. requestin or replyin)
requestout	Request to an outside class
replyout	Reply to an outside class
msgout	Any outgoing message (i.e. requestout or replyout)
requestpeer	Request sent between first-level contained classes
replypeer	Reply sent between first-level contained classes
msgpeer	Any internal message (i.e. requestpeer or replypeer)

nisms, such as dependency stereotypes. Figure 29 illustrates a process with a constant context switch delay when a request or a reply arrives from another process. Requests that are sent and received within the same process do not satisfy the *requestin* condition and do not have a delay associated with them.

A triggering diagram can use the service demand values of the operation that satisfies the triggering condition. Suppose, for example, that each operation contains an *adapter* property representing the use of the network adapter. Figure 30 illustrates a node where the *adapter* property is mapped to the *Adapter* device. Service demands in triggering diagrams can be adjusted with arithmetic expressions. For instance, a fast network adapter might be modeled with the property *d* = *adapter* * 0.9. These and similar arithmetic expressions can refer to properties that have been specified for any modeling element. For example, if the *CNode* class has

```
class CProcess {
  property msgin=CProcessMsgin;
  CtxSwitch : Delay;
};

collaboration CProcessMsgin {
  1: CtxSwitch() {d=2};
};
```

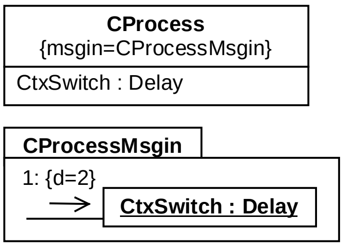


Figure 29. A process with a constant context switch delay for incoming messages

```
class CNode {
  property requestin=CNodeRequestin;
  Adapter : Queue;
};

collaboration CnodeRequestin {
  1: Adapter() {d=adapter};
};
```

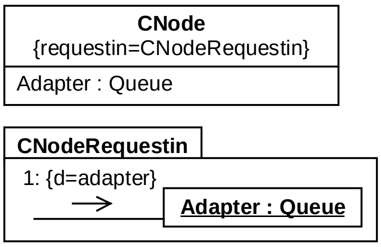


Figure 30. The service demand of the network adapter depends on the request satisfying the triggering condition.

an *AdapterRate* property for indicating the relative rate of the adapter, the service demand of the network adapter could be defined with the expression *d* = *adapter* * *CNode.prop.AdapterRate*.

5.5 Service demand binding

If a service demand property is not used in any triggering condition, the service demand is associated directly with the object that receives the invocation. In some cases, however, the service demand should be bound to some other resource. For example, if the model includes a CPU resource for each node, the service demands of software server requests should be bound to the CPU resource in order to model CPU contention.

Service demand binding is implemented by declaring a *binding resource,* such as a CPU, with a *binding property*. The binding resource is often placed inside a container class, such as a node, to indicate the scope of the binding. The binding property indicates the name of the service demand property it binds. Figure 31 illustrates how the *cpu* and *disk* properties can be bound to model CPU and disk contention. Arithmetic expressions can be used in binding properties. For example, the binding expression *d* = *cpu* * 1.5 could be used to model a slow CPU.

Nested service demand binding is allowed. For example, a disk driver

```
class CNode {
  Cpu : Queue {d=cpu};
  Disk : Queue {d=disk};
};
```

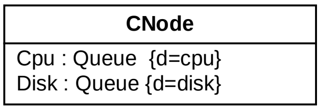


Figure 31. Service demand binding

resource could bind all application-level service demands for file access. The service demand of this resource might then be bound to the actual hard disk. It is also possible to bind the same service demand more than once. For example, if a node has two disks and data accesses are equally divided between them, this might be modeled by splitting the service demand of each disk access with the binding property *d* = *disk* * 0.5 for both disks.

5.6 Network connections

In UML deployment diagrams, a connection between nodes may be refined by a stereotype for identifying the communication protocol or the network medium (e.g. «*TCP/IP*» or «*Ethernet*»). We propose an extension to the UML for specifying the details of such association refinements. Essentially, we need the same triggering properties that are available for UML classes. Hence, we define a class stereotype «*connection*» for specifying association stereotypes for network connections. Usually, connection classes contain a delay resource for representing network delays and a number of queuing resources for modeling physical devices, such as modems or routers. These resources are typically accessed only from triggering diagrams. In other words, network access is typically a side effect of some other behavior in the system. Figure 32 shows a connection with a constant delay for each message that passes between two nodes in a «*LAN*». Figure 33 shows how this connection can be used in a deployment diagram.

```
connection LAN {
  property msgpeer=LANMsgpeer;
  Latency : Delay;
};

collaboration LANMsgpeer {
  1: Latency() {d=5};
};
```

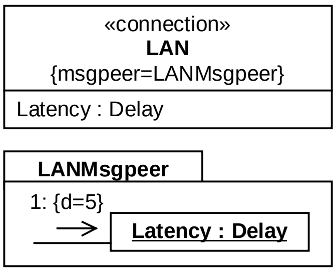


Figure 32. An example definition for a network connection.

```
MyLAN : LAN {
  Node1 : CNode {
    : CProcess {
      : CFront;
      [2] : CDatabase;
    };
  };
  Node2 : CNode {
    : CProcess {
      : CFront;
    };
  };
  Node3 : CNode {
    : CProcess {
      : CDatabase;
    };
  };
};
```

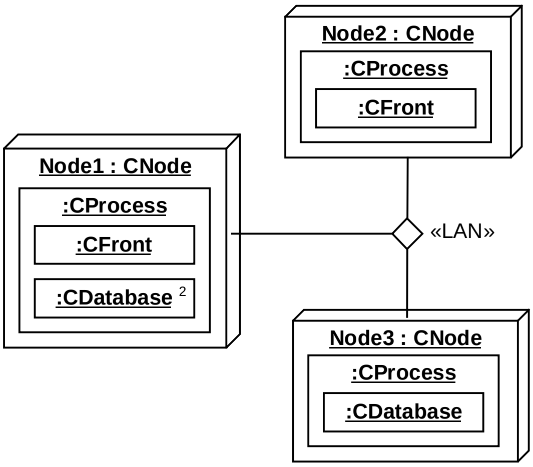


Figure 33. An example deployment diagram.

5.7 Run-time configuration

A description of the run-time configuration is needed for obtaining performance metrics for the modeled system. This can be carried out with deployment diagrams containing elements for the relevant physical entities, such as nodes, networks, and processes. Application objects are instantiated to their appropriate locations in the configuration. Figure 33 illustrates a deployment diagram of a simple system that contains three nodes attached to a LAN.

In many cases, however, the full run-time configuration is not known. In such cases, the configuration can be described with simple object diagrams, and the effect of some deployment choices is not present in the obtained results. Still, it may be possible to obtain useful metrics for the model. For example, the model might be used for identifying potential bottlenecks in the software design.

The full configuration of complex systems may be specified with several deployment diagrams. For example, there may be separate diagrams for each node, and an additional diagram for the network connections between the nodes.

5.8 Creating the AQN representation

So far, we have discussed how performance models can be represented in terms of UML modeling elements and diagrams. We now describe how such models can be transformed into multi-class AQNs so that the method of decomposition can be applied. Section 5.9 illustrates the transformation with an example.

Classes. The transformation of UML classes is straightforward. Each instance of a resource class is mapped to an AQN resource. All other classes disappear during the transformation. The actual number of instances for a particular resource class is determined from the object and deployment diagrams. An additional *thinktime* delay resource is added to the AQN.

Workloads. Each workload diagram is transformed into a class of jobs in the resulting AQN. The mapping is implemented in four steps. First, the *thinktime* property is mapped to an access to the *thinktime* resource. Second, the service demands for all invocations are determined from the workload, class, and interface specifications. The values in workload diagrams override those given elsewhere. If the target of an invocation is a non-resource class, the service demand is attributed to the closest surrounding resource class. Third, all accesses to *singleuser* resources are handled by increasing the service demand of the *thinktime* resource appropriately.

The fourth step during workload transformation is the expansion of class invocations into one or more accesses to resource objects. We use a special *class resolution algorithm* for this purpose. The algorithm is based on the use of resource contexts that are defined by packages and composite objects. For the first invocation in a workload diagram, the initial resolution context is the current package. For all subsequent invocations, resolution is first attempted in the chain of invocations that preceded the current one. If this fails, resolution is attempted in the smallest context containing the object that executed the previous invocation. If there is no match in the initial context, resolution is attempted in the next surrounding context, and so forth until the whole system is the resolution context.

When one or more instances of the target class are found in a resolution context, the service demand is distributed evenly among them, and the subsequent resolution contexts are determined by these instances.

Hence, a single invocation in a workload diagram may spawn any number of invocations in the AQN representation. The class resolution algorithm can be overridden by explicitly specifying a resolution context for the operation, or by using a named target object.

Consider the example in Figure 33. If the *Read* operation were invoked after an operation in *Node2*, the service demand would be distributed evenly between all three instances of the *CDatabase* class. However, if we did the same inside *Node1*, only the two database instances in *Node1* would be accessed. To ensure that all databases are accessed evenly, we can indicate the intended resolution scope explicitly with a compound name, such as *MyLAN.Read*. Such compound names can contain any number of elements to indicate the exact scope for the operation.

Triggering properties. If an invocation in a workload diagram satisfies a triggering condition, a sequence of additional resource accesses are copied from the corresponding triggering diagram into the workload diagram. A single message in a workload diagram may satisfy any number of triggering conditions for several objects. For example, an access to a remote object generates a message that leaves a node, passes through a network, and enters another node. As a result, three triggering diagrams might be involved in the transformation. If the message being expanded is a synchronous call, the calling object remains blocked during the additional accesses, and also the backward path is checked for triggering conditions. Otherwise, the sending resource is released immediately and direct uses are generated.

Service demand binding. The transformation generates additional invocations for service demand bindings. After the original access, a new access is made to the binding resource. If the target resource of the original access is a queuing device, the binding is transformed into a synchronous call. If the target resource of the original access is a delay device, a direct use is generated from the binding. Notice that the service demand of the original target resource may become zero after this transformation (i.e. all service demand properties get bound). This is typical for software resources when the performance model also includes a CPU resource.

Conditional execution and iteration. Conditional execution and iteration are both handled after expanding triggering properties and service demand bindings into additional resource accesses. For both cases, we require that the UML model contains a factor *f* indicating the branching probability ($f < 1$) or the repetition count ($f > 1$). The shorthand notation presented in Section 5.3 gives a convenient way for expressing this factor. All service demands that are within the affected execution path are simply multiplied by *f*. A single service demand may be multiplied by several factors if there are nested conditional messages and iterations.

Network connections. Association stereotypes representing network connections are treated as if they were container classes for the nodes that are contained within the network. Accordingly, the transformation for network connections is the same as the mapping for classes that have triggering properties or service demand bindings.

5.9 Example

This section illustrates UML based performance modeling with a simple example of a network monitoring system. The system operates as follows. A number of receiver objects accept messages from network elements and forward them to handler objects that are responsible for executing appropriate actions. A special database object maintains descriptions for the actions. The essential elements of the system are represented by the *CReceiver*, *CHandler*, and *CDatabase* classes. To illustrate the use of interfaces, the operations of the *CHandler* class are defined in the *IHandler* interface. Figure 34 illustrates the static structure of the system. Service demand estimates for the operations are in milliseconds.

```
class CReceiver {
  property queue;
  ReceiveJob() {cpu=10};
};
class CHandler {
  property queue;
  realizes IHandler;
};
class CDatabase {
  property queue;
  GetActions() {cpu=10,disk=50};
};
interface IHandler {
  AcceptJob() {cpu=5};
  DoActions() {cpu=20};
};
```

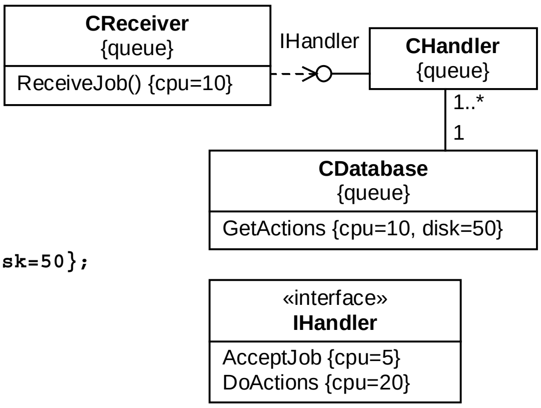


Figure 34. The static structure of the example system.

```
collaboration BgLoad {
  property arrivalrate=0.001;
  1: GetActions();
};

collaboration Jobs {
  property arrivalrate=0.007;
  property adjustopen=1;
  1: ReceiveJob();
  2: AcceptJob();
  2.1: GetActions();
  2.2: DoActions();
};
```

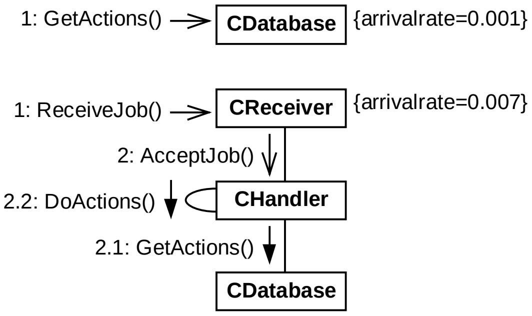


Figure 35. Workloads for the example system.

The model has two workloads: a background load for the database and the main load. The workloads are illustrated in Figure 35. The background load has an estimated rate of 1 database query per second. The main load represents the handling of messages from network elements. In steps 1 and 2, a message is received and forwarded to a handler. In steps 2.2 and 2.3, the handler consults the database and executes appropriate actions. We estimate an arrival rate of 7 messages per second.

The system uses an object-oriented infrastructure for implementing object communication. To keep the example short, we model the infrastructure with simple communication delays. A more detailed model would map communication delays to appropriate lower-level resources. We assume that there is an average 3 ms delay when the sender and re-

```
connection LAN {
  property msgpeer=LANMsgpeer;
  Latency : Delay;
};
collaboration LANMsgpeer {
  1: Latency() {d=3};
};

node CNode {
  property msgpeer=CNodeMsgpeer;
  Ctxswitch : Delay;
  Cpu : Queue {d=cpu};
  Disk : Queue {d=disk};
};
collaboration CNodeMsgpeer {
  1: Ctxswitch() {d=2};
};
```

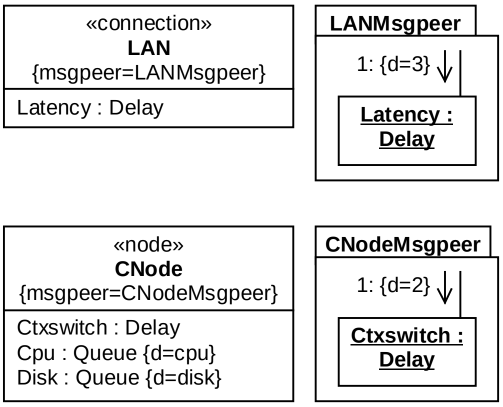


Figure 36. Node and network specifications for the example system.

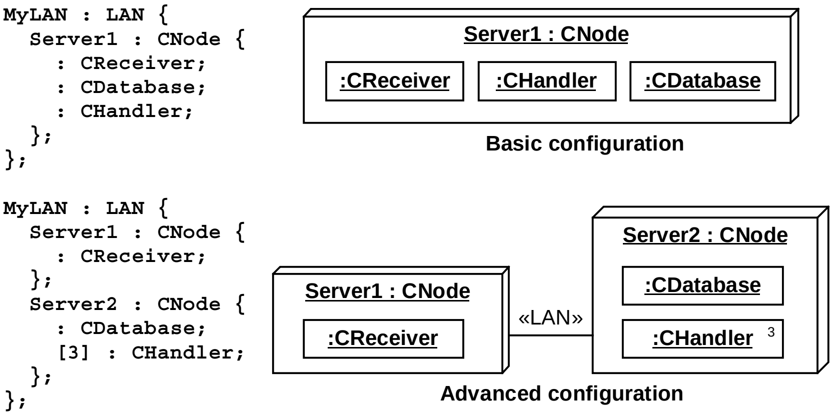
ceiver are in different nodes, and a 2 ms context switch delay when they are in the same node. We also model hardware resource contention by presenting explicit CPU and disk resources for all nodes. The service demands for application level operation requests are bound to these resources. The definitions for the network infrastructure and the nodes are illustrated in Figure 36.

We consider two different configurations for the system. The basic configuration has a single server node containing a receiver, a handler, and a database. The advanced configuration has a receiver in one server node and three handlers together with a database in another server node. These configurations are illustrated in Figure 37.

When the basic configuration is transformed into the AQN representation, the result contains six resources and two job classes as illustrated in Figure 38. Three issues are worth noting. First, the actual execution takes place exclusively in hardware resources as a result of service demand binding. Software resources are only controlling the order of accessing the hardware.

Second, the *Ctxswitch* resource illustrates the use of triggering properties. It is accessed three times, once for the *AcceptJob* message and twice for the synchronous call *GetActions*. However, there is no context switch for the recursive call *DoActions*.

Finally, the complexity of the diagram is worth noting. The AQN in Figure 38 was generated from a simple model with only three application-level messages in a single node. This way, number of automatically generated messages is small and the sequence diagram in Figure 38 is



```
MyLAN : LAN {
  Server1 : CNode {
    : CReceiver;
    : CDatabase;
    : CHandler;
  };
};
```

```
MyLAN : LAN {
  Server1 : CNode {
    : CReceiver;
  };
  Server2 : CNode {
    : CDatabase;
    [3] : CHandler;
  };
};
```

Figure 37. Two configurations for the example system.

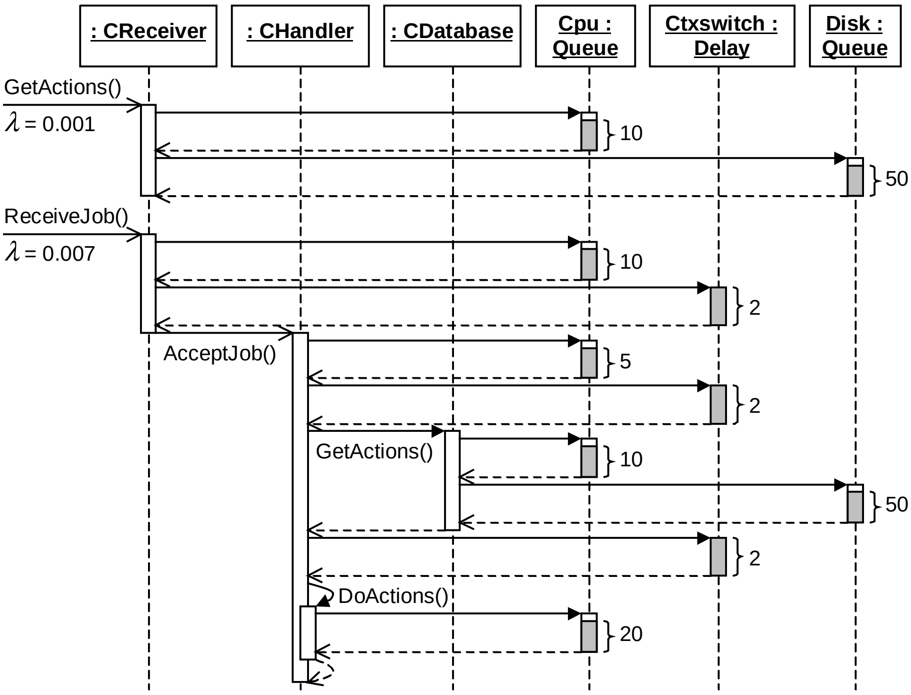


Figure 38. The AQN representation for the basic configuration.

still readable. However, more complex models would produce larger and less readable diagrams. This observation provides an additional justification for the proposed modeling techniques.

When the advanced configuration in Figure 37 is transformed into the AQN representation, the result contains twelve resources and 35 messages between them. In Figure 39, we use the PML notation for illustrating the result of the transformation. The list of instantiated resources is given before the actual workload messages.

A point worth noting in the advanced configuration is the expansion of the *AcceptJob* message into several resource accesses. There are three *Handler* instances in the AQN and the service demand of the *AcceptJob* message is divided evenly between them. In addition, all nested accesses that are made during the activation of *AcceptJob* are divided in three equal shares. Notice that the recursive call *DoActions* is routed to a single handler unlike the *AcceptJob* message. This is because the current job is already accessing one of the three handlers and it does not make sense to route a recursive access to any other handler.

```
MyLAN.Latency : Delay;
MyLAN.Server1.$CReceiver : Queue;
MyLAN.Server1.Cpu : Queue;
MyLAN.Server1.Ctxswitch : Delay;
MyLAN.Server1.Disk : Queue;
MyLAN.Server2.Cpu : Queue;
MyLAN.Server2.Ctxswitch : Delay;
MyLAN.Server2.Database : Queue;
MyLAN.Server2.Disk : Queue;
MyLAN.Server2.Handler[1] : Queue;
MyLAN.Server2.Handler[2] : Queue;
MyLAN.Server2.Handler[3] : Queue;

collaboration BgLoad {
  property arrivalrate = 0.001;
  1: GetActions() {d=0};
  1.1: Cpu() {d=10};
  1.2: Disk() {d=50};
};
collaboration Jobs {
  property arrivalrate = 0.007;
  1: ReceiveJob() {d=0};
  1.1: Cpu() {d=10};
  2: Latency() {d=1};
  3: Latency() {d=1};
  4: Latency() {d=1};
  5: AcceptJob() {d=0};
  5.1: Cpu() {d=1.66667};
  5.2: Ctxswitch() {d=0.666667};
  5.3: GetActions() {d=0};
  5.3.1: Cpu() {d=3.33333};
  5.3.2: Disk() {d=16.6667};
  5.4: Ctxswitch() {d=0.666667};
  5.5: DoActions() {d=0};
  5.5.1: Cpu() {d=6.66667};
  6: AcceptJob() {d=0};
  6.1: Cpu() {d=1.66667};
  6.2: Ctxswitch() {d=0.666667};
  6.3: GetActions() {d=0};
  6.3.1: Cpu() {d=3.33333};
  6.3.2: Disk() {d=16.6667};
  6.4: Ctxswitch() {d=0.666667};
  6.5: DoActions() {d=0};
  6.5.1: Cpu() {d=6.66667};
  7: AcceptJob() {d=0};
  7.1: Cpu() {d=1.66667};
  7.2: Ctxswitch() {d=0.666667};
  7.3: GetActions() {d=0};
  7.3.1: Cpu() {d=3.33333};
  7.3.2: Disk() {d=16.6667};
  7.4: Ctxswitch() {d=0.666667};
  7.5: DoActions() {d=0};
  7.5.1: Cpu() {d=6.66667};
};
```

Figure 39. The AQN representation for the advanced configuration.

5.10 Discussion

We briefly discuss two limitations of the proposed modeling techniques and consider potential improvements. The first limitation is the lack of support for all UML features in collaboration diagrams. In particular, it is not possible to spawn, kill, or synchronize threads, although multi-threading is supported as explained in Section 4.7. To have full thread support, the MOD algorithm should be extended so that explicitly created threads are transformed into additional classes of jobs in the primary and secondary networks. The details of this proposal are a topic for further research.

The second limitation is the partial support for UML diagram types. We intentionally use a relatively small set of UML elements in order to keep the framework easy to use. However, UML state and activity diagrams can express performance related information in a convenient manner, and future extensions of the framework should take them into consideration. For example, state diagrams indicate how objects change their behavior based on previous behavior, and this information should be taken into account when generating the AQN representation from high-level UML diagrams. In the same way, activity diagrams might be used to combine several workload diagrams into a single class of jobs for the AQN representation. This way, complex workload specifications could be split into smaller pieces.

Currently, the UML is rapidly progressing towards new directions. There is an ongoing initiative to specify the requirements for UML 2.0, and this initiative may change the way performance-related information is represented with UML [OMG99j]. The emerging UML profile for scheduling, performance, and time may also influence our framework [OMG99i]. An interesting feature in the framework could also be the support for stating performance requirements in UML diagrams. Such requirements can be specified with the current UML notation but they are currently treated as informal comments. Ideally, a modeling tool could automatically point out performance violations in the system design by comparing requirements against computed performance metrics.

5.11 Summary

In this chapter, we have presented a performance modeling notation that is based on the UML. In addition, a collection of modeling techniques has been proposed for CORBA based distributed systems.

In the proposed techniques, UML class diagrams are used for modeling resources in the performance models, and UML collaboration diagrams are used for representing the relevant workloads. In addition, a number of techniques have been proposed for increasing the modeling flexibility. Triggering properties and service demand binding are used for separating application-level specifications from the side effects that application behavior may have on the infrastructure. The «connection» stereotype has been proposed for describing the behavior of network connections. All proposed techniques are compatible with the core UML and follow the guidelines for UML extensions. Moreover, the techniques allow the coexistence of performance related modeling elements with other UML elements.

Finally, we have described how the UML diagrams can be transformed into the AQN representation. This way, a number of relevant performance metrics can be obtained for the models.

Chapter 6

Performance modeling methodology

In this chapter, we describe a concise methodology for using the performance modeling framework in the development of CORBA based distributed systems. First, we state the goals of the methodology and give an overview of its elements. Second, we describe a generic layered structure for performance models. Third, we propose a number of modeling activities for producing and refining performance models. Finally, we discuss how the methodology is related to software engineering. The primary contribution of this chapter is to indicate how the proposed modeling techniques can be used to enhance object-oriented software engineering practices.

6.1 Goal and overview

The goal of the performance modeling methodology is to support the creation, refinement, and exploitation of performance models in the development of CORBA based distributed systems. The methodology is intended to complement UML based software engineering methodologies, such as the Catalysis approach [Sou98] and the Unified Method [Jac98]. Our methodology focuses on performance model creation and refinement and, consequently, other elements of software performance engineering should be taken from dedicated SPE methodologies [Smi90, Jai91, Men94, Woo98].

Our methodology consists of a layered model structure for CORBA based distributed systems, and a number of modeling activities for creating and refining the models. The idea is to populate the skeletal model structure by the modeling activities in a stepwise process that progresses in parallel with the software engineering process. For this purpose, the following modeling activities have been identified:

- Extending use cases for performance modeling,
- Defining the software performance model,
- Defining the system performance model,
- Model validation and calibration.

The rest of this chapter briefly describes the layered model structure and each of the above activities.

6.2 Layered model structure

To cope with the complexity of CORBA based distributed systems, we separate performance modeling concerns into the following six layers:

- Application layer,
- Interface layer,
- Behavior layer,
- Infrastructure layer,
- Network layer,
- Deployment layer.

While the modeling techniques in Chapter 5 are designed to support this layering, it may sometimes be useful to have a slightly different structure for the performance model. For example, it may be appropriate to merge two adjacent layers for simple systems. In complex environments, on the other hand, it may be useful to divide a layer into multiple sublayers.

The *application layer* describes the application's static structure with UML class diagrams. Resources that are relevant for performance modeling are indicated with the *delay* or *queue* properties. In addition, if operations have been explicitly given in class diagrams, they may be equipped with service demand properties. Some middleware services that are explicitly visible at the application logic, such as the CORBA Naming Service, can also be modeled at this layer. Since the application layer requires only a few additional properties in otherwise normal class diagrams, it may be possible to maintain a single set of UML class diagrams for both functional and performance modeling.

The *interface layer* describes operations that are implemented by application layer objects. The use of an explicit interface layer reflects the object-oriented principle of separating interfaces from implementations (see [Käh98b] for a discussion). For CORBA based systems, a natural way of representing interfaces is to use the OMG IDL, although UML diagrams can also be used. The framework requires interfaces to be extended with service demands for each operation, unless these service demands have already been given at the application layer. Certain parts of service demand (e.g. marshaling time and network latency) could be inferred automatically from the interface specifications in well-controlled environments where the data contents of CORBA invocations are known in advance. However, as service demands highly depend on the semantics of the target operation, we assume in the current version of the framework that they are given explicitly in all cases.

The *behavior layer* describes the application's behavior in terms of interactions between application layer objects. The behavior layer is modeled with UML collaboration diagrams. All diagrams that represent essential workloads for the system must be extended with the framework features. This effectively means that a choice must be made between open and closed workloads. For the former, the arrival rate needs to be specified and, for the latter, the population is required. In addition, the probabilities of conditional execution paths and the number of repetitions for iterative execution paths must be estimated and noted explicitly in the workload diagrams.

The *infrastructure layer* describes the infrastructure support for the applications. This may include, for example, the middleware, the operating system, and the hardware. This layer is represented with class diagrams equipped with triggering properties and corresponding collaboration diagrams. Additional links between applications and the infrastructure layer can be created with service demand binding. Elements at this layer, such as processes and nodes, are organized into nested classes according to their physical structure. In complex models, this layer can be further divided into sublayers that correspond to the various elements of the infrastructure, such as the operating system and the middleware.

The *network layer* describes the performance characteristics of the underlying network. In particular, the «*connection*» stereotype allows network connections to use the same techniques that are available for the infrastructure layer. The network layer can be divided into sub-layers for modeling the network details. For example, LAN and WAN connections might be in separate layers.

The *deployment layer* specifies the run-time configuration in terms of objects, processes, nodes, networks, etc. with deployment diagrams. This layer can be described at different levels of detail. For example, at an early stage of development, it may be useful to instantiate only the application objects. Later, when the effect of network traffic is investigated, network layer objects might be added. Finally, a full model with all layers might be specified when the final system configuration needs to be validated.

The proposed performance model structure reflects the layered architectures that are commonly used for designing and analyzing CORBA based distributed systems [Käh98b]. A similar but less sophisticated layered performance model structure is discussed in [Wat97]. This approach divides performance models into three submodels that loosely correspond to our application, behavior, and deployment layers. A different modeling tool is used for each layer. In our case, it is assumed that all performance specification work is done with UML based tools and, consequently, a more fine-grained layering can be used to reflect better the structure of the target system.

6.3 Extending use cases for performance modeling

In our modeling framework, we exploit use case in two ways. On one hand, performance requirements can be specified with use cases and, on the other hand, the system workload can also be described with them.

Specifying performance requirements with use cases

The primary goal for use cases is to specify the functionality of the system. However, use cases should also specify related non-functional requirements, such as responsiveness, availability, accuracy, and recovery time [Jac92]. Some of the non-functional requirements are performance related and, hence, they can be used to validate the design and the actual system at different stages of development. For this to be successful, two conditions must be met. Performance related requirements must be

- Measurable and sufficiently detailed,
- Compatible with the metrics obtainable from the framework.

The first condition calls for detailed requirements like “the response time for this interaction must be within 3 seconds in 90% of all cases and it should never exceed 30 seconds”. See [Men94] for additional discussion. The second condition says that requirements in user understandable metrics, or *natural forecasting units* (see [Smi90]), must be converted into metrics that can be obtained from our framework. These metrics are

- Average response times,
- Throughputs,
- Utilizations,
- Queue lengths.

Some of the above metrics, such as the queue length of a particular resource, are not provided directly by the method of decomposition, but they can be easily obtained by applying Little's law. If the requirements are stated in natural forecasting units, the transformation may require additional information, such as the measured or estimated distribution of transaction types and other domain specific information.

Some performance requirements, especially those related to the overall throughput of the system, are not necessarily associated with any particular use case. Therefore, system analysts must explicitly look for such requirements and deal with them appropriately. Sometimes system-wide requirements can be converted into use case specific ones.

Describing workloads with use cases

If we wish to define system workloads with use cases, they must contain relevant scalability factors that indicate the expected intensity of the involved actions. Several issues need to be considered.

As a first step, it is important to identify use cases that are relevant for the performance of the system and to concentrate on them. If a use case is executed relatively seldom (e.g. a daily report) it should be left outside consideration, unless it is executed during a performance sensitive time period (e.g. during the nightly batch window).

Second, the correct scalability factors must be identified for the relevant use cases. A choice must be made between open and closed workloads. For open workloads, only the arrival rate needs to be estimated and the performance models tend to be relatively simple. For closed workloads, the expected number of users or workstations and the estimated think time must be specified. The resulting models are often more complex but the results may also be more accurate. It is possible to start modeling with open workloads and turn them into closed ones when more information becomes available.

Third, workloads must be formed correctly from the relevant use cases. The mapping between use cases and workloads is not necessarily one-to-one. For example, the complexity of the involved operations may force the analyst to describe a single logical sequence of actions with multiple use cases [Jac92, Jac98, Sou98]. When defining workloads for the model, such use cases should be combined to form a single workload.

Fourth, conditional and iterative executions need special consideration. If a use case has several alternative execution paths, the probabilities for each path should be estimated and noted down. For iterative executions, the average number of iterations should also be estimated.

6.4 Defining the software performance model

A *software performance model* is a high-level performance model of the application software. It is used when there is little or no information on the available infrastructure, hardware, or network. This way, developers can have a coarse performance model of the system already at an early stage of systems development. It must be borne in mind, however, that the metrics obtained from such models are often optimistic due to the missing elements.

In spite of the shortcomings caused by missing information, software performance models can be used for several purposes. We briefly mention three of them. First, the obtained metrics can be used to validate software designs against performance requirements. Unfortunately, the fact that the metrics at this stage are usually optimistic implies that some unsuitable designs pass this first check. Second, the models can help to identify critical elements in the application so that further modeling and design efforts can be focused on those elements. Finally, the software performance model provides a good starting point for building the system performance model that yields more accurate performance estimates when more design information is available.

A software performance model contains elements from the application, interface, behavior, and deployment layers. We briefly discuss how each layer can be obtained.

The application layer can be obtained from functional class diagrams with a straightforward transformation. The designer must identify and mark all resource classes representing software resources, as opposed to, say, container classes for structuring the system or interface classes for representing object capabilities. A distinction has to be made between queuing and delay resources. For example, a single-threaded CORBA object implementation is a queuing resource, while a multi-threaded implementation is typically a delay resource since it avoids queuing by spawning a new thread for each incoming operation request.

An optional task is the transformation of the application layer class diagrams into a format that better supports performance modeling. This may involve, for example, the removal of classes that are estimated not to be relevant for the performance of the system. This task simplifies later performance modeling and may improve the accuracy of the estimates but, as a side effect, it leads to separate class diagrams for the functional and performance models.

The interface layer may be obtained in different ways. If there is an IDL specification for the application interfaces, the complete interface layer can be specified by converting the IDL specification into UML interfaces and by equipping all operations with service demand estimates. However, if no IDL specification is available, a minimal approach may be taken by identifying those operations that are used at the behavior layer and by listing them in application layer diagrams with service demand estimates. As a result, the application and interface layers get merged. In both cases, service demand estimates can be either deduced from the operation descriptions, or they can be measured from a benchmark or a prototype. See [Jai91] for a discussion on making measurements, [Smi90] for obtaining estimates by analyzing the software structure, and [Gra91] for using benchmarks.

There are two practical ways to represent service demands. In the first approach, they are expressed directly by using the execution time that has been obtained, say, from measurements made for a prototype system. The advantage of this approach is the simplicity of interpreting the results since the performance estimates are given in the same units as the measurements. However, there may be differences in CPU speeds, CPU architectures, hard disk access times, etc. As a result, the same operation may have different service demands depending on its environment, and this may lead to inaccuracies in the model. There are ways to solve this difficulty at the infrastructure layer by using service demand binding (see Section 6.5), but this layer is not typically used in the software performance model.

The second way to represent service demands is to use normalized units to take into account differences in the hardware and software environment. For example, we might normalize CPU demands into machine cycles of a reference CPU. However, this approach complicates the interpretation of the results for the software performance model, since the obtained metrics must be converted back to environment-specific metrics before they can be used. If the infrastructure layer is included into the model, this conversion can be automated

A related issue is the question of using one or several service demand properties for a single operation. For example, there might be properties for representing the time to execute the actual operation, the time to marshal and demarshal the parameters, and time spent in the network. If the service demands have been obtained by measuring the total execution time of an operation in a prototype, it is usually better to have only one service demand property that directly reflects the measurement. This modeling style postpones the (usually inaccurate) decision that has to be made on the time shares for each property. If the need arises, the total execution time can be logically divided into several components at the infrastructure layer. However, if the service demands have been obtained from multiple measurements or from a detailed analysis of the software structure, it may be advisable to use several properties. This way, the model captures all information that is available on the system.

The behavior layer is created from those use cases that are relevant for the performance of the system. We assume that these use cases have been extended appropriately as discussed in Section 6.3. The procedure is straightforward: each step in the use case is converted into one or more messages in a workload diagram. Delays that are external to the system, such as think time or a call to an outside system, can be modeled either with the *thinktime* property or with explicit delay resources. The resulting workload diagrams must invoke only those operations that are available at the interface layer. For conditional and iterative messages, the estimated probabilities and repetition counts must be indicated at the workload diagram. If the service demand of a particular operation depends on the workload, the collaboration diagram may contain service demand estimates that override those given at the interface layer. Finally, the estimated arrival rates for open workloads, and the populations for closed workloads must be copied from the relevant use cases to the workload diagrams.

The deployment layer for the software performance model is needed to make the model solvable. At this stage, however, the deployment layer can be a simple object diagram without any information on the application structure, infrastructure, hardware, or network. The essential thing is to estimate the number of application level class instances.

In many cases, however, more information is available for building the deployment layer. For example, the locality of objects may be known with respect to other objects. In such cases, the deployment layer may contain additional objects for indicating the relevant modeling elements, such as processes, nodes, and networks. This information is used by the class resolution algorithm for mapping class-level invocations into object accesses.

6.5 Defining the system performance model

A *system performance model* is a full performance model of the target system. It contains all elements of the software performance model and additional information on the execution environment, the software infrastructure, the network, and the hardware. As a result, the model yields more precise metrics that can be used for tasks that were not possible with the software performance model. The system performance model can be defined at multiple levels of accuracy depending on the available information and on the goals of the modeling work. Hence, there may be several system performance models for the same target system at different stages of its lifetime.

The increased accuracy of system performance models makes them suitable for a number of tasks that cannot be carried out satisfactorily with software performance models. In particular, the metrics obtained from the model can be used to perform a full check of the system design against performance requirements. If the design does not meet the set requirements, the model also gives information on the possible causes of the problem. For example, an unsatisfactory response time may result from software or hardware bottlenecks, or from the delays caused by too many operation requests over the network. The former can be detected by looking at the utilization of resources, and the latter by checking the time share of network latency in the response time of a workload diagram. Another possible use for a system performance model is capacity planning for an existing or a future system. The goal of capacity planning is to predict the time in future when the system does not meet its performance goals due to increasing workloads, and to prevent this from happening in the most cost-effective way. See [Men94] and [Jai91] for extensive discussions.

A system performance model contains elements from all six layers. The application, interface, and behavior layers are similar to those defined for the software performance model. However, design decisions concerning the infrastructure may affect the software design and entail changes for the software performance model. For example, the designer might choose a particular CORBA platform and decide to use the available OMG Naming Service for configuration management. As a result, the behavior of some application classes must be changed to use the service. In addition, the objects that implement the Naming Service must appear among application objects in order to be accessible by them in the relevant collaboration diagrams.

The infrastructure layer models the software and hardware support that is needed for running the applications. The infrastructure layer is created by identifying software and hardware entities that are not part of the application but may affect the performance of the system. This layer commonly includes hardware resources, such as CPUs and hard disks, software resources, such as ORB daemons and operating system services, and elements for defining the software structure, such as processes, nodes, and software component wrappers. The layer is defined with class diagrams where the identified elements are represented by classes that are nested according to their physical structure. For example, a PC based workstation could be modeled with a node class that has attributes for representing a CPU, a hard disk resource, a network adapter, and an ORB daemon.

The influence of application activities on the infrastructure is modeled with triggering properties and service demand binding. The former is used for representing the side effects of messages passing between application objects, and the latter is used for mapping the execution of application operations into relevant hardware resources, such as the CPU and the hard disk.

When service demand binding is used to map service demands from the interface layer into the infrastructure layer, the effect of triggering conditions must be taken into account. There are two cases to consider. In the first case, the service demands indicate only the time to execute the operations. In this case, the service demand can be mapped directly to the infrastructure resource. In the second case, the service demand estimates at the interface layer include both the time to execute the operation and the infrastructure overhead. Such estimates can be obtained, for example, from prototype measurements. In this case, the infrastructure overhead (i.e. the service demand modeled by triggering conditions) must be subtracted from the interface layer service demand before mapping the service demand to the relevant hardware resource. Suppose, for example, that we have measured service demands for all operations and we estimate that the one-way communication delay is 0.5 ms. If the measured execution time for a synchronous call is t ms, the correct service demand for the CPU would be $t - 1$ ms. Figure 40 illustrates the infrastructure layer for implementing this example.

```
connection LAN {
  property msgpeer=LANMsgpeer;
  Latency : Delay;
};
collaboration LANMsgpeer {
  1: Latency() {d=0.5};
};

class CNode {
  Cpu : Queue {d=cpu-1};
};
```

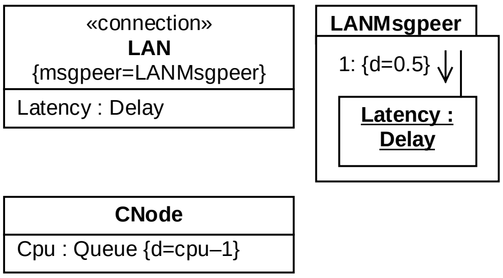


Figure 40. Network delays are subtracted from the interface layer service demands before mapping them to the CPU resource.

Additional problems arise from heterogeneous hardware and software. A possible technique is to multiply service demands with a suitable factor before they are bound to an infrastructure resource or used in triggering properties. For example, if the service demand for the CPU resource has been obtained from a 120 MHz Intel Pentium PC, an approximation for a 300 MHz machine could be obtained by multiplying the service demands with the factor 120/300. This could be expressed with a binding property $d = 0.4 * cpu$. To cope with such differences in the infrastructure, multiple definitions for nodes and other infrastructure elements may be needed. For example, there may be separate definitions for a 120 MHz node and a 300 MHz one. Inheritance can be used to avoid the rewriting of modeling elements that are common for both types of nodes.

The network layer is created in the same way as the infrastructure layer except that the identified resources are now related to the networking environment, and the structuring elements reflect the network topology. The essential modeling techniques, such as triggering properties and service demand binding, remain the same.

An additional issue has to be considered if network contention has an important role for the performance of the system and the model needs to be precise. MVA based performance analysis techniques are seldom sufficient for modeling network communications accurately due to the small set of available service time distributions, resource types, and scheduling disciplines. Also, the workload specification techniques in the framework are oriented towards software development and, consequently, they do not provide enough information for specifying network traffic at the protocol level. For example, to build a satisfactory model of an ATM switch, both the structure of the switch and the traffic patterns need to be taken into account [Bey98, Dau99]. As a result, we may conclude that our framework is not well suited for building detailed network communication models and, therefore, the modeling should be carried out with techniques and tools that are outside the scope of this work. See, e.g., [Hav98] for a discussion on a number of available techniques.

If external techniques are used for modeling the communication network, triggering properties and service demand binding provide a straightforward way to incorporate the results of such models into the system performance model. A special delay resource is first added to all network connections that employ such externally estimated delays. We describe two possible ways to use the added delay resources. On one hand, if the delay is the same for all messages passing through particular network connection, a triggering property can be used to invoke the delay resource. As a result, the behavior layer remains unchanged. On the other hand, if we want to have a message-by-message control over the delay, we can use service demand binding and introduce an additional service demand property for each message to indicate the correct delay. The price of this technique is the added complexity in the behavior layer. Sometimes, the metrics that are obtained from our framework can be fed to the external communications model as input parameters. In such cases, iteration may be needed to produce a combination of two models that have a compatible set of results and input parameters.

An alternative technique for modeling externally estimated delays is to extend the framework with load-dependent resources. In this approach, the service demand of the external resource is first estimated separately for different queue lengths, and a special load-dependent resource is then introduced into the system performance model. This technique requires a special version of the approximate MVA algorithm for handling load-dependent resources [Men94].

In some cases, network delays may be several orders of magnitude smaller than delays caused by software and hardware contention. This is the case with many LAN based applications if the messages do not contain large amounts of data, such as multimedia or large images. In such cases, the network layer can be removed altogether and the network delays can be included into the delays at the infrastructure layer.

The deployment layer is obtained by instantiating objects for the classes that have been specified at the application, infrastructure, and network layers. Possible objects include network connections, nodes, processes, software components, and application specific objects. The location of objects is indicated by using composition in deployment diagrams. In some cases, the appropriate number of objects and their locations can be determined from use cases but, more often, there are various configuration choices that the designer wishes to experiment with.

When experimenting with the different configuration choices, there are a number of things that the designer can do with deployment diagrams. For example, he can change the multiplicity of class instances provided that the class supports replication. He can also change the location of class instances if the class is not tied to any physical entity. In addition, he can switch between the *queue* and *delay* attributes of CORBA object implementations to experiment the effect of using a multi-threaded implementation instead of a single-threaded implementation, or *vice versa*. Moreover, he can add any new attributes to the instantiated resources to experiment with new hardware or infrastructure configurations. For example, a new CPU resource could be added to a node to experiment the effect of using multi-processor servers. Finally, any class specific properties can be overridden in the deployment diagram. For instance, the effect of a different communication media can be experimented by overriding the relevant triggering properties.

An additional check must be made to identify those resources that are queuing resources in the technical sense but are only used by a single job at a time and, therefore, do not impose queuing for their clients. For example, there may be resources in a client application that are only used by a single end user. To ensure that the model does not incorrectly impose contention on such resources, they can be marked with the *singleuser* property. In many cases, it is appropriate to mark all workstation nodes with the *singleuser* property.

6.6 Model validation

The validation of performance models is an essential part of software performance engineering. It improves the accuracy of the model and increases the developer’s understanding of the target system. We briefly discuss three tasks that are related to the validation of performance models: instrumentation, testing against performance requirements, and model calibration. We concentrate on issues that are specific to our framework. More information on validation can be found, for example, in [Men94] and [Jai91].

In performance sensitive systems, application *instrumentation* is important for at least two reasons. On one hand, the metrics that can be obtained from generic performance monitoring tools are seldom those that are the most relevant for the system, such as the average response time of a critical end user interaction. On the other hand, explicit software instrumentation allows designers to have full control over the measurement process. This way, the system under test is affected minimally by the measurement process, the results are more accurate, and they are easier to interpret. The instrumentation of operations in IDL interfaces is relatively easy in several CORBA implementations due to the available interceptor mechanisms [Ion97, Inp99]. These mechanisms can be embedded into CORBA interfaces without modifying the application code. There is an ongoing initiative to include such interceptors into the CORBA specification [OMG99c]. If the performance model has been constructed using the guidelines in Section 6.4, our framework also produces response time estimates for the operations in IDL specifications. We may conclude that operations at IDL interfaces are good candidates for instrumentation.

The testing of CORBA based systems against performance requirements is particularly challenging due to the flexibility of the CORBA platform. The available distribution transparencies support a large number of configuration choices. Therefore, the testing of a prototype or a complete system should follow a rigorous methodology to avoid unnecessary work and erroneous conclusions. A naïve approach, where one factor at a time is changed to see its impact on the performance, is unlikely to succeed due to the complex interactions caused by the omnipresent middleware. A number of possible methodologies and test designs are proposed in [Jai91].

The second problematic issue in the testing of CORBA based systems is the black box approach adopted by many ORB vendors. The source code of the ORB implementation is seldom available for detailed analysis. The implementation's threading policies, activation policies, scheduling disciplines, and other operational choices may strongly influence the performance of the system – especially with a large number of concurrent clients and intense workloads [Sch98a, Lit98]. Unfortunately, such effects may be difficult to predict only by reading the product's documentation. To cope with this problem, we suggest a two-phase testing methodology. First, an initial set of experiments should be designed and carried out for gaining an understanding of the system and the selected middleware platform. Once the results of the first round have been analyzed and the effect of the CORBA implementation has been understood, a second round of experiments should be carried out to produce the final test results that aim to validate the system against performance requirements.

The *calibration* of performance models is particularly useful for capacity planning where the figures obtained from performance predictions are directly used for decision making. Calibration means that the model is changed to match with the actual system. It is possible to use all parts of a performance model for calibration, but the infrastructure layer in our framework is particularly well suited for this purpose. Minor changes at the infrastructure layer can affect the behavior of the complete system. In addition, the application logic remains unaffected by such changes. Flowers and Dowdy discuss and compare a number of techniques for calibrating queuing network models [Flo89].

In our framework, the key techniques for calibration are service demand binding and triggering properties. Suppose, for example, that a performance model yields shorter response times than the actual system and we wish to calibrate the model by increasing our service demand estimates, say, with 20%. This calibration can be implemented with a single change in the model by using the factor 1.2 in the binding property that maps service demands to the CPU resource. Triggering properties on the other hand can be used for making modifications that are specific to a particular resource. For example, if we suspect that a service implementation imposes excessive queuing due to internal concurrency control mechanisms, we can add a special queuing resource and use triggering properties to invoke this resource during each access to the service. This way, the additional queuing at the service implementation is taken into account with minimal changes to other parts of the model.

6.7 Relationship with software engineering

This section briefly discusses the relationship between our performance modeling methodology and object-oriented software engineering. The three first modeling activities, extending use cases, defining the software performance model, and defining the system performance model, are directly connected to the requirements specification, analysis, and design phases of systems development. The last modeling activity, validation, is related to several phases.

If use cases are the primary tool for specifying system requirements, the use case extension activities given in Section 6.3 are a natural extension to the requirements specification phase. However, if use cases are not employed for describing requirements, the information in use cases should nevertheless be created for essential workloads. A possible technique is to use performance walkthroughs, as suggested by [Smi90], and document the results with extended use cases that already contain the additions required by the framework. After this, performance modeling can proceed as usual.

Our methodology assumes the existence of two separate performance models: the software performance model and the system performance model. This separation reflects the practice of many object-oriented methodologies to distinguish between analysis and design phases [Boo94, Rum91, Jac92, Jac98]. The analysis phase creates a conceptual model for the system so that its functionality can be described in a structured manner. The design phase in turn specifies an implementation-oriented model that contains enough technical details for constructing the system with the selected development tools and ready-made components.

The software performance model exploits the results of the *analysis phase* for producing meaningful performance estimates that can assist in validating and refining the analysis model. In some cases, however, intermediate software performance models can be created during the analysis phase to provide guidance in analysis phase decisions. The goals of our software performance model are close to those of the *software execution model* proposed by Smith [Smi90]. However, our approach yields more accurate results as we propose to use queuing networks already at this stage.

The system performance model in turn uses the results of the *design phase* for providing an in-depth understanding of the performance aspects of the system, and for yielding accurate predictions for the final implementation. However, it is also possible to build preliminary system performance models during the design phase in order to validate a particular design decision or experiment with different alternatives.

Performance model validation is related to several phases in systems development: instrumentation is part of implementation, performance testing is part of overall system testing, and calibration often precedes deployment. Validation may not be required for short-lived systems. However, long-lived systems are likely to become targets for capacity planning and, consequently, an accurate and validated performance model is needed for them. Also, a need may arise to revise the functionality of the system, and it may be beneficial to validate the performance model so that it can be used during the development of the new functionality. In some cases, it may be necessary to validate a performance model before the actual system exists. This can be carried out with a prototype implementation that represents performance sensitive elements of the system with sufficient accuracy.

6.8 Summary

In this chapter, we have presented a methodology for supporting the creation, refinement, and exploitation of performance models in the development of CORBA based distributed systems. The methodology defines a generic performance model structure that contains six layers for representing system from different points of view. The layers are: the application layer, the interface layer, the behavior layer, the infrastructure layer, the network layer, and the deployment layer.

The methodology specifies four activities that can be performed at various phases of systems development. The first activity extends use cases for performance modeling purposes, and it is intended to take place during requirements specification. The second activity produces the software performance model based on information that is obtained during the analysis phase. The third activity produces the system performance model based on the detailed information produced by the design phase. Finally, the fourth activity validates the model against the actual system or a prototype. Its main purpose is to support capacity planning and further development of the target system.

The main contribution of this chapter is to indicate how object-oriented software engineering can proceed in parallel with performance engineering practices. This involves resolving a fair amount of practical details, and we have covered these details from the viewpoint of our own performance modeling framework. The example in the next chapter illustrates some of the issues in more detail.

Chapter 7

Experimental results

In this chapter, we present experimental results that we have obtained from using the performance modeling framework. We first discuss our tool prototype and point out additional needs for a full performance modeling tool. Then, we present a case study where the framework has been used extensively. The goal of the case study is to illustrate the framework's expressive power and outline possible ways to use the framework's modeling techniques.

7.1 Tool prototype

Performance engineering of complex software systems cannot be carried out adequately without the support of suitable tools. In particular, tools are needed for constructing performance models, for solving them, and for presenting the results appropriately. Our prototype implementation for an object-oriented performance modeling and analysis tool (OAT) contains the following elements:

- A parser that reads PLM input files,
- An expander that generates AQN representations,
- A solver that solves the models with the MOD algorithm,
- An output module that presents the results.

The *parser* accepts one or more PML input files and creates an initial tree-like representation of the complete model. In the prototype implementation, the goal is to concentrate on performance related features of UML and, consequently, the tool only accepts PML input files. However, a full implementation of the OAT tool should also accept the XML Metamodel Interchange (XMI) format that has been specified as a means for exchanging UML models between tools [OMG98b]. Since XMI is not primarily intended to be readable by humans, this approach requires a modeling tool for producing XMI files. Such functionality is currently emerging in commercial modeling tools, but was not available at the time of starting the OAT prototype development.

The *expander* converts the initial representation into the AQN representation using the transformations described in Section 5.8. The model's static structure given by classes, interfaces, etc. is converted into a flat representation consisting of resource objects with the relevant properties, attributes, and operations attached to them. The model's behavioral aspects given by collaboration diagrams are combined into one complex collaboration diagram per workload.

The *solver* takes the AQN representation produced by the expander and uses the MOD algorithm for solving the model. During the execution of the MOD algorithm, the AQN model is decomposed into product-form queuing networks and, for each network, a choice is made between the exact MVA algorithm (Algorithm 1) and the Schweizer approximation (Algorithm 3) depending on the execution cost. Hence, models with small populations get more accurate treatment while models with large populations are solved in a sufficiently short time. This adjustment is compatible with the observation that the Schweizer approximation is more accurate for large populations [Agr85, Men94].

The *output module* displays metrics for the solved AQNs. To illustrate the tool's output, we solve the network monitor example from Section 5.9. The tool's report for the basic configuration with a single server node is shown in Figure 41. The first section of the report summarizes the utilization of resources. The second section displays the response times, throughputs and the number of jobs in the system for each workload. In addition, this section also indicates how the response time of each workload is divided between the resources. The last section gives a detailed report of the expanded AQN representation and the average times spent in each step. This section may get fairly complex since it contains all infrastructure-level interactions embedded into the application level workload diagrams. However, it also gives the possibility to verify the correctness of the AQN representation, and it allows additional metrics of interest to be derived.

An ideal output for the OAT tool would be a set of annotated UML diagrams where workload specific metrics, such as response times and throughputs, were embedded in behavioral diagrams and resource spe-

```
Utilization     Type    Device
-----------     ----    ------
0 %             Delay   MyLAN.Latency
53.3116 %       Queue   MyLAN.Server1.$CDatabase
67.7291 %       Queue   MyLAN.Server1.$CHandler
8.37806 %       Queue   MyLAN.Server1.$CReceiver
32.5002 %       Queue   MyLAN.Server1.Cpu
4.20001 %       Delay   MyLAN.Server1.Ctxswitch
40.0002 %       Queue   MyLAN.Server1.Disk

BgLoad
------
Resp.time:  132.338     Throughput: 0.001      Nbr.in system: 0.132338
Time share: 38.4263 %   MyLAN.Server1.$CDatabase
            10.3279 %   MyLAN.Server1.Cpu
            51.2458 %   MyLAN.Server1.Disk

Jobs
----
Resp.time:  314.887     Throughput: 0.007      Nbr.in system: 2.20421
Time share: 1.81783 %   MyLAN.Server1.$CDatabase
            62.4507 %   MyLAN.Server1.$CHandler
            0.347649 %  MyLAN.Server1.$CReceiver
            16.7913 %   MyLAN.Server1.Cpu
            1.90545 %   MyLAN.Server1.Ctxswitch
            16.6871 %   MyLAN.Server1.Disk

collaboration BgLoad {                  // Throughput: 0.001
  property arrivalrate = 0.001;         // Residence times
  1: GetActions() {d=0};                // 50.8526
  1.1: Cpu() {d=10};                    // 13.6677
  1.2: Disk() {d=50};                   // 67.8176
};                                      //*132.338

collaboration Jobs {                    // Throughput: 0.007
  property arrivalrate = 0.007;         // Residence times
  1: ReceiveJob() {d=0};                // 1.0947
  1.1: Cpu() {d=10};                    // 11.9684
  2: Ctxswitch() {d=2};                 // 2
  3: AcceptJob() {d=0};                 // 196.649
  3.1: Cpu() {d=5};                     // 6.96839
  3.2: Ctxswitch() {d=2};               // 2
  3.3: GetActions() {d=0};              // 5.72412
  3.3.1: Cpu() {d=10};                  // 11.9684
  3.3.2: Disk() {d=50};                 // 52.5454
  3.4: Ctxswitch() {d=2};               // 2
  3.5: DoActions() {d=0};               // 1.34865e-005
  3.5.1: Cpu() {d=20};                  // 21.9684
};                                      //*314.887
```

Figure 41. Example report from the OAT tool.

cific metrics, such as utilizations, were part of deployment diagrams (see also [Woo98]). If the output is produced in the XMI format, the OAT tool could be conveniently integrated with commercial graphical model-

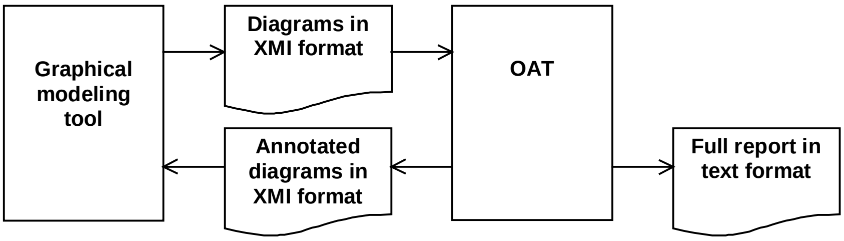


Figure 42. Tool integration through XMI files.

ing tools through standard XMI files. Figure 42 illustrates this possibility. The graphical modeling tool would be used for specifying the performance models and for representing the results, while the OAT tool would solve the models in the background.

An additional feature in the OAT tool could be support for performance requirements in UML diagrams. For example, a special property could be used for indicating an upper bound for the average response time for an interaction, and the tool could automatically produce a warning if this requirement is not met by a particular design.

7.2 Overview of the case study

The goal of this case study is to illustrate the use of the framework in practical performance modeling work. On one hand, it provides an example on how to model CORBA based applications and, on the other hand, it indicates how some elements of the CORBA infrastructure can be specified with the available modeling techniques. It is not our goal to create a complete model for the target system or to produce a full performance analysis for it.

The target system of our case study is a reduced version of the TPC Web Commerce Benchmark revision D-5.0 (TPC-W) as described in [TPC99]. We have slightly changed the original design that assumes HTTP communication between client and server nodes. In our approach, remote communication is implemented with CORBA, but the functionality of the system and all end user interactions are kept as close as possible to the original design. This way, we can retain the same primary metric of interest: system throughput in terms of end user interactions per second with a fixed number of clients. In addition, we show the average response times for the interactions and the utilizations of the server CPU.

To keep the case study sufficiently small, we omit the transaction processing part of the functionality and, hence, we assume zero service demand for the actual operations that model the application functionality. As a consequence, the performance model concentrates on the communication part of the system and on the activities that take place within the infrastructure. This way, we can better validate the framework's ability to model CORBA based communication and the CORBA infrastructure. Also, the selected approach gives interesting results on the products that were used for the case study. It is possible to extend the model with transaction processing and database management features by using traditional performance modeling techniques, such as those discussed in [Smi90] and [Men94].

Our prototype system was implemented with the Java language using the JBuilder 3.0 development environment and the Visibroker 3.4 CORBA platform from Inprise [Inp99]. The tests were carried out on two PCs that were connected with a dedicated 10 Mbs Ethernet LAN. Both PCs were running the Windows NT 4.0 operating system. The server functionality was assigned to a relatively slow 120 MHz PC. This way, the 400 MHz client PC was able to produce workloads that saturated the server. A third PC was connected to the network for gathering metrics on the other two PCs using the Windows NT performance monitor. This information proved to be particularly useful for understanding of the behavior of the infrastructure.

7.3 The application layer

The functionality of the application is defined in terms of 14 end user interactions that are summarized in Table 2. All interactions have the same overall structure that consists of three steps. First, a user initiates the interaction by requesting an operation based on the outcome of the previous interaction. Second, the client application carries out the requested action by making zero or more invocations to the server. Finally, the outcome of the interaction is displayed to the user who has now the opportunity to examine the results and to proceed to the next interaction.

The detailed descriptions of the interactions lead to eight application level classes for the system. The *CClient* class represents the client side application and it contains 14 operations for modeling the end user interactions. The *CShop* class provides a starting point for the client application when it starts looking for application objects at the server side. The

Table 2. End user interactions in the electronic commerce system.

Interaction	*Share %*	*Description*
Admin confirm	0.09	Confirm price change for a product
Admin request	0.10	Ask for a product price change
Best sellers	11.00	Show 50 best selling products
Buy confirm	0.69	Confirm the order made by the user
Buy request	0.75	Ask credit card and shipping information
Customer reg.	0.82	Ask user data to allow order processing
Home	16.00	Display the home page
New products	11.00	Show 50 newest products
Order display	0.25	Display the last order of the user
Order inquiry	0.30	Ask the user's identity to get his last order
Product detail	21.00	Display the details of a selected product
Search request	12.00	Ask product search criteria from the user
Search results	11.00	Display product search results
Shopping cart	2.00	Display the contents of the shopping cart

CProduct class represents products to be sold by the electronic commerce system. The *CCustomer* class represents end users that have chosen to purchase one or more products. The *CCart* class represents shopping carts that collect one or more product instances to be purchased by the customers. The *COrder* class represents orders that have been placed by customers The *CGate* class represents an interceptor that monitors user actions once he has identified himself in order to purchase the contents of a shopping cart. Finally, the *CLog* class represents a security log where every relevant operation request is registered. A class diagram with the above classes is shown in Figure 43. The associations in the diagram indicate the most important relationships between the classes.

7.4 The interface layer

To implement the electronic commerce system with CORBA, an explicit interface specification is needed. The application layer in the case study is relatively simple and, consequently, it is possible to use a one-to-one mapping between application classes and CORBA interfaces. Each class in Figure 43 is simply transformed into an interface. The resulting IDL file is listed in Appendix B.

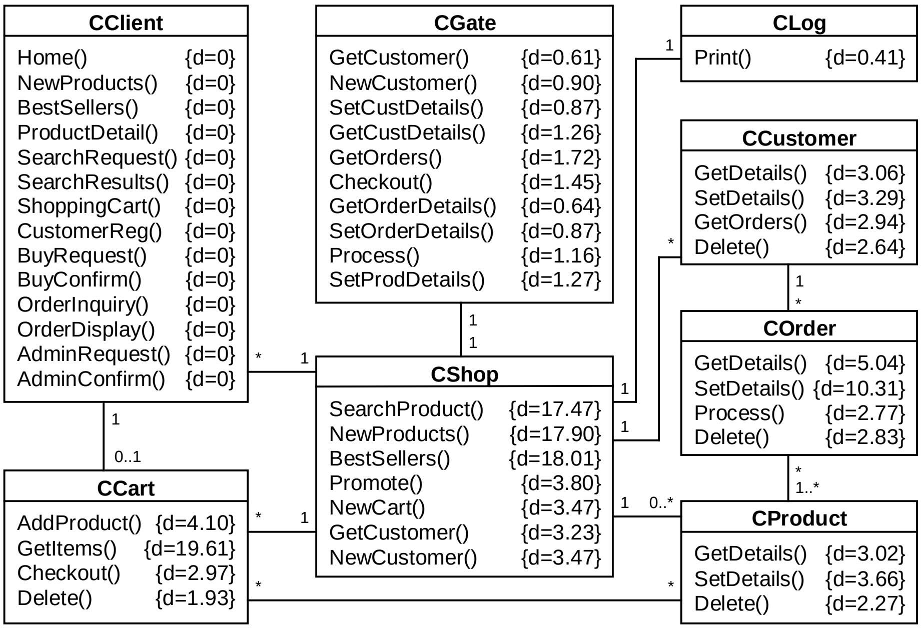


Figure 43. Class diagram for the electronic commerce system.

To obtain service demands for the operations in the IDL interfaces, we implemented a simple baseline benchmark with a single client and without any application logic. We used the code generation feature of the JBuilder tool to create the client and server applications. The server application consisted of CORBA object implementations representing each of the classes at the application layer. These object implementations are the actual application-level resources in the performance model. For each operation, we added parameters and return values that were similar to those proposed by the TPC-W specification. Also, the client application was instrumented to obtain response times for the operation invocations. Finally, we placed an adjustable think time between the invocations to experiment the effect of different think times.

The baseline test was conducted 14 times with think time values that ranged from 0 to 500 ms. We observed large variations in the baseline results. For example, the measurements for the *SearchProduct* operation ranged from 17.47 ms to 29.89 ms. Since the baseline test did not contain any application code, no disk activities were required, and there were no other applications executing in the background, we assumed that the high variation resulted from variable scheduling delays in the operating system, in the Java virtual machine, and in the CORBA implementation.

We decided to use the minimum of the results instead of the mean value. The idea was to eliminate the scheduling delays as much as possible, and to have a value that is fairly close to the actual execution time. While our measurement technique was relatively inaccurate, the alternative methods, such as instrumenting the operating system, would have been considerably more difficult to carry out. The obtained baseline results combine the network latency, the CPU time needed for parameter handling and other middleware activities at both ends, and the time spent in the network adapters. The results of the baseline measurements are shown in Figure 43 in milliseconds.

7.5 The behavior layer

The TPC-W specification defines the behavior of the client application with a transition probability matrix. It is used for selecting the next interaction after completing the previous one. We use the *shopping* matrix representing a shopping scenario at an electronic commerce site. Table 2 shows the resulting share percentages for end user interactions.

The think time between interactions is specified in the TPC-W document to have negative exponential distribution with the mean of 7 seconds. Since our implementation does not execute any application code, we executed our tests with the mean of 2 seconds to impose a higher workload for the server node. For normal TPC-W benchmarks, this value is used in the *overload run* for verifying the implementation's capability to withstand excessive workloads [TPC99].

Since each end user interaction contains only a few CORBA invocations, the behavior of the electronic commerce system can be modeled with one workload diagram. The first nine interactions are illustrated in Figure 44 and the last five are shown in Figure 45. The execution probability for each interaction is directly the share indicated by Table 2.

Several issues are worth noting in the workload diagram. First, we have simplified the diagram by omitting invocations that are irrelevant for the performance of the system. For example, new carts are created at most once during a user session and, consequently, calls to the *NewCart* operation have a minimal effect on performance. Such invocations can be included into the diagram without affecting the performance metrics by marking explicitly their execution probability to zero.

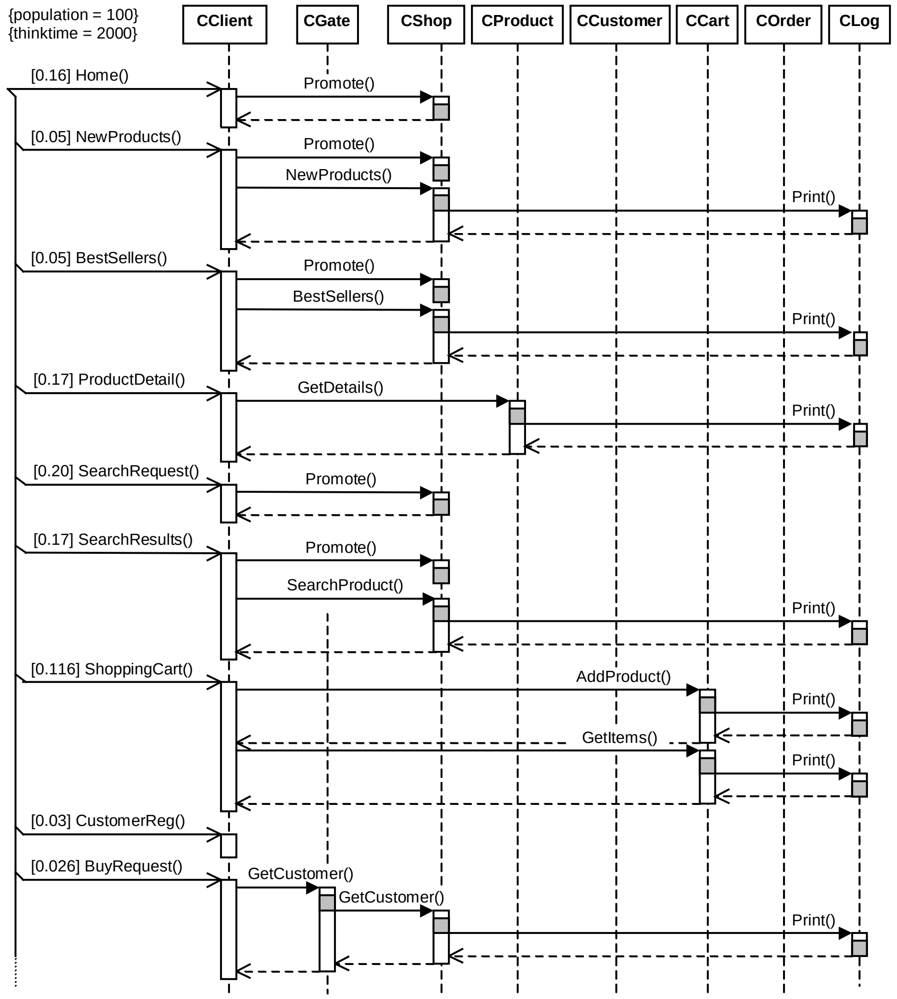


Figure 44. The first nine interactions of the behavior layer.

The second observation concerns interactions that make no requests to the server node (i.e. *CustomerReg*, *OrderInquiry*, and *AdminRequest*). These interactions allow the user to enter data before proceeding to some other interactions that actually involve server requests. The response times for these interactions are effectively zero, but they have been included into the model as they affect the primary metric of interest: the average number of user interactions per second.

The third observation concerns the usability of the model in its current state. We can produce a simple solvable model – the software perform-

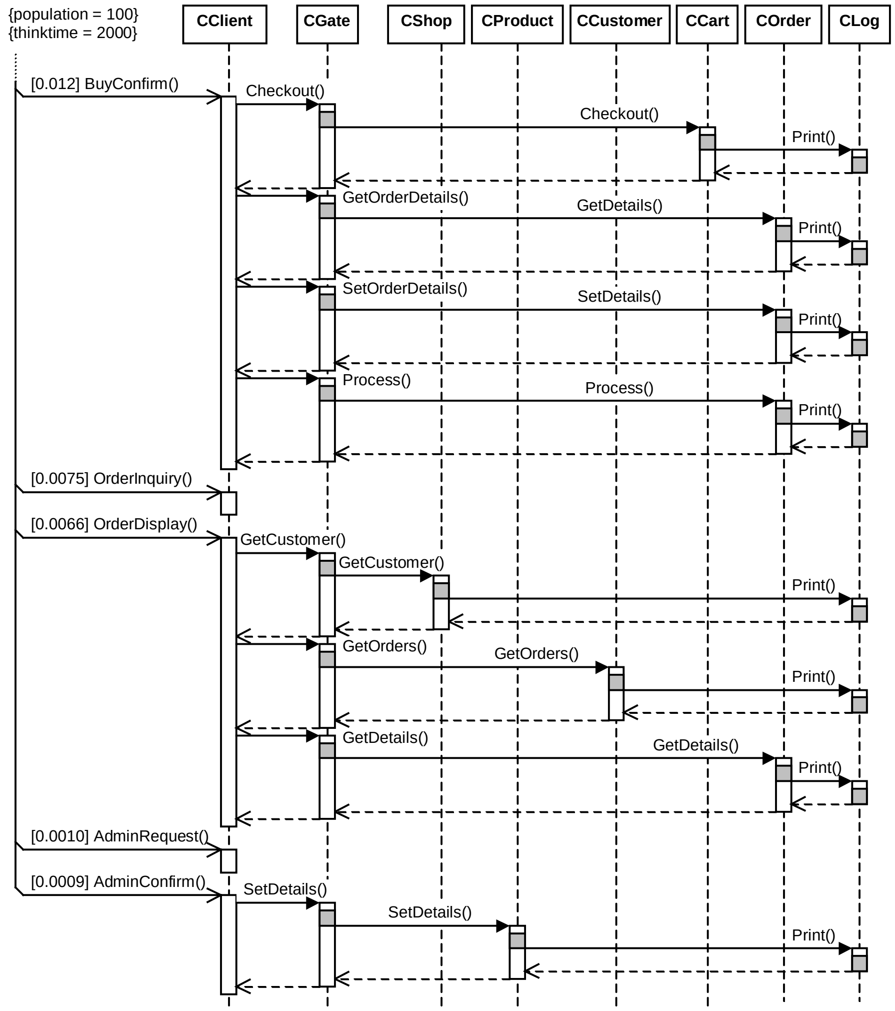


Figure 45. The last five interactions of the behavior layer.

ance model – by specifying an object diagram of eight objects. This model is illustrated in Figure 46. In this form, the model represents a system where each object implementation is running on a dedicated machine. In principle, it could be used for finding potential bottlenecks in the application logic. However, since there is no application logic in our reduced system, the obtained utilizations are highly optimistic and cannot be used for such purposes.

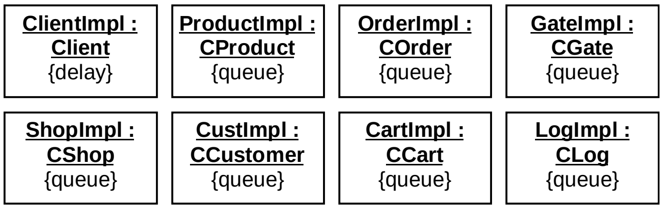

ClientImpl : Client {delay}
ProductImpl : CProduct {queue}
OrderImpl : COrder {queue}
GateImpl : CGate {queue}
ShopImpl : CShop {queue}
CustImpl : CCustomer {queue}
CartImpl : CCart {queue}
LogImpl : CLog {queue}

Figure 46. Trivial configuration for the electronic commerce system.

7.6 The network and infrastructure layers

The network and infrastructure layers are represented with a single set of diagrams due to the simplicity of the networking environment. To get more information for modeling the infrastructure layer, we carried out a series of experiments with the baseline application. During these experiments, we used the Windows NT performance monitor for obtaining general-purpose metrics for the server node: the CPU utilization, the number of threads in each server process, and the number of context switches per second. The experiments gave us the following information:

- The number of threads at the server node grows linearly with the number of concurrent clients,
- The number of context switches per time unit grows first linearly with the number of concurrent clients. When the server node gets congested and response times increase dramatically, the number of context switches per time unit drops significantly.
- The utilization of the CPU never exceeded 35% even when the server node seems to be congested and the response times are excessively high.

These observations suggest that context switching imposes a considerable queuing overhead for the server node (see also [Mic97]). Moreover, it is clear that the CPU is not the only resource involved. A likely cause for this phenomenon is the inappropriateness of PC-based hardware for frequent context switching requiring large amounts of data to be moved back and forth. Hence, we introduce a *MemBus* queuing device in each node for modeling additional queuing that is related to context switching. Finally, we include a *Cpu* queuing resource in each node.

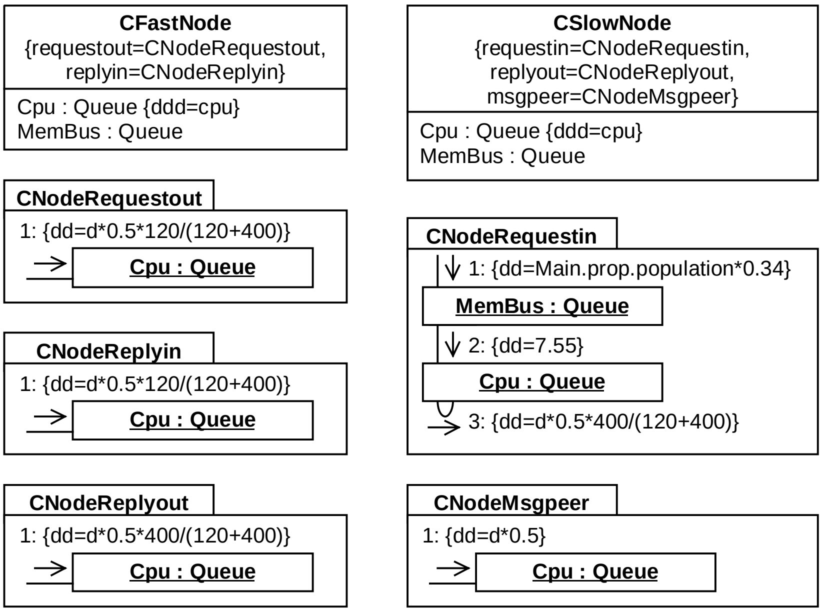


Figure 47. The infrastructure layer for the electronic commerce system.

To cope with different CPU speeds, we specify two separate classes, *CFastNode* and *CSlowNode*, for modeling the client and server nodes. In our simple example, the *CFastNode* class needs triggering properties for outgoing invocations and incoming replies, while the *CSlowNode* class requires triggering properties for incoming invocations, outgoing replies, and message passing between objects within the same node. The service demands that we obtained from the baseline test are normalized into machine cycles and divided evenly between the four triggering properties. This way, we get service demand estimates for activities that take place at both ends of the communication channel, such as the marshaling and demarshaling of parameters and the use of communication services. For example, the following service demand expression is obtained for the *Cpu* resource in the fast client node for outgoing requests:

```
dd = d * 0.5 * 120/(120+400)
```

Property *d* is the measured service demand from the baseline test, and property *dd* is the amount of service demand to be attributed to the *Cpu* resource during an outgoing request.

To complete the infrastructure layer, we need service demand estimates for invocation routing at the server node. These estimates are fairly difficult to come by since we have no direct access to the responsible system components, i.e. the operating system, the Java virtual machine, and the CORBA implementation. Therefore, we make some rough approximations based on our experience with the baseline application. First, we directly follow the observation indicating that the service demand for the *MemBus* resource increases linearly with the number of concurrent clients that directly corresponds to the number of threads in the server node. Hence, we use the factor *Main.prop.population* in the service demand expression for *MemBus*. Second, we assume that the service demand for the CPU resource is constant for each incoming request.

Finally, to obtain appropriate scaling factors for the service demand expressions, we carried out a series of experiments with a lightly loaded server (20 concurrent clients) using the baseline application. We were able to adjust the expressions so that the difference for the throughput, response time and CPU utilization were less than 5% between the computed values and the measurements. Figure 47 illustrates the resulting infrastructure layer. The adjusted service demand expressions for the *Cpu* and the *MemBus* resources are in the *CNodeRequestin* package.

7.7 The deployment layer

The deployment layer for the electronic commerce benchmark is illustrated in Figure 48. The system has two nodes. The client node contains a multi-threaded client application that simulates up to 100 simultaneous end users accessing the electronic commerce system. The server node contains five multi-threaded and two single-threaded CORBA object im-

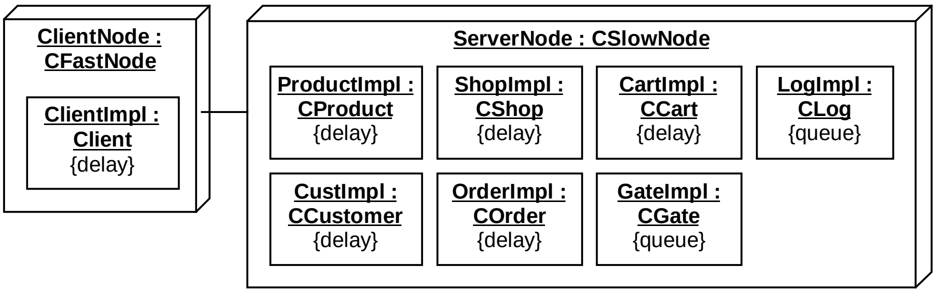


Figure 48. The deployment layer for the electronic commerce system.

plementations that are together responsible for the server side functionality. Appendix C contains the complete PML representation of the model shown in Figures 43 through 48.

7.8 Model validation

To validate the presented performance model, we conducted a series of measurements with our benchmark application. The application directly realizes the model described in Figures 43 through 48. To find the maximal throughput for the application, we let the number of concurrent clients vary between 10 and 100 with an increment of 10 clients.

The measured throughputs and the predictions from the model are illustrated in Figure 49. The average response times for end user interactions are shown in Figure 50, and the different utilizations for the server CPU are illustrated in Figure 51. The model is able to predict the throughput of the system reasonably well – the greatest difference between the measurement and the prediction is 18%. As for the two other metrics of interest, the relative error stays mostly under 25%.

While the results are fairly satisfactory, it should be borne in mind that the interactions in the system are relatively simple and similar to each other. Most of them are simpy reading or updating the data store. Therefore, it was a good choice to calibrate the model's infrastructure layer with a baseline application that was using similar simple interactions. However, the predictive power of such a simple baseline test may be sig-

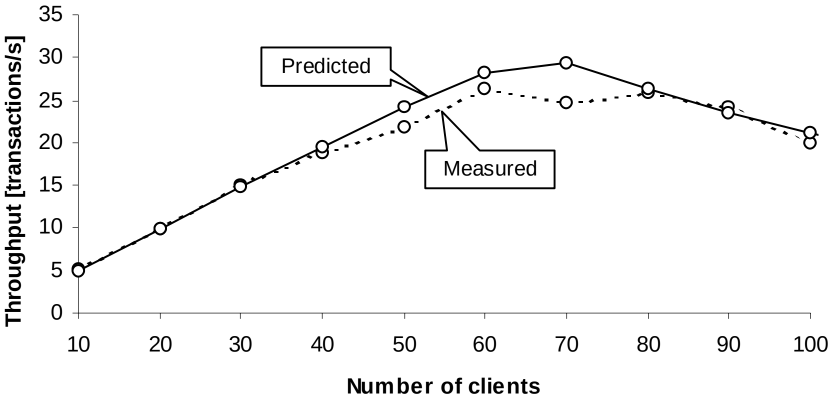


Figure 49. Throughputs for the electronic commerce system.

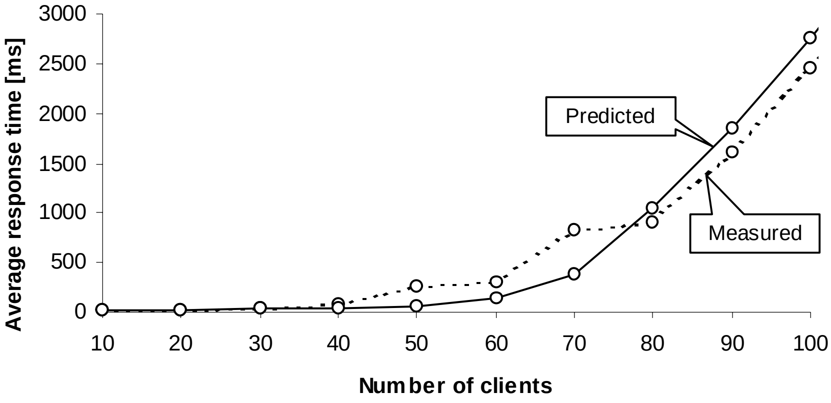


Figure 50. Average response times for the electronic commerce system.

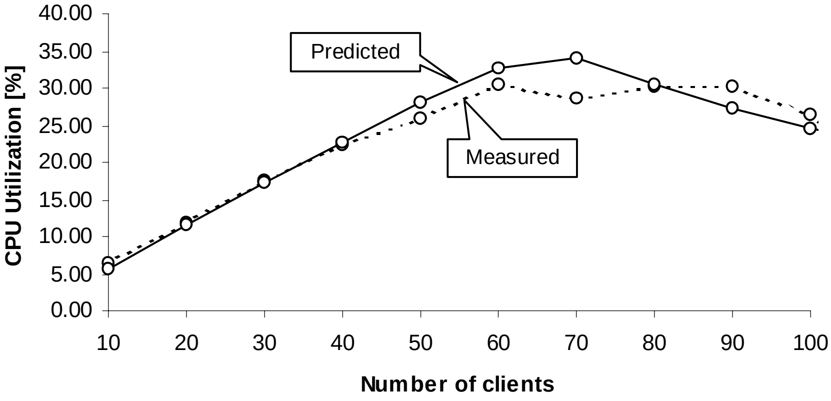


Figure 51. Server CPU utilization for the electronic commerce system.

nificantly inferior if the goal is to model a system that is based on more complex interactions. For example, if the goal is to use CORBA for supporting distributed technical calculations in a network, similar baseline tests would probably not be sufficient for achieving the same accuracy of results.

7.9 Discussion

Our case study is fairly limited in scope, but some observations are still in order. First, the proposed model for the electronic commerce system is relatively simple but it gives a reasonably good picture of the target system. This suggests that performance modeling is a feasible technique for CORBA based distributed systems in spite of the additional complexities caused by the middleware. The straightforward structure of the model further encourages software engineers to exploit performance modeling, since the models can be created and utilized without the explicit use of heavy mathematical machinery.

Second, the UML diagrams that we used for presenting the performance model are relatively easy to read. Performance related additions are given with simple properties attached to classes, operations, and diagrams. Due to the simplicity of the networking environment, it was not even necessary to use the «*connection*» stereotype for describing the structure of the network. This suggests that the expressive power of the UML is sufficient for performance modeling, and the use of special performance modeling notations is not necessary in normal software engineering.

Third, the usability of the infrastructure layer is worth noting. It provides a flexible tool for modeling the performance aspects of the middleware, the operating system, and the hardware in a structured manner with very little links to the application level. The complexity and accuracy of the infrastructure layer may range from a straightforward representation of communication delays to a full model for the middleware, operating system, and hardware. In our case, we used an abstract model where a hypothetical *MemBus* resource represented queuing in those system components that were beyond our access. The infrastructure layer is also a good target for model calibration, since changes at this layer do not distort the application level.

The fourth observation concerns the results for the target system. They indicate that queuing for software and hardware resources is an essential factor for the performance of CORBA based distributed systems, especially with a large number of simultaneous clients. Therefore, a simple analysis of communication delays is not enough for ensuring the responsiveness of an application in large distributed systems, such as those used in electronic commerce, telecommunications, and other similar domains.

7.10 Summary

In this chapter, we have discussed the practical aspect of our work. On one hand, we have presented our tool prototype and, on the other hand, we have described a case study illustrating the use of the modeling framework.

Our tool prototype allows the specification of performance models with PML, and this is sufficient for experimenting with the framework. However, additional functionality is needed for actual software engineering. In particular, graphical representation is needed both when creating the performance models and also when presenting the results. The emerging XMI specification may help to provide this functionality in a tool-independent way.

In the case study, we have created a performance model for a simple electronic commerce system. We have also implemented a prototype for the system and carried out measurements to validate the performance model. The model was constructed using the methodology proposed in Chapter 6, and it illustrates many of the modeling techniques presented in Chapter 5. If we consider the simplicity of the model, the obtained results were reasonably accurate – the relative error for throughput in the calibrated model was under 20% for the considered client populations. However, the most important result is the observation that our framework is sufficient for modeling CORBA based distributed systems and, furthermore, the models are relatively easy to create and use.

Chapter 8

Conclusions

In this work, we have presented a performance modeling framework for supporting the development and maintenance of CORBA based distributed systems. The framework contains five main elements. The first element is a UML based notation for describing performance models in a way that is familiar for software engineers. Performance related information is added to UML diagrams with standard UML extension features. The second element is a set of modeling techniques for constructing performance models for distributed systems based on the CORBA platform. These techniques allow models of complex distributed systems to be presented with a set of simple and understandable diagrams. The third element is an algorithm for solving the models for a number of relevant performance metrics, such as average response times, throughputs, utilizations, and queue lengths. The fourth element is a concise modeling methodology that defines how to build the actual models and how to use them in the context of object-oriented analysis and design methodologies. Its primary contribution is to offer means for enhancing object-oriented software engineering practices towards performance related issues. The fifth element is an experimental tool that can automate some of the tasks implied by the framework. In addition, we have illustrated the use of the framework with a case study.

In Section 1.2, we stated four generic requirements for the framework. The first requirement, automatic solvability, is satisfied due to the mapping from the UML based performance models into augmented queuing networks that can be solved with the method of decomposition. The proposed algorithm uses a number of approximations that may limit its usability. However, experiences with the framework indicate that the results of the algorithm are sufficiently accurate to be used in software performance engineering.

The second requirement, support for normal UML modeling style, is satisfied for most diagram types since the framework mainly operates through properties that can be attached to any UML element. However, to ensure the solvability of the performance models, we impose a number of limitations for them. For example, there is no support for spawning, synchronizing, and killing threads. Also, the use of triggering properties and related collaboration diagrams is a novel feature that slightly changes the appearance of UML based models for complex systems.

The third requirement, clear distinction between different architectural aspects of CORBA based distributed systems, is addressed by the layered structure of the UML representation. Six layers are proposed: the application, interface, behavior, infrastructure, network, and deployment layers. Only weak and non-restrictive links exist between modeling elements that belong to different layers. In particular, there is a clear distinction between elements that are part of the application logic and those belonging to the supporting infrastructure.

The fourth requirement, support for incremental development, is addressed by the performance modeling methodology that suggests to construct performance models on a layer-by-layer basis. Such models can be solved at all stages of systems development since some of the layers, such as the infrastructure and network layers, can be completely or partially omitted.

In Section 2.5, we stated four technical goals that are related to the architectural choices of the CORBA platform. The first goal, support for hidden interactions, is satisfied by triggering properties and related triggering diagrams. In some cases, service demand binding can also be used to serve this need. The second goal, support for flexible changes in configurations, has been achieved by defining a separate deployment layer that can be specified independent of any other layer in the model. The third goal, interface support, is directly implemented by the interface layer. The last goal, support for heterogeneity, is satisfied by the layered modeling style. In particular, high-level abstractions can be used at the application layer so that heterogeneity is only visible at the infrastructure layer. This way, the model for the application logic can be specified independent of any particular choice of hardware or system software. Also, the use of inheritance allows heterogeneous elements to be specified in a concise way. The example in Chapter 7 illustrates how these technical goals can be reached in practical modeling work.

While the framework meets the set requirements, a number of limitations and potential problems have also been observed. We briefly review some of the limitations, and point out possible ways of removing them. These limitations also lead to a number of interesting topics for further research.

First, the use of the MVA algorithm for solving queuing network models limits significantly the possible service time distributions, scheduling disciplines, and arrival rate distributions in the performance models. In addition, the obtained solutions do not indicate distributions for the metrics. These limitations can be removed by using simulation, but the cost of obtaining the results may become prohibitive for large systems. To alleviate this problem, special techniques can be used for controlling the length of the simulation run while still meeting the set precision requirements [Raa95]. Another alternative is to use analytic techniques for obtaining information on the resulting distributions [Raa89b, Con89]. A possible solution is to combine several techniques, so that rapid prototyping could be done with MVA based techniques while detailed results could be obtained from simulation and analytic techniques that yield information on the resulting distributions. It is also possible to extend the method of decomposition with various approximation techniques that support, for example, the modeling of job priorities and the use of non-exponential service demand distributions for the FCFS scheduling discipline.

The second limitation and an additional item for further work is the incomplete support for the UML in the current framework. On one hand, there are limitations on the way some UML diagram types can be used and, on the other hand, state and activity diagrams have been omitted from the framework. Additional work is needed to align the framework with the complete UML notation. It should be borne in mind, however, that the ongoing UML 2.0 initiative and the emerging new UML profiles may change the way performance-related elements are presented with UML. A possible extension is also the support for stating performance requirements in UML diagrams, so that a modeling tool could automatically detect performance violations in the system design.

An interesting area of further research is the addition of a graphical end user interface in our framework. On one hand, it could be used for creating performance models and, on the other hand, it could also help in visualizing the obtained performance metrics with annotated UML diagrams. One way to solve this problem is to provide an interface to an existing graphical tool through XMI files. However, it might also be useful to experiment with a dedicated performance-oriented graphical tool in order to fully exploit the possibilities of UML in performance modeling.

Finally, an important area of additional work is the extension of the framework outside its original scope, namely CORBA-based distributed systems and the UML. The proposed modeling techniques, perhaps with some modifications and extensions, can be applied to other types of information systems, such as those based on the COM platform. Since the proposed modeling techniques and the methodology are both based on object-oriented concepts, future extensions of this work could lead to a generic object-oriented performance modeling framework. Furthermore, the proposed modeling techniques are relatively close to some non-UML notations, such as message sequence charts defined by the ITU [ITU99] (see [Har99] for a discussion). Hence, our performance modeling framework could also be extended towards non-UML modeling techniques.

References

[Agr85] Agrawal, S.C., Metamodeling: A Study of Approximations in Queuing Models, MIT Press, Cambridge, MA, USA, 1985.

[Bar76] Barbour, A.D., Networks of Queues and the Method of Stages, Advances in Applied Probability 8, No. 3, 1976, 584-591.

[Bar79] Bard, Y., Some extensions to multiclass queuing network analysis, Performance of Computer Systems, Proceedings of the 4th International Symposium on Modelling and Performance Evaluation of Computer Systems, North Holland, Amsterdam, 1979, 51-61.

[Bas75] Baskett, F., Chandy, K.M., Muntz, R.R., Palacios, F.G., Open, closed, and mixed networks of queues with different classes of customers, Journal of the ACM, Vol. 22, No. 2, April 1975, 248-260.

[BEA99] BEA Systems, BEA WebLogic Enterprise Getting Started, Part Number 861-001001-003, Sunnyvale, CA, USA, 1999.

[Bey98] Beylot, A.-L., Becker, M., Performance analysis of multipath ATM switches under correlated and uncorrelated IBP traffic patterns, In: Körner, U., Nilsson, A. (Eds.), Performance of Information and Communication Systems, Proceedings of the Seventh IFIP TC 6/WG 6.3 International Conference of Information and Communication Systems, Chapman & Hall, Great Britain, 1998, 14-25.

[Boo94] Booch, G., Object-oriented Analysis and Design with Applications, 2nd edition, The Benjamin/Cummings Publishing Company, Redwood City, CA, USA, 1994.

[Cha75] Chandy, K.M., Herzog, U., Woo, L., Approximate Analysis of General Queuing Networks, IBM Journal of Research and Development, Vol. 19, No. 1, January 1975, 43-49.

[Cha82] Chandy, K.M., Neuse, D.N., Linearizer: A Heuristic Algorithm for Queuing Network Models of Computing Systems, Journal of the ACM, Vol. 25, No. 2, February 1982, 126-134.

[Con89] Conway, A.E., Georganas, N.D, Queuing Networks – Exact Computational Algorithms: A Unified Theory Based on Decomposition and Aggregation, The MIT Press, Cambridge, MA, USA, 1989.

[Cop96] Coplien, J., Schmidt, D. (Eds.), Pattern Languages of Program Design, Addison-Wesley, Reading, MA, USA, 1994.

[Cox55] Cox, D.R., A Use of Complex Probabilities in the Theory of Stochastic Processes, Proceedings of the Cambridge Philosophical Society 51, 1955, 313-319.

[Dau99] Daut, D., Yu, M., Performance Modeling of a Multistage Buffered ATM Switch with Bursty Traffic, In: Simon, R., Znati, T. (Eds.), Proceedings of the Communications Networks and Distributed Systems Modeling and Simulation Conference (CNDS'99), The Society for Computer Simulations International, San Diego, California, 1999, 165-172.

[DEC96] Digital Equipment Corporation, ObjectBroker – Designing and Building Applications, Maynard, MA, USA, May 1996.

[Den78] Denning, P.J., Buzen, J.P., The Operational Analysis of Queueing Network Models, Computing Surveys, Vol. 10, No. 3, 1978, 225-261.

[Dou99] Douglass, B.P., Real-Time UML, Second Edition, Addison-Wesley, Reading, MA, USA, 1999.

[Eid97] Eide, E., Frei, K., Ford, B., Lepreau, J., Lindstrom, G., Flick: A Flexible, Optimizing IDL Compiler, In: Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI'97), Las Vegas, Nevada, June 1997, 44-56.

[Els98] El-Sayed, H., Cameron, D., Woodside, M., Automated Performance Modeling from Scenarios and SDL Designs of

Telecom Systems, In: Proceedings of the International Symposium of Software Engineering for Parallel and Distributed Systems (PDSE98), Kyoto, April 1998.

[Eri98] Eriksson, H-E., Penker, M., UML Toolkit, John Wiley & Sons, New York, 1998.

[Flo89] Flowers, J., Dowdy, L.W., A comparison of calibration techniques for queuing network models, In: Proceedings of the 1989 CMG Conference, Reno, Nevada, December 1989, 644-655.

[Gam94] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: elements of reusable object-oriented software, Addison-Wesley, Reading, MA, USA, 1994.

[Gok96] Gokhale, A., Schmidt, D., Measuring the Performance of Communication Middleware on High-Speed Networks, In: Proceedings of the SIGCOMM Conference, Computer Communication Review, Vol. 26, No. 4, October 1996, 306-317.

[Gok97] Gokhale, A., Schmidt, D., Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA, Proceedings of GLOBEGOM'97, Phoenix, Arizona, November 1997.

[Gok98a] Gokhale, A., Schmidt, D., Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks, IEEE Transactions on Computers, Vol. 47, No.4, April 1998, 391-413.

[Gok98b] Gokhale, A., Schmidt, D., Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance, In: Proceedings of the 31th Hawaii International Conference on System Sciences, Volume VII, Hawaii, USA, 1998.

[Gra91] Gray, Jim (Ed.), The Benchmark Handbook for Database and Transaction Processing Systems, Morgan Kaufmann Publishers, San Mateo, CA, USA, 1991.

[Har99] Harel, D., On the Behavior of Complex Object-Oriented Systems, In: France, R., Rumpe, B. (Eds.), «UML»'99 – The Unified Modeling Language, Beyond the Standard, LNCS 1723, Springer-Verlag, Berlin, Germany, 1999, 324-329.

[Hav98] Haverkort, B.R., Performance of computer communication systems: a model-based approach, John Wiley & Sons, New York, NY, USA, 1998.

[Hei83] Heidelberger, P., Triverdi, K.S., Analytical Queuing Models for Programs with Internal Concurrency, IEEE Transactions on Computers, Vol. C-32, No. 1, January 1983, 73-82.

[Hei84] Heidelberger, P., Lavenberg, S., Computer performance methodology, IEEE Transactions on Computers, Vol. C-33, No. 12, December 1984, 1195-1220.

[Hel96] Hellemans, P., Steegmans, F., Vanderstraeten, H., Zuidweg, H., Implementation of Hidden Concurrency in CORBA Clients, In: Spaniol, O., Linnhoff-Popien, C., Meyer, B., Trends in Distributed Systems, CORBA and Beyond, LNCS 1161, Springer-Verlag, Aachen, Germany, October 1996, 30-42.

[Inp99] Inprise Corporation, JBuilder 3.0 Documentation, Scotts Valley, CA, USA, 1999.

[Ion97] IONA Technologies PLC, Orbix 2.3 Documentation, Dublin, Ireland, 1997.

[ISO95] ISO/IEC, Open Distributed Processing – Reference Model – Part 3: Architecture, International Standard IS10746-3, 1995.

[ITU99] International Telecommunication Union, ITU-T Recommendation Z.120, Message Sequence Chart (MSC), 1999.

[Jac82] Jacobson, P.A., Lazowska, E.D., Analyzing Queuing Networks with Simultaneous Resource Possession, Communications of the ACM, Vol. 25, No. 2, February 1982, 142-151.

[Jac92] Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G., Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, Harlow, England, 1992.

[Jac98] Jacobson, I., Booch, G., Rumbaugh, J., The Unified Software Development Process, Addison-Wesley, Reading, MA, USA, 1998.

[Jai91] Jain, R., The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, John Wiley & Sons, New York, NY, USA, 1991.

[Kel76] Kelly, F.P., Networks of queues, Advances in Applied Probability 8, No. 2, 1976, 416-432.

[Käh98a] Kähkipuro, P., A survey of techniques and guidelines for improving the performance of CORBA-based distributed systems, In: Ilkka Niemelä (Ed.), Proceedings of the HeCSE Workshop on Emerging Technologies in Distributed Systems, Research Report A 50, Digital Systems Laboratory, Helsinki University of Technology, 1998, 23-32.

[Käh98b] Kähkipuro, P., Object-Oriented Middleware for Distributed Systems, Licentiate Thesis, Report C-1998-43, Department of Computer Science, University of Helsinki, Finland, 1998.

[Käh99a] Kähkipuro, P., The Method of Decomposition for Analyzing Queuing Networks with Simultaneous Resource Possessions, In: Simon, R., Znati, T. (Eds.), Proceedings of the Communications Networks and Distributed Systems Modeling and Simulation Conference (CNDS'99), The Society for Computer Simulations International, San Diego, California, 1999.

[Käh99b] Kähkipuro, P., UML Based Performance Modeling Framework for Object-Oriented Distributed Systems, In: France, R., Rumpe, B. (Eds.), «UML»'99 – The Unified Modeling Language, Beyond the Standard, LNCS 1723, Springer-Verlag, Berlin, Germany, 1999, 356-371.

[Käh99c] Kähkipuro, P., Reference Guide for the Performance Modeling Language, The CORBA-FORTE Project, Department of Computer Science, University of Helsinki, 1999.

[Lit61] Little, J.D.C., A proof of the queuing formula: $L = \lambda W$, Operations Research, Vol. 9, 1961, 383-387.

[Lit98] Litoiu, M., Rolia, J., Serazzi, G., Designing Process Replication and Threading Policies: A Quantitative Approach, In: Puigjaner, R., Savino, N., Sera, B. (Eds.), 10th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (Tools'98), LNCS 1469, Springer-Verlag, Berlin, Germany, 1998, 15-26.

[Maf97] Maffeis, S., Schmidt, D., Constructing Reliable Distributed Communication Systems with CORBA, IEEE Communications Magazine, Vol. 35, No. 2, February 1997, 56-60.

[Mar94] Martinka, J., Friedrich, R., Sienknecht, T., Murky Transparencies: Clarity using Performance Engineering, In: Raymond, K., and Armstrong, L. (Eds.), Open Distributed Processing – Experiences with distributed environments, Proceedings of the third IFIP TC 6/WG 6.1 international conference on open distributed processing, Chapman & Hall, Great Britain, 1994, 507-510.

[Men94] Menascé, D.A., Almeida, V.A.F., Dowdy, L.W., Capacity Planning and Performance Modeling, Prentice Hall, Englewood Cliffs, NJ, USA, 1994.

[Mes94] Meszaros, Gerard, Pattern: Half-object + Protocol (HOPP), In: Coplien, J., Schmidt, D. (Eds.) Pattern Languages of Program Design, Addison-Wesley, Reading, MA, USA, 1994.

[Mic97] Microsoft Corporation, Microsoft Visual C++ Version 5.0 Professional Edition, Redmond, WA, USA, 1997.

[Mic98] Microsoft Corporation, Microsoft Component Services, Server Operating System – A Technical Overview, Redmond, WA, USA, 1998.

[MLC98] MLC Systeme, Charles University, CORBA Comparison Project – Final Project Report, MLC Systeme GmbH, Rattingen, Germany, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic, 1998.

[Mow97] Mowbray, T., Malveau, R., CORBA Design Patterns, John Wiley & Sons, Inc., New York, USA, 1997.

[OMG92] Object Management Group, Soley, R.M. (Ed.), Object Management Architecture Guide, OMG TC Document 92.11.1, Revision 2.0, John Wiley & Sons, New York, USA, 1992.

[OMG98a] Object Management Group, CORBAservices: Common Object Services Specification, OMG TC Document formal/98-12-09, Framingham, MA, USA, 1998.

[OMG98b] Object Management Group, XML Metadata Interchange (XMI), OMG TC Document ad/98-10-05, Framingham, MA, USA, 1998.

[OMG98c] Object Management Group, White Paper on Wireless Access and Terminal Mobility in CORBA, OMG TC Document telecom/98-11-09, Framingham, MA, USA, 1999.

[OMG99a] Object Management Group, White Paper on Benchmarking, Version 1.0, OMG TC Document bench/99-12-01, Framingham, MA, USA, 1999.

[OMG99b] Object Management Group, A Human-Usable Textual Notation for the UML Profile for EDOC, Request for Proposal, OMG TC Document ad/99-03-12, Framingham, MA, USA, 1999.

[OMG99c] Object Management Group, Portable Interceptors, Joint Revised Submission, OMG TC Document orbos/99-12-02, Framingham, MA, USA, 1999.

[OMG99d] Object Management Group, The Common Object Request Broker: Architecture and Specification, Minor revision 2.3.1, OMG TC Document formal/99-10-07, Framingham, MA, USA, 1999.

[OMG99e] Object Management Group, Notification Service, OMG TC Document telecom/99-07-01, Framingham, MA, USA, 1999.

[OMG99f] Object Management Group, CORBA Messaging, OMG TC Document orbos/98-05-05, Framingham, MA, USA, 1999.

[OMG99g] Object Management Group, Real-Time CORBA, OMG TC Document orbos/99-02-12, Framingham, MA, USA, 1999.

[OMG99h] Object Management Group, CORBA Components – Volume I, OMG TC Document orbos/99-07-01, Framingham, MA, USA, 1999.

[OMG99i] Object Management Group, UML Profile for Scheduling, Performance, and Time, Request for Proposal, OMG TC Document ad/99-03-13, Framingham, MA, USA, 1999.

[OMG99j] Object Management Group, UML 2.0 Request for Information, Version 1.0, OMG TC Document ad/99-08-08, Framingham, MA, USA, 1999.

[Pet99] Petriu, D.C., Wang, X., From UML description of high-level software architecture to LQN performance models, In: International Workshop on Applications of Graph Transformation with Industrial Relevance, AGTIVE'99, Monastery Rolduc, Kerkrade, The Netherlands, September 1-3, 1999, to appear in Lecture Notes in Computer Science, Springer, 2000.

[Poo99] Pooley, R., King, P., The unified modeling language and performance engineering, IEE Proceedings Software, Vol. 146, No. 1, February 1999, 2-10.

[Raa89a] Raatikainen, K.E.E., Modelling and Analysis Techniques for Capacity Planning, Ph.D. Thesis, Report A-1989-6, Department of Computer Science, University of Helsinki, 1989.

[Raa89b] Raatikainen, K.E.E., Approximating Response Time Distributions, In: ACM SIGMETRICS and Performance 89 International Conference on Measurement and Modeling of Computer Systems, Performance evaluation review, Vol. 17, No. 1, 1989, 190-199.

[Raa95] Raatikainen, K.E.E., Simulation-Based Estimation of Proportions, Management Science, Vol. 41, No. 4, July 1995, 1202-1223.

[Ram98] Ramesh, S., Perros, H.G., A Multi-Layer Client-Server Queueing Network Model with Synchronous and Asynchronous Messages, In: Proceedings of the First International Workshop on Software and Performance WOSP 98, ACM, New York, NY, USA, 1998, 107-119.

[Rat97] Rational Software Corporation, Unified Modeling Language Documentation, version 1.1, Cupertino, CA, USA, 1997.

[Rei80] Reiser, M., Lavenberg, S., Mean-Value Analysis of Closed Multichain Queuing Networks, Journal of the ACM, Vol. 27, No. 2, April 1980, 313-322.

[Rev96] Reverbel, F., Persistence in Distributed Object Systems: ORB/ODBMS Integration, Ph.D. Dissertation, University of New Mexico, USA, 1996.

[Rol92] Rolia, J.A., Predicting the Performance of Software Systems, Ph.D. Thesis, Technical Report CSRI-260, Computer Systems Research Institute, University of Toronto, Canada, 1992.

[Rol95] Rolia, J.A., Sevcik, K.C., The Method of Layers, IEEE Transactions on Software Engineering, Vol. 21, No. 8, August 1995, 689-699.

[Rum91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ, USA, 1991.

[Rum99] Rumbaugh, J., Jacobson, I., Booch, G., The Unified Modeling Language Reference Manual, Addison-Wesley, Reading, MA, USA, 1999.

[Sch97] Schmidt, D., Gokhale, A., Harrison, T., Parulkar, G., A High-Performance End System Architecture for Real-Time CORBA, IEEE Communications Magazine, Vol. 35, No. 2, February 1997, 72-77.

[Sch98a] Schmidt, D., Evaluating Architectures for Multi-threaded CORBA Object Request Brokers, Communications of the ACM, Vol. 41, No. 10, October 1998, 54-60.

[Sch98b] Schmidt, D., Levine, D., Mungee, S., The design of the TAO real-time object request broker, Computer Communications, Vol. 21, No.4, 1998, 294-324.

[Sev77] Sevcik, K., Priority Scheduling Disciplines in Queuing Network Models of Computer Systems, In: Proceedings of IFIP Congress 77, North-Holland, Amsterdam 1977, 565-570.

[Sev81] Sevcik, K., Mitrani, I., The distribution of queuing network states at input and output instants, Journal of the ACM, Vol. 28, No. 2, April 1981, 358-371.

[Sho98] Shousha, C., Petriu, D., Jalnapurkar, A., Ngo, K., Applying Performance Modeling to a Telecommunication System, In: Proceedings of the First International Workshop on Software and Performance WOSP 98, ACM, New York, NY, USA, 1998, 1-6.

[Shu77] Shum, A.W., Buzen, J.P., The EFP Technique: A Method for Obtaining Approximate Solutions to Closed Queuing Networks with General Service Times, In: Beilner H., Gelenbe E. (Eds.), Third International Symposium on Measuring, modelling and evaluating computer systems, Bonn-Bad Godesberg, North Holland, Amsterdam 1977, 201-220.

[Sil90] de Souza e Silva, E., Munz, R.R., A Note on the Computational Cost of the Linearizer Algorithm for Queuing Networks, IEEE Transactions on Computers, Vol. 39, No. 6, June 1990, 840-842.

[Sla99] Slama, D., Garbis, J., Russell, P., Enterprise CORBA, Prentice Hall, Upper Saddle River, NJ, USA, 1999.

[Smi90] Smith, C.U, Performance Engineering of Software Systems, Addison-Wesley, Reading, MA, USA, 1990.

[Smi97] Smith, C.U., Williams, L.G., Performance engineering evaluation of object-oriented systems with SPE.ED, In: Marie, R., Plateau, B., Calzarossa, M., Rubino, G. (Eds.), Computer Performance Evaluation – Modeling Techniques and Tools, LNCS 1245, Springer-Verlag, Berlin, Germany, 1997, 135-154.

[Sou98] D'Souza, D.F., Wills, A.C., Objects, Components, and Frameworks with UML: the Catalysis Approach, Addison-Wesley, Reading, MA, 1998.

[Ste98] Steppler, M., Performance Analysis of Communications Systems Formally Specified in SDL, In: Proceedings of the First International Workshop on Software and Performance WOSP 98, ACM, New York, NY, USA, 1998, 49-62.

[Sun99] Sun Microsystems, Java Development Kit Documentation, JDK 1.2, Palo Alto, CA, USA, 1999.

[TPC99] Transaction Processing Performance Council (TPC), TPC Benchmark W (Web Commerce), Revision D-5.0, TPC, San Jose, CA, USA, 1999.

[Utt97] Utton, P., Hill, B., Performance Prediction: An Industry Perspective, In: Marie, R., Plateau, B., Calzarossa, M., Rubino, G. (Eds.), Computer Performance Evaluation – Modeling Techniques and Tools, LNCS 1245, Springer-Verlag, Berlin, Germany, 1997, 1-5.

[Wat97] Waters, G., Linington, P., Akehurst, D., Symes, A., Communications software performance prediction, In: Kouvatsos, D. (Ed.), 13th UK Workshop on Performance Engineering of Computer and Telecommunication Systems, Ilkley, West

Yorkshire, BCS Performance Engineering Specialist Group, 1997, 38/1-38/9.

[Woo95] Woodside, C.M., Neilson, J.E., Petriu, D.C., Majumdar, S., The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software, IEEE Transactions of Computers, Vol. 44, No. 1, January 1995, 20-34.

[Woo98] Woodside, M., Hrischuk, C., Selic, B., Bayarov, S., A Wideband Approach to Integrating Performance Prediction into a Software Design Environment, In: Proceedings of the First International Workshop on Software and Performance (WOSP'98), ACM, New York, NY, USA, 1998.

[Zie96] Zielinski, K., Uszok, A., Steinder, M., Crossing Technological Domains Using the Inter-ORB Request Level Bridge – Preliminary Performance Study, In: Spaniol, O., Linnhoff-Popien, C., Meyer, B., Trends in Distributed Systems, CORBA and Beyond, LNCS 1161, Springer-Verlag, Aachen, Germany, October 1996, 148-161.

Appendix A

Abstract grammar for the PML notation

The following symbols are used in the definitions of the PML grammar:

*	The preceding element can appear zero or more times
+	The preceding element must appear at least once
( )	Parentheses are used for grouping
[ ]	Optional elements are enclosed in square brackets
|	Vertical bars separate alternative elements
" "	Quotes enclose terminals that must appear exactly as written
...	A long list may be abbreviated with ellipsis.

specification ::=	package-body
package ::=	"package" package-name "{" package-body "}" ";"
package-body ::=	(package | class | collaboration | instance | property)*
package-name ::=	identifier
class ::=	class-keyword class-name "{" class-body "}" ";"
class-body ::=	property* realization* inheritance* attribute* operation*
class-keyword ::=	"class" | "actor" | "node" | "connection" | "interface"
class-name ::=	identifier
property ::=	"property" tagged-value ("," tagged-value)* ";"
realization ::=	"realizes" interface-name ("," interface-name)* ";"

```
interface-name ::=   identifier
inheritance ::=      "inherits" class-name  ("," class-name)* ";"
attribute ::=        attribute-name ["[" multiplicity "]"] ":" class-name
                     [properties] ";"
attribute-name ::=   identifier
multiplicity ::=     intconst
properties ::=       "{" [tagged-value ("," tagged-value)*] "}"
operation ::=        operation-name "(" ")" [properties] ";"
operation-name ::=   identifier
collaboration ::=    "collaboration" coll-name "{" coll-body "}" ";"
coll-body ::=        property* message*
coll-name ::=        identifier
message ::=          ["[" multiplier "]"] sequence ":" target-name "(" ")"
                     [properties] ";"
multiplier ::=       Floatconst
sequence ::=         intconst ("." intconst)*
target-name ::=      identifier ("." identifier)*
instance ::=         [instance-name] ["[" multiplicity "]"] ":" class-name
                     ["{"instance-body"}"] ";"
instance-body ::=    property* (instance-name ";" | instance)*
instance-name ::=    identifier
tagged-value ::=     tag-name ["=" value]
tag-name ::=         identifier
value ::=            expression | identifier
expression ::=       term (("+" | "-") term)*
term ::=             factor (("*" | "/") factor)*
factor ::=           "(" expression ")" | floatconst | identifier
intconst ::=         digit+
floatconst ::=      [sign] digit* ["." digit+] [("e" | "E") [sign] digit+]
sign ::=            "+" | "-"
identifier ::=      letter (letter | digit)*
letter ::=          "a" | ... | "z" | "A" | ... | "Z" | "_"
digit ::=           "0" | ... | "9"
```

Appendix B

IDL specification for the electronic commerce system

```
module Ecom {
  interface Shop;
  interface Product;
  interface Customer;
  interface Cart;
  interface Order;
  interface Log;
  interface Gate;

  // Exceptions for order processing
  exception AlreadyProcessed{};
  exception InvalidOrder{};
  exception NoItemsInCart{};

  // Search result
  struct SearchResult {
    string author;        // Author name
    string title;         // Product title
    Product item;         // Actual product
  };
  typedef sequence<SearchResult> SearchResultList;

  // List of items in a cart/order
  struct Item {
    Product item;         // Actual product
    string title;         // Product title
    long cost;            // Price in cents
    long srp;             // List price in cents
    string backing;       // Paper or hardback
    long qty;             // Quantity
    string comment;       // Comment
  };
```

```
typedef sequence<Item> ItemList;

// Product details
struct ProductDetails {
  long productid;       // Product id
  string title;         // Product title
  string fname;         // Author’s first name
  string lname;         // Author’s last name
  string pubdate;       // Publication date
  string publisher;     // Published
  string subject;       // Product’s subject
  string desc;          // Description
  long cost;            // Price in cents
  long srp;             // List price in cents
  string avail;         // Availability date
  string isbn;          // ISBN number
  long pages;           // Number of pages
  string backing;       // Paper or hardback
  string dimension;     // Book dimensions
};

// Customer details
struct CustomerDetails {
  long customerid;      // Customer id
  string fname;         // First name
  string lname;         // Last name
  string street1;       // Street
  string street2;       // Street
  string city;          // City
  string state;         // State
  string zip;           // Zip code
  string country;       // Country
  string phone;         // Phone number
  string email;         // Email address
  string birthdate;     // Birthdate
  string data;          // Notes
};

// Order details
struct OrderDetails {
  long orderid;         // Order id
  string orderdate;     // Date of order
  string shipdate;      // Date of shipment
  string status;        // Shipment status
  Customer buyer;       // Customer
  ItemList items;       // Items and quantities
  string shipping;      // Shipping method
  long subtotal;        // Sum of prices
  long tax;              // Tax
  long shipcost;         // Shipping and handling
  long total;            // Total
  string cctype;         // Credit card type
  string ccname;         // Credit card name
  string ccnumber;       // Credit card number
  string ccexpiry;       // Credit card expiry
  string ship_street1;   // Shipping street
  string ship_street2;   // Shipping street
  string ship_city;      // Shipping city
  string ship_state;     // Shipping state
  string ship_zip;       // Shipping zip code
  string ship_country;   // Shipping country
};

// List of orders
typedef sequence<Order> OrderList;

// Manages multiple shops
interface ShopManager {
  Shop GetShop();
  long Register(in Shop s);
  void Unregister(in long shopkey);
};

// Home interface for a shop
interface Shop {
  void SearchProduct(in string type,
    in string key,
    out SearchResultList srl);
  void NewProducts(in string subject,
    out SearchResultList srl);
  void BestSellers(in string subject,
    out SearchResultList srl);
  void Promote(in Product item,
    out SearchResultList srl);
  Cart NewCart();
  Customer GetCustomer(in string uname,
    in string passwd);
  Customer NewCustomer();
  Log GetLog();
  Gate GetGate();
};

// Product
interface Product {
  ProductDetails GetDetails();
  void SetDetails(in ProductDetails dets);
  void Delete();
};

// Customer
interface Customer {
  CustomerDetails GetDetails();
  void SetDetails(in CustomerDetails dets);
  void GetOrders(out OrderList ol);
  void Delete();
};

// Shopping cart
interface Cart {
  void AddProduct(in long qty, in Product item);
  void GetItems(out ItemList items);
  Order Checkout()
    raises (NoItemsInCart);
  void Delete();
};

// Order
interface Order {
  OrderDetails GetDetails();
  void SetDetails(in OrderDetails dets)
    raises (AlreadyProcessed);
  void Process()
    raises (AlreadyProcessed,InvalidOrder);
  void Delete()
    raises (AlreadyProcessed);
};

// Log
interface Log {
  void Print(in string line);
};

// Gatekeeper for secure actions
interface Gate {
  // Shop operations
  Customer GetCustomer(in Shop sh, in string uname,
    in string passwd);
  Customer NewCustomer(in Shop sh);
  // Customer operations
  void SetCustomerDetails(in Customer cust,
    in CustomerDetails dets);
  CustomerDetails GetCustomerDetails(in Customer cust);
  void GetOrders(in Customer cust, out OrderList ol);
  // Cart operations
```

```
    Order Checkout(in Cart crt) raises (NoItemsInCart);
    // Order operations
    OrderDetails GetOrderDetails(in Order ord);
    void SetOrderDetails(in Order ord,
      in OrderDetails dets)
      raises (AlreadyProcessed);
    void Process(in Order ord)
      raises (AlreadyProcessed,InvalidOrder);
    // Product
    void SetProductDetails(in Product prod,
      in ProductDetails dets);
  };
};
```

Appendix C

PML specification for the electronic commerce system

```
// Application and interface layers
class CClient {
  property delay;
  Home()              {d = 0};
  NewProducts()       {d = 0};
  BestSellers()       {d = 0};
  ProductDetail()     {d = 0};
  SearchRequest()     {d = 0};
  SearchResults()     {d = 0};
  ShoppingCart()      {d = 0};
  CustomerReg()       {d = 0};
  BuyRequest()        {d = 0};
  BuyConfirm()        {d = 0};
  OrderInquiry()      {d = 0};
  OrderDisplay()      {d = 0};
  AdminRequest()      {d = 0};
  AdminConfirm()      {d = 0};
};

class CShop {
  property delay;
  SearchProduct()     {d = 17.47};
  NewProducts()       {d = 17.90};
  BestSellers()       {d = 18.01};
  Promote()           {d = 3.80};
  NewCart()           {d = 3.47};
  GetCustomer()       {d = 3.23};
  NewCustomer()       {d = 3.47};
};

class CProduct {
  property delay;
  GetDetails()        {d = 3.02};
  SetDetails()        {d = 3.66};
  Delete()            {d = 2.27};
};

class CCustomer {
  property delay;
  GetDetails()        {d = 3.06};
  SetDetails()        {d = 3.29};
  GetOrders()         {d = 2.94};
  Delete()            {d = 2.64};
};

class CCart {
  property delay;
  AddProduct()        {d = 4.10};
  GetItems()          {d = 19.61};
  Checkout()          {d = 2.97};
  Delete()            {d = 1.93};
};

class COrder {
  property delay;
  GetDetails()        {d = 5.04};
  SetDetails()        {d = 10.31};
  Process()           {d = 2.77};
  Delete()            {d = 2.83};
};

class CGate {
  property queue;
  GetCustomer()       {d = 0.61};
  NewCustomer()       {d = 0.90};
  SetCustDetails()    {d = 0.87};
  GetCustDetails()    {d = 1.26};
  GetOrders()         {d = 1.72};
  Checkout()          {d = 1.45};
  GetOrderDetails()   {d = 0.64};
  SerOrderDetails()   {d = 0.87};
  Process()           {d = 1.16};
  SetProdDetails()    {d = 1.27};
};
```

```
class CLog {
  property queue;
  Print()            {d = 0.41};
};

// Behavior layer
collaboration Main {
  property population = 120;
  property thinktime = 2000;
  // Home interaction
  [0.1600] 1: CClient.Home();
           1.1: CShop.Promote();
  // New Products interaction
  [0.0500] 2: CClient.NewProducts();
           2.1: CShop.Promote();
           2.2: CShop.NewProducts();
           2.2.1: CLog.Print();
  // Best Sellers interaction
  [0.0500] 3: CClient.BestSellers();
           3.1: CShop.Promote();
           3.2: CShop.BestSellers();
           3.2.1: CLog.Print();
  // Product Detail interaction
  [0.1700] 4: CClient.ProductDetail();
           4.1: CProduct.GetDetails();
           4.1.1: CLog.Print();
  // Search Request interaction
  [0.2000] 5: CClient.SearchRequest();
           5.1: CShop.Promote();
  // Search Results interaction
  [0.1700] 6: CClient.SearchResults();
           6.1: CShop.Promote();
           6.2: CShop.SearchProduct();
           6.2.1: CLog.Print();
  // Shopping Cart interaction
  [0.1160] 7: CClient.ShoppingCart();
           7.1: CCart.AddProduct();
           7.1.1: CLog.Print();
           7.2: CCart.GetItems();
           7.2.1: CLog.Print();
  // Customer Reg interaction
  [0.0300] 8: CClient.CustomerReg();
  // Buy Request interaction
  [0.0260] 9: CClient.BuyRequest();
           9.1: CGate.GetCustomer();
           9.1.1: CShop.GetCustomer();
           9.1.1.1: CLog.Print();
  // Buy Confirm interaction
  [0.0120] 10: CClient.BuyConfirm();
           10.1: CGate.Checkout();
           10.1.1: CCart.Checkout();
           10.1.1.1: CLog.Print();
           10.2: CGate.GetOrderDetails();
           10.2.1: COrder.GetDetails();
           10.2.1.1: CLog.Print();
           10.3: CGate.SetOrderDetails();
           10.3.1: COrder.SetDetails();
           10.3.1.1: CLog.Print();
           10.4: CGate.Process();
           10.4.1: COrder.Process();
           10.4.1.1: CLog.Print();
  // Order Inquiry interaction
  [0.0075] 11: CClient.OrderInquiry();
  // Order Display interaction
  [0.0066] 12: CClient.OrderDisplay();
           12.1: CGate.GetCustomer();
           12.1.1: CShop.GetCustomer();
           12.1.1.1: CLog.Print();
           12.2: CGate.GetOrders();
           12.2.1: CCustomer.GetOrders();
           12.2.1.1: CLog.Print();
           12.3: CGate.GetOrderDetails();
           12.3.1: COrder.GetDetails();
           12.3.1.1: CLog.Print();
  // Admin Request interaction
  [0.0010] 13: CClient.AdminRequest();
  // Admin Confirm interaction
  [0.0009] 14: CClient.AdminConfirm();
           14.1: CGate.SetProdDetails();
           14.1.1: CProduct.SetDetails();
           14.1.1.1: CLog.Print();
};

// Infrastructure layer
class CFastNode {
  property requestout = CNodeRequestout;
  property replyin = CNodeReplyin;
  Cpu : Queue {ddd = cpu};
  MemBus : Queue;
};

class CSlowNode {
  property requestin = CNodeRequestin;
  property replyout = CNodeReplyout;
  property msgpeer = CNodeMsgpeer;
  Cpu : Queue {ddd = cpu};
  MemBus : Queue;
};

collaboration CNodeRequestout {
  1: Cpu() {dd = d * 0.5 * 120/(120+400)};
};

collaboration CNodeRequestin {
  // Invocation routing
  1: MemBus() {dd = Main.prop.population * 0.34};
  2: Cpu() {dd = 7.55};
  // Demarshaling
  3: Cpu() {dd = d * 0.5 * 400/(120+400)};
};

collaboration CNodeReplyout {
  1: Cpu() {dd = d * 0.5 * 400/(120+400)};
};

collaboration CNodeReplyin {
  1: Cpu() {dd = d * 0.5 * 120/(120+400)};
};

collaboration CNodeMsgpeer {
  1: Cpu() {dd = d * 0.5};
};

// Deployment layer
ClientNode : CFastNode {
  ClientImpl : CClient;
};

ServerNode : CSlowNode {
  ShopImpl : CShop;
  CartImpl : CCart;
  OrderImpl : COrder;
  ProductImpl : CProduct;
  CustomerImpl : CCustomer;
  GateImpl : CGate;
  LogImpl : CLog;
};
```