INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

XMI-based Transformation from UML Models to LQN Performance Models

by

Hui Shen

A thesis submitted to the Faculty of Graduate Studies
in partial fulfillment of the requirement for the degree of

**Master of Science
Information and Systems Science**

School of Computer Science
Carleton University
Ottawa, Canada

April, 2002

© Copyright 2002, Hui Shen

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-72091-8

Canada

The undersigned recommend to the Faculty of
Graduate Studies and Research acceptance of the thesis

**XMI-based Transformation from UML
Models to LQN Performance Models**

Submitted by **Hui Shen**.

in partial fulfillment of the requirements for the degree of
Master of Science
Information and Systems Science

Director, School of Computer Science

Dr. Dorina Petriu, Thesis Supervisor

Carleton University

April, 2002

ABSTRACT

The thesis proposes a graph grammar-based method for transforming automatically a UML model annotated with performance information into a Layered Queueing Network (LQN) performance model. The input to the transformation algorithm is an XML file that contains the UML model in XML format according to the standard XMI interface. The output is the corresponding LQN model description file, which can be read directly by the existing LQN solvers. The LQN model structure is generated from the high-level software architecture and from deployment diagrams indicating the allocation of software components to hardware devices. The LQN model parameters are obtained from detailed models of key performance scenarios, represented as UML activity diagrams. A Java application was designed and built in the thesis for realizing the proposed transformation method.

ACKNOWLEDGEMENT

Above everyone else, I acknowledge my thesis supervisor, Professor Dorina Petriu, for her encouragement, time, friendship, and thoroughness. Without her continuous support, guidance and trust, my studies in this area would not have been possible.

I'd like also give my thanks to Professor Murray Woodside, who taught me a lot about Performance Analysis and always gave me trust.

I would like to thank Carleton University, the School of Computer Science and the Department of Systems and Computer Engineering for giving me the opportunity to do my thesis work.

I extend my appreciation to all my friends for their helpful comments and suggestions.

I thank the Communication and Information Technology Ontario (CITO) for the financial support.

Finally, I thank my family for their love.

To My Parents and Brothers

TABLE OF CONTENTS

LIST OF FIGURES.....	IX
LIST OF TABLES.....	XII
LIST OF ACRONYMS.....	XIII
1 INTRODUCTION	14
1.1 Thesis Motivation	14
1.2 Thesis Contributions	17
1.3 Brief content descriptions	18
2 LITERATURE REVIEW	19
2.1 Software Performance Engineering(SPE).....	19
2.2 Layered Queueing Network(LQN)	20
2.2.1 LQN model elements	22
2.2.2 LQN parameters	26
2.2.3 Solve the LQN Model	27
2.3 The Unified Modeling Language(UML).....	27
2.3.1 UML Metamodel	30
2.4 Performance Profile	31
2.5 XML and XMI	34
2.6 UML Design Tools	38
2.6.1 ArgoUML	38
2.6.2 Rational Rose	39
2.7 Architectural Patterns.....	40
3 TRANSFORMATION FROM UML MODELS TO LQN MODELS AT DIAGRAM LEVEL.....	43
3.1 Conceptual Description.....	43
3.1.1 Deployment Diagram	44
3.1.2 High Level Collaboration Diagram.....	45
3.1.3 Activity Diagram	46

3.1.3.1 Basic elements	47
3.2 UML to LQN Transformation Rules at UML diagram level	48
3.2.1 UML to LQN structure Transformation – Step1	50
3.2.2 UML to LQN behavior Transformation –Step2.....	54
3.2.2.1 Grammar	57
3.2.2.2 Transformation Step 2.1	60
3.2.2.3 Transformation step 2.2	65
4 TRANSFORMATION FROM UML TO LQN MODEL AT METAMODEL	
LEVEL.....	72
4.1 Conceptual Description	72
4.1.1 UML metamodel and Novosoft UML API	72
4.1.1.1 UML 1.3 metamodel	72
4.1.1.2 UML 1.3 physical metamodel	74
4.1.1.3 Novosoft UML metamodel.....	74
4.1.1.4 Novosoft UML API.....	74
4.2 UML Metamodel Representation for the Related Diagrams	75
4.2.1 Deployment Diagram	75
4.2.2 Collaboration Diagram	76
4.2.3 Activity Diagram	78
4.3 Reading XML files.....	81
4.4 Generate LQN structure.....	82
4.4.1 Algorithm for structure transformation.....	82
4.4.1.1 Step 1.1: Obtain information from the Deployment diagram	83
4.4.1.2 Step 1.2: Obtain information from the high level collaboration	
diagram.....	84
4.5 Generate LQN detailed elements	85
4.5.1.1 Transformation step 2.1	86
4.5.1.1.1 Conceptual Description	86
4.5.1.1.2 Pseudocode	87
4.5.1.2 Transformation step 2.2	88
4.5.1.2.1 Conceptual Description	90

4.5.1.2.2 Pseudocode	90
4.6 Practical issues related to the XML input files	98
5 CASE STUDY	99
5.1 Description of the Group Communication Server application	99
5.2 Apply transformation algorithm to GCS application	101
5.2.1 Case 1: Subscribe Document & Unsubscribe Document	101
5.2.1.1 Use case 1: subscribing to a document	101
5.2.1.2 Use Case2: unsubscribe from the specific Document.....	107
5.2.2 Case 2: Retrieve Documents & Submit Document.....	108
5.2.2.1 Use case 3: retrieving the most recent version of a document.....	108
5.2.2.2 Use case 4: submitting a new document	117
6 CONCLUSIONS.....	118
6.1 What was done.....	118
6.2 Future work	119
6.2.1 Forward translation from UML to LQN.....	119
6.2.2 Backward translation from LQN to UML	119
APPENDIX “A”.....	121
APPENDIX “B”.....	122
APPENDIX “C”	123
APPENDIX “D”	125
REFERENCES	127

LIST OF FIGURES

Figure 1-1 Tool inter-operability	15
Figure 1-2 Global view of thesis scope	16
Figure 2-1 a simple example of LQN model.....	22
Figure 2-2 Activity Graphical Notation.....	24
Figure 2-3 Different types of LQN messages	26
Figure 2-4 Four-layer Metamodel Architecture	30
Figure 2-5 Domain model in the UML Performance Profile.....	33
Figure 2-6 structure and behaviour aspects of Client/Server Pattern.....	41
Figure 2-7 structure and behaviour aspects of Forwarding Pattern	42
Figure 3-1 Simple Example of Deployment Diagram	44
Figure 3-2 Simple Example of high level collaboration diagram.....	45
Figure 3-3 Simple Example of activity diagram	46
Figure 3-4 Step 1 applied to the Client/Server with blocking client	51
Figure 3-5 Step 1 applied in Forwarding pattern.....	53
Figure 3-6 <Branch/Merge> structure	60
Figure 3-7 <Fork/Join> structure	60

Figure 3-8 Step 2.1 applied to Client/Server with blocking client	61
Figure 3-9 Step 2.1 applied to Client/Server with non-blocking client.....	63
Figure 3-10 Step 2.1 applied to Forwarding pattern.....	64
Figure 3-11 Algorithm Step 2.2 applied to Client/Server with blocking client.....	66
Figure 3-12 Algorithm Step 2.2 applied to Client/Server with non-blocking client.....	68
Figure 3-13 Algorithm Step 2.2 applied to Forwarding pattern	71
Figure 4-1 Top-Level Packages	73
Figure 4-2 Foundation Package	73
Figure 4-3 Behavioral Elements Package.....	73
Figure 4-4 Part of MetaModel Representation for a Deployment Diagram	75
Figure 4-5 Part of MetaModel Representation for high level collaboration diagram	77
Figure 4-6 Part of MetaModel Representation for activity diagram.....	78
Figure 4-7 Flow of thesis implementation	82
Figure 4-8 The main tree for the recursive descent parser.....	91
Figure 4-9 The subtree <partition>	91
Figure 4-10 subtree for <RVreq>.....	95
Figure 4-11 subtree for <RecProcReply>.....	96

Figure 5-1 Use Cases in the GCS application.....	100
Figure 5-2 Group Communication Server: high-level architecture, deployment and activity diagram for “Subscribe” scenario	103
Figure 5-3 parsing tree for UserT partition	104
Figure 5-4 parsing tree for MainProc partition.....	106
Figure 5-5 parsing tree for DocInf Partition	106
Figure 5-6 LQN model for Subscribe scenario	107
Figure 5-7 Group Communication Server: high-level architecture, deployment diagram.....	109
Figure 5-8 Group Communication Server: activity diagram for “Retrieve ” scenario	110
Figure 5-9 Parsing tree for the UserT partition.....	112
Figure 5-10 Parsing tree for MainProcess.....	113
Figure 5-11 Parsing tree for RetriveProcess	115
Figure 5-12 LQN submodel for Retrieve Document use case.....	116

LIST OF TABLES

Table 1 Nonterminal labels for Client/Server and Forwarding Patterns	58
--	----

List of Acronyms

API	Application Program Interface
CASE	Computer Aided Software Engineering
DTD	Document Type Declaration
DOM	Document Object Model
GCS	Group Communications Server
LQN	Layered Queueing Network
MOF	Meta Object Facility
NSUML	Novosoft UML
OCL	Object Constraint Language
OMG	Object Management Group
OO	Object-Oriented
OMT	Object Modeling Technique
PROGRES	Programmed Graph Rewriting System
PGML	Precision Graphics Markup Language
RFP	Request for Proposal
SAX	Simple API for XML
SPE	Software Performance Engineering
UML	Unified Modeling Language
W3C	World Wide Web Consortium
JAXP	JavaSoft Java API for XMI Parsing
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

1 Introduction

1.1 Thesis Motivation

The need for integrating performance analysis into the software development process throughout the software life-cycle has been recognized a long time ago[Smith90]. However, it is not applied very often in practice, due to a gap that exists between the software development on one hand and the performance analysis field on the other hand.

In the current practice, performance analysts inspect, analyze and translate software specifications into performance models “by hand”, then solve these models under different workload factors in order to identify performance problems and recommend design alternative for performance improvement. This performance analysis cycle is starting at the early design stage and continues through all software development stages. Building the models “by hand” is time consuming and expensive. Automated techniques are therefore needed to ease and accelerate the process of building performance models.

An open research problem and challenge is to automate the process of deriving performance models from software specification and to integrate the supporting tools in a unique environment, as shown in Figure 1.1. The goal of the proposed research is to bridge that gap, by automating the construction of the performance model from a UML model, and by keeping the two models consistent. Any change in the UML model will be reflected immediately in the performance model. Also, the software developer does not need to learn how the performance model is built and solved. This will increase the usability of performance modeling. The big advantage is that the software development team will be able to verify from early stages if their product will meet its performance

requirements. This will reduce the failure rate of software projects due to poor performance, and will increase the overall productivity.

Attempts have been made for the automatic transformation of a UML model into a LQN performance model ([Wang99], [Amer01]) using a graph rewriting tool PROGRES [Schurr90]. One of the limitations of using PROGRES is that it introduces an extra step necessary for translating UML models in XMI format into PROGRES input files.

Figure 1-1 Tool inter-operability

This thesis proposes a graph grammar-based method for transforming automatically a UML model annotated with performance information into a Layered Queueing Network(LQN) performance model. The input to the transformation algorithm is an XML file that contains the UML model in XML format according to the standard XMI interface. The output is the corresponding LQN model description file, which can be read

directly by the existing LQN solvers. The thesis does the “forward” UML to LQN transformation (black arrow in Figure1.1), but does not attempt the “backward” LQN to UML transformation(gray arrows in Figure 1.1). A CASE tool, (e.g. Rose or ArgoUML), generates an XML file representing a UML model that contains up to nine different diagram types. In the thesis, however, we are using only a subset of UML diagrams: deployment diagram, collaboration diagram and activity diagram. The Java application built in the thesis is shown in Figure1.2. First, an XMI reader reads the XML file generated by a UML tool, converts its elements to UML metaobjects, and builds the internal data structure for the model. Then the information in deployment diagram, collaboration diagram and activity diagram are extracted from the model and converted to the LQN model by applying the appropriate rules. The LQN model is in text file format. Our application uses the metamodel library NSUML and its API [NSUML99], which help us to read and process XMI files.

Figure 1-2 Global view of thesis scope

1.2 Thesis Contributions

- Contributions to knowledge
 - Propose a graph grammar- based method for transforming automatically a UML model annotated with performance information into a LQN model. The input to our transformation algorithm consists of XML files that contain the UML model in XML format. The output is the corresponding LQN model description file, which can be read directly by existing LQN solvers.
 - The main part of the algorithm is a graph grammar- based method for generating LQN elements from activity diagrams. An important contribution of the thesis is the development of the grammar and of its parsing algorithm.
- Practical Contributions
 - Implementations in Java of the proposed method to transform UML models to LQN model. The implementation is consistent with version 1.3 of the UML standard, and uses a free-source library named NovoSoft UML[NSUML99] that implements the UML metamodel and its API.

The main results of the thesis were published in the following paper:

D.C.Petriu, H.Shen, "Applying the UML Performance Profile: Graph Grammar based derivation of LQN models from UML specifications", in Computer Performance Evaluation - Modelling Techniques and Tools, (Tony Fields, Peter

Harrison, Jeremy Bradley, Uli Harder, Eds.) Lecture Notes in Computer Science 2324, pp.159-177, Springer Verlag, 2002.

1.3 Brief content descriptions

The thesis includes 6 chapters, which are structured as follows:

Chapter 2 gives an overview of the background literature for the thesis, such as Software Performance Engineering (SPE), Layered Queueing Networks (LQN), the Unified Modeling Language (UML), Performance Profile, XML/XMI, UML Design Tools and Architecture Patterns used in the thesis.

Chapter 3 discusses the notation for the UML diagrams used in our thesis: deployment diagram, collaboration diagram, and activity diagrams and then provides the transformation grammar used by the transformation algorithm. Next we propose transformation rules represented at the notation level, which includes UML to LQN structure and behaviour transformation.

Chapter 4 investigates the UML metamodel and the Novosoft UML API that was used to implement the transformation [NSUML99]. The API implements the UML metamodel. The chapter continues by describing the transformation rules at the metamodel level. The algorithm for the transformation is also given.

Chapter 5 presents a case study that is developed to validate and verify the transformation process.

Chapter 6 concludes the thesis research and opens some directions for future work.

2 Literature review

2.1 Software Performance Engineering(SPE)

Meeting performance requirements such as responsiveness or throughput is vital for many software products, especially for real -time systems[Smith90]. However, the software engineering practice has largely ignored early performance evaluation. The performance problems have not been eliminated by distributed systems, parallel processing or any other hardware advances. Most current practices in software design and implementation are based on a “design it now and fix performance later” approach [Smith90][Smith+01]. In many situations, the prototype fails to meet the performance requirements, which will result in an expensive redesigning of the system. It is economically prudent to evaluate hardware and software cost alternatives for achieving system requirements and to select the most cost-effective alternative before constructing new systems.

Software Performance Engineering (SPE) introduced in [Smith90] is an alternative to “fix-it-later” approach. The primary goal of SPE is to provide a method for the evaluation of the performance of software from its initial conception through its entire lifecycle. It adopts quantitative methods successfully used in engineering and other fields in order to assess the performance effects of different design and implementation alternatives. The process begins early in the software lifecycle and uses quantitative methods to identify satisfactory designs and to eliminate those that are likely to have unacceptable performance before developers invest significant time in implementation [Smith90]. SPE continues throughout the whole lifecycle of the system development, from the requirements analysis phase, preliminary design, detailed design, coding to testing states

in order to predict and manage the performance of the evolving software and monitor and report actual performance against specifications and predictions. Software Performance Engineering uses analytic performance models because of their lower cost in comparison to simulation and measurement-based approaches.

In the early development states, the input parameter values to an analytic model are estimates based on previous experience with similar systems, on measurement of reusable components, on known platform overheads (such as system call execution times), or on time budgets allocated to different components. As the development progresses and more components are implemented and measured, the model parameters become more accurate and so do the result.

2.2 Layered Queueing Network(LQN)

Today's computer systems are more complex , more rapidly evolving, and more essential to conducting business than those of even a few years ago. The result is an increasing need for tools and techniques that assist in understanding the behavior of these systems. System modeling is one of the answers to solve this problem.

A model is an abstraction of a system, an attempt to distill from the many details of the system, only those aspects that are essential to capture the characteristic studied in the model (e.g.,performance). Once the model has been defined through this abstraction process, it can be parameterized to reflect any of the alternatives under study and then evaluated to determine its characteristics for different alternatives.

Queueing Network modeling has been widely used to model and analyze the performance of traditional time-sharing computer systems involving service. Examples of such systems

range from communication systems, computer networks and transaction models to models of production jobs and vehicular traffic. However, it often fails to capture complex interactions among various software and hardware components in client-server distributed processing system.

The Layered Queueing Networks(LQN) introduced in [Woodside89],[Woodside+95], [Frank+95] provides examples of new modeling techniques that was intended to handle such complex interactions. Other researchers have also adopted the layered model [Rolia+95],[Ramesh+2000] and [Kähkipuro01]. The main difference with respect to QN is that LQN can easily represent nested services: a server which receives and serves client requests, may become in turn a client to other servers from which it requires nested services while serving its own clients. A layered model describes the system architecture as software and hardware modules with resources embedded in them in a layered fashion, and with resource demands as parameters [Woodside89].

A LQN model is represented as an acyclic graph, whose nodes named tasks represent software entities and hardware devices, and arcs denote service requests. The software entities are drawn as parallelograms, and the hardware devices as circles. The nodes with outgoing but no incoming arcs play the role of clients, the intermediate nodes with both incoming and outgoing arcs are usually software servers and the leaf nodes are hardware servers such as processors, I/O devices, communication network, etc.

Figure 2.1 shows a simple example of a LQN model of a web server: at the top there is a customer class with a given number of stochastic identical clients. Each client sends demands for different services of the WebServer. Each kind of service offered by a LQN task is modeled as a so-called *entry* which will be discussed with more details in the next

section, drawn as a parallelogram “slice” in the figure. Every entry has its own execution time and demands for other services (given as model parameters). In this case, the WebServer entries require services from different entries of the Database task. Each software task is running on a processor shown as a circle. Also as circles are shown the communication network delays and the disk device used by the Database. It is worth mentioning that the word *layered* in the LQN name does not imply a strict layering of tasks (for example, tasks in a layer may call each other or skip over layers).

Figure 2-1 a simple example of LQN model

The next section describes in more details the elements of a LQN model.

2.2.1 LQN model elements

Task

The task is a very important entity in LQN. It models a concurrent process or object, or a logical or physical resource. In other words, a task is used to model software entities as

well as hardware devices.

As mentioned before, a LQN model is represented as an acyclic graph, where the nodes are tasks. The tasks can be classified into three categories: pure clients, pure servers, and active servers. Pure clients, which can only send messages (or requests), are used to model actual users and other input sources. Pure servers, which only receive requests, are normally used to model hardware devices such as processors, network devices, or disks. Active servers, which can receive requests as well as make their own request to other servers, are used to model typical operating system processes.

As we've already mentioned, the word *layered* in the LQN name does not imply a strict layering of tasks. Tasks in a layer may call each other or skip over layers. Another thing that should be mentioned here is that, although not explicitly illustrated in the LQN notation, each server has an implicit message queue called the request queue, where the incoming requests are waiting their turn to be served. The default scheduling policy of the request queue is FIFO, but the LQN model also supports other policies such as preemptive policy, non-preemptive, random, etc.

Entries:

A LQN task may offer more than one kind of services. Each service offered by a LQN task is modeled by a so-called entry, drawn as a parallelogram slice in Figure 2.1. Every entry has its own execution time and demands for other services (given as model parameters).

An entry can be decomposed into more detailed activities which describe its execution scenario. Activities are typically required when fork & join, branch & merge interactions occur in the entry.

Activities

A recent extension to LQN [Franks99] allows for an entry to be further decomposed into activities if more details are required to describe its execution. Activities are used to model the lowest level of details in the LQN performance model. They can be connected sequentially, by branch & merge, or by fork & join. An activity may have service time demand on the processor on which its task runs on, and it can make request to other entries of other tasks by synchronous or asynchronous messages. Figure 2.2. shows the graph notation of activities in LQN model. The successor activities after the And-Fork will execute in parallel, while in the Or-Fork case, only one of the successor activities is executed, with probability P_i . The sequentially connected activities will be executed one

Figure 2-2 Activity Graphical Notation

Arcs (Request)

Arcs in the LQN model describe the requests from one phase of one entry (or activity if any) to another entry. The labels on the arcs denote the average number of requests.

There are three types of LQN messages, synchronous, asynchronous, and forwarding shown in Figure 2.3. This figure is taken from [Petriu+2000].

- A synchronous message represents a request for service sent by a client to a server, where the client remains blocked until it receives a reply from the provider of service. If the server is busy when a request arrives, the request is queued and waits its turn. After accepting a message, the server starts to serve it by executing a sequence of phases. At the end of phase 1, the server replies to the client, which is unblocked and continues its work. The server continues with the following phases, if any, working in parallel with the client, until the completion of the last phase. After finishing the last phase, the server begins to serve a new request from the queue, or becomes idle if the queue is empty.
- An asynchronous message represents that after sending this message, the client does not block and the server does not reply back, instead it will execute its phase.
- In the case of forwarding message, the client sends a synchronous request to Server1, which begins to process the request, then after some processing, it will forward the request to Server2 at the end of phase1. Server1 proceeds normally with the remaining phases in parallel with Server2, then goes on to another cycle. The client remains blocked until it receives the reply from Server2. We should mention that the forwarding request can be served by a chain of servers which means this chain contains any number of servers, in which case the client waits until it receives a reply

from the last server in the chain.

Figure 2-3 Different types of LQN messages

2.2.2 LQN parameters

The performance information includes:

- The number of processor nodes and the task allocated on them;
- The multiplicity of each task and processor node in the system;
- Scheduling discipline for each software and hardware server;
- The mean service time demand per visit for each activity and /or phase of an entry;
- The mean number of synchronous, asynchronous, or forwarded messages sent from a phase or activity of an entry to another entry;

2.2.3 Solve the LQN Model

The solving tools provided by the toolset in [Franks99] can be used to solve LQN models analytically. Typical results of a LQN model are response times, throughput, utilization of servers on behalf of different types of requests, and queueing delays. The LQN results may be used to identify the software and/or hardware bottlenecks that limit the system performance under different workloads and configurations[Neilson+95].

2.3 The Unified Modeling Language(UML)

The Unified Modeling Language (UML), arising from the combination of Booch's, Jacobson's and Rumbaugh's methodologies [Booch94][Jacobson+92] [Rumbaugh+91], is becoming a standard notation for analysis, design and implementation of object oriented systems. It has almost completely superseded the earlier OO (Object-Oriented) methodologies, such as the Object Modeling Technique (OMT) [Rumbaugh+91], Booch's Methodology [Booch94], OORAM [Reenskaug96], Syntropy [Cook+94] and many others. It is a language for specifying, visualizing, constructing, and documenting software systems, including their structure and design, in a way that meets all of these

requirements.

The version 1.3 [UML1.3] is used throughout the thesis. The latest version 1.4 has been adopted as the standard in September, 2001, which is described in [UML1.4] and UML2.0, the next major upgrade, is already in the works.

UML is defined by an Object Management Group (OMG)[OMG] document. We will discuss briefly the specific parts which are relevant to the thesis.

- The UML Semantics section is primarily intended as a comprehensive and precise specification of the UML's semantic constructs. It provides the semantics for all modeling notations described in the UML Notation Guide, which includes support for a wide range of diagram techniques: class diagram, object diagram, use case diagram, sequence diagram, collaboration diagram, state diagram, activity diagram, component diagram and deployment diagram[UML1.3].

This section defines the UML "abstract syntax" in the form of a set of UML package. The Package Specification Structure contains one or more of the following subsections.

- Abstract Syntax, which is presented in the form of UML class diagrams. Each package contains a set of metaclasses defining the constructs and their relationships. The metaclasses in the package and their attributes are described in English.
- Well-Formedness Rules, which are used to specify constraints over attributes and associations defined in the metamodel. They are expressed in the Object

Constraint Language (OCL).

- **Semantics:** which is used to define the meanings of the constructs using English. Since only concrete metaclasses have a true meaning in the language, only these are described in this section.
- **UML Notation Guide** describes the notation for the visual representation of the Unified Modeling Language (UML). This provides examples of UML diagrams together with English description. The mapping of notation elements to metamodel elements is also described.

The UML provides a rich set of modeling concepts and notations that have been carefully designed to meet the needs of typical software modeling projects. However, users may sometimes require additional features and/or notation beyond those defined in the UML standard. In addition, users often need to attach non-semantic information to models. These needs are met in UML by three built-in extension mechanisms:

- **Stereotype** is used to classify other UML elements so that they behave in some respects as if they were instances of new “virtual” metamodel classes.
- **TaggedValue** is used to permit arbitrary information to be attached to any model element. More than one tagged value pairs may be attached to a given model element, but each of them needs to have a unique tag name.
- **Constrains** are used to attach semantic restrictions to a ModelElement. UML uses the OCL to define constrains.

In the thesis, we use the TaggedValue to attach performance information to the activity

diagrams, which can be used to get the detailed LQN elements and parameters.

2.3.1 UML Metamodel

The UML metamodel is defined as one of the layers of a four-layer metamodeling architecture shown in Figure 2.4[UML 1.3].

M3: is responsible for defining the language for specifying a metamodel A meta-metamodel defines a model at a higher level of abstraction than a metamodel, and is typically more compact than the metamodel that it describes.

M2: is an instance of a meta-metamodel and is responsible for defining a language specifying models. It is the definition of UML.

M1: is an instance of a metamodel and is responsible for defining a language that describes an information domain.

M0: is an instance of a model and is responsible for describing a specific information domain.

Figure 2-4 Four-layer Metamodel Architecture

This architecture is a proven infrastructure for defining the precise semantics required by complex models. The fundamental relationship between these layers is intended to be the instance-of relationship, which is clearly expressed in UML specification [UML 1.3]. In chapter 4, we will give more details about the UML metamodel related to deployment diagram, activity diagram and collaboration diagram.

2.4 Performance Profile

Object Management Group (OMG) has noticed that the lack of a quantifiable notion of time and resources in the UML was an impediment to its broader use for real-time and embedded systems. So OMG issued a request for proposal (RFP) asking for a UML profile for “schedulability, performance and time”. The first draft of the profile was made public in August, 2000, and the improved version in June 2001 [Profile01]. The profile focuses on properties that are related to modeling of time and time-related aspects such as timeliness, performance and schedulability.

According to [Profile01], the UML Performance Profile provides facilities for:

- capturing performance requirements within the design context
- associating performance-related QoS characteristics with selected elements of the UML model
- specifying execution parameters which can be used by modeling tools to compute predicted performance characteristics
- presenting performance results computed by modeling tools or found by measurement.

The Profile describes a domain model, shown in Figure2.5, which identifies basic

abstractions used in performance analysis. *Scenarios* define response paths through the system, and can have QoS requirements such as response times or throughputs. Each scenario is executed by a job class, called here a *workload*, which can be closed or open and has the usual characteristics (number of clients or arrival rate, etc.). Scenarios are composed of *scenario steps* that can be joined in sequence, loops, branches, fork/joins, etc. A scenario step may be an elementary operation at the lowest level of granularity, or may be a complex sub-scenario composed of many basic steps. Each step has a mean number of repetitions, a host execution demand, other demand to resources and its own QoS characteristics. Resources are another basic abstraction, and can be active or passive, each with their own attributes.

The Performance profiles maps the classes from Figure 2.5 to stereotypes that can be applied to a number of UML model elements, and each class attribute to a tagged value.

My thesis uses the following classes in this domain model to attach the performance information:

- The basic abstraction PStep is mapped to the stereotype <<PAstep>> that can be applied to the following UML model elements: Message and Stimulus (when the scenario is represented by an interaction diagram) or ActionState and SubactivityState (when the scenario is represented by an activity diagram). The thesis uses PADemand tag to attach the host execution demand to each of the ActionState and/or SubactivityState in the activity diagram. The other tags of <<PAstep>> used in the thesis include PApProb representing the probability that the ActionState and/or SubactivityState will be executed, PARep representing the number of times the ActionState and/or SubactivityState is repeated.

Figure 2-5 Domain model in the UML Performance Profile

- The basic abstraction **ProcessingResource** is mapped to the stereotype `<<PAhost>>` that can be applied to the following UML model elements: Classifier, Node, ClassifierRole, Instance, and Partition. The thesis uses `PAschdPolicy` tag to attach the CPU scheduling policy to Node.
- The basic abstraction **ClosedWorkload** is mapped to the stereotype `<<PAclosedLoad>>` that can be applied to model elements such as Message, Method, ActionState etc. The thesis uses `PApopulation` tag to attach the information about the number of users to the first ActionState of the Client partition. The thesis also uses `PAextDelay` tag to identify the user's "Think time". This tag is also attached to the first ActionState of the Client partition in the activity diagram.

In UML, scenarios are most directly modeled either using collaborations or activity graphs. The ways in which the performance domain concepts are represented in the two

approaches can be quite different. So, mapping performance domain concepts into UML equivalents has two approaches: the collaboration-based approach and the activity-based approach.[Profile01] In the thesis, we choose activity diagram to represent scenario. In this approach, scenarios are modeled by the set of states/activities and transitions of the activity graph. Each action or subactivity state the graph is stereotyped as a <PAstep>.

In our implementation, we process XML files produced by current UML tools, which obviously do not support the Performance Profile yet. Therefore, we have attached the tagged values such as PAdemand, PAprior, associated with the stereotypes "by hand" to different model elements.

2.5 XML and XMI

XML

XML is the Extensible Markup Language. It is designed to improve the functionality of the Web by providing more flexible and adaptable information identification and is fast becoming the standard for data interchange on the Web.

It is called extensible because it is not a fixed format like HTML (a single, predefined markup language). Instead, XML is actually a 'metalanguage' --a language for describing other languages--which let you design your own customized markup languages for limitless different types of documents. XML can do this because it is written in SGML, the international standard metalanguage for text markup systems (ISO 8879).

There are a number of reasons for XML's growing acceptance. This section lists a few of the most important ones.

- Plain Text

Since XML is not a binary format, you can create and edit files with anything from a standard text editor to a visual development environment. That makes it easy to debug the programs, and makes it useful for storing small amounts of data. At the other end of the spectrum, an XML front end to a database makes it possible to efficiently store large amounts of XML data as well. So XML provides scalability for anything from small configuration files to a company-wide data repository.

- Data Identification

XML tells what kind of data there is, not how to display it. Because the markup tags identify the information and break it up into parts, the data can be used in different ways by different applications.

- Style

Since XML is style-free, it can be processed by different stylesheet to produce output in postscript, TEX, PDF, or some new format that hasn't even been invented yet.

- Easily Processed

As mentioned earlier, regular and consistent notation makes it easier to build a program to process XML data. That restriction is a critical part of the constraints that make an XML document well formed. (Otherwise, the XML parser won't be able to read the data.) Since XML is a vendor-neutral standard, one can choose among several XML parsers, any one of which simplifies the processing XML data.

- Hierarchical

Finally, XML documents benefit from their hierarchical structure. Hierarchical document structures are, in general, faster to access because one can find easily a certain part, like stepping through a table of contents. They are also easier to rearrange, because each piece is delimited.

XML consists of two parts: documents and DTDs (Document Type Declarations). DTD serves as a grammar for the underlying XML document ([Martin99], [Maruyama99]). A DTD specifies the kinds of tags that can be included in a XML document, and the valid arrangements of those tags. DTD are used to make sure that the XML structure created is correct or that the XML read is valid. However, while the DTD mechanism was the first method defined for specifying valid document structure, it was not the last. Several newer schema specifications have been devised[Schema].

The use of XML introduces the need for extra tools such as XML parsers [XML1.0] or APIs like Simple API for XML (SAX) [Armstrong01]. More on these topics used for interchanging UML models are discussed in chapter 4.

More information on XML can be found in [W3C], [Ducharme99], [Leventhal+98], [Maruyama99] and [Armstrong01].

XMI

XMI stands for “XML metadata Interchange”. The main purpose of XMI is to enable easy interchange of metadata between modeling tools(based on the OMG UML) and metadata repositories (OMG MOF based) in distributed heterogeneous

environments. XMI integrated three key industry standards:

- XML - eXtensible Markup Language, a W3C standard.
- UML- Unified Modeling Language, an OMG modeling standard.
- MOF – Meta Object Facility, an OMG metamodeling and metadata repository standard.

The integration of these three standards into XMI combines the best of OMG and W3C metadata and modeling technologies, allowing developers of distributed systems to share object models and other metadata over the internet.[XMI1.0]

XMI defines two sets of rules that provide open interchange and leverage the capabilities of XML: DTD generation and document generation. The DTD generation is used to specify an interchange format, and document generation creates documents that use a given XMI DTD. The current official version of XMI is 1.1 and described in [XMI1.1].

XMI DTDs alone do not have the ability to express the semantic meaning appropriate for the model. They require the whole sets of additional concepts that are only available through complete information architectures, such as UML, MOF, and others being developed by the OMG. For example, an UML-based DTD allows interchange of object-oriented UML models. This results in the ability to interchange at both the data level (XML) and the semantic level (UML) [OMG].

Every XMI DTD contains the elements generated from an information model, e.g. a UML model, plus a fixed set of element declarations that may be used by all XMI

documents. UML DTD is the most widely used XMI DTD. It is a physical mechanism for interchanging UML models conforming to the UML metamodel.

2.6 UML Design Tools

UML Design Tools are UML-based CASE (Computer Aided Software Engineering) tools that can support the use of design diagrams in the development of an object-oriented software tools. In the thesis, we use two UML Design Tools: ArgoUML [ArgoUML] and Rational Rose [Rose]. Both tools are XMI-compliant. ArgoUML and Rose use different mechanisms to generate an XMI file for a model. In ArgoUML, several files will be generated each time when a model is saved. This includes an XMI file that stores model information and PGML files that store layout information. Rose saves a model in its .mdl file. A separate XMI file (Rose uses .xml as its extension) is obtained through a function called "Export Model to UML" in the Tools menu.

The following two sections will briefly describe these two design tools. More detailed user manuals can be found in [ArgoUML] and [Rose], respectively.

2.6.1 ArgoUML

ArgoUML is not only a free UML modeling tool, it is also an Open Source Development project where people are invited to contribute.

Following are the major Argouml features regarding UML and XMI :

- **Runs on any platform with Java 1.2** : ArgoUML is coded entirely in Java and uses the Java Foundation Classes. This allows ArgoUML to run on virtually any

platform.

- **Standard UML Meta-Model :** ArgoUML is compliant with the OMG Standard for UML in its version 1.3. The code for the internal representation of an UML model is completely generated from the specification and, thus, follows it very closely. To achieve this, a special metamodel library [NSUML] was developed by Novosoft company.
- **XMI-Support:** XMI is an XML based exchange format between UML tools. ArgoUML uses this as a standard saving mechanism so that easy interchange with other tools and compliance with open standards are secured. XMI version 1.0 for UML 1.3 is used. This also permits one to convert Rational Rose models to ArgoUML. This currently only includes model information, but no graphical information (like layout of diagrams).

2.6.2 Rational Rose

Rational Software Corporation's well-known Rose modeling tool has led the object-oriented analysis and design market for years and it was one of the first tools to support the UML [Rose]. Today, Rational is one of the OMG's most active participants in maintaining and enhancing the standard [Hess00].

Rose features not only include expanded round-trip engineering, support for UML 1.3, but they also include Rose Extensibility for developing add-in functionality. Several third-party vendors already have used the extensibility features to integrate Rose with their tools or environments. One of them is Unisys' XMI add-in. Rose does not support directly the

generation of XMI from UML models. Instead, Unisys' XMI add-in provides this support, which is available at Rational's web site [Rose]. The thesis used the XMI add-in to generate XML files containing activity diagrams, which is part of the input to our transformation process.

2.7 Architectural Patterns

In our approach, the structure of the LQN model is generated from the high-level architecture, and more exactly from the architectural patterns used in the system. Frequently used architectural solutions are identified in literature as *architectural patterns* (such as pipeline and filters, client/server, client/broker/server, layers, master-slave, blackboard, etc.). A pattern introduces a higher-level of abstraction design artifact by describing a specific type of collaboration between a set of prototypical components playing well-defined roles, and helps our understanding of complex systems. Each architectural pattern describes two inter-related aspects: its *structure* (what are the components) and *behaviour* (how they interact). In the case of high-level architectural patterns, the components are usually concurrent entities that execute in different threads of control, compete for resources, and their interaction may require some synchronization. The patterns are represented as UML *collaborations* (not to be confused with UML *collaboration diagrams*, a type of interaction diagrams) [UML1.3]. The symbol for a collaboration is an ellipse with dashed lines that may have an "embedded" square showing the roles played by different pattern participants.

In Figure2.6 and Figure2.7 are shown the structure and behaviour of two patterns used in our case study: Client/Server and ForwardingServerChain. The Client/Server

pattern has two alternatives: the one shown in Figure 2.6.b is using a rendezvous communication style (where the client sends the requests then remains blocked until the sender replies). We will name this “Client/Server with blocking client”. The alternative from Figure 2.6.c is through 2 asynchronous messages. The Client will continue its work, and only later will try to receive the reply, waiting for it if necessary. The reply is sent by the server through another asynchronous request. We will name this “Client/Server with non-blocking client”.

a). Client/Server collaboration

b). Client/Server with rendezvous

c). Client/Server without client blocking after sending request

Figure 2-6 structure and behaviour aspects of Client/Server Pattern

The ForwardingServerChain, shown in Figure 2.7, is an extension of the Client/Server

pattern, where the client's request is served by a series of servers instead of a single one. There may be more than two servers in the chain (only two are shown in Figure 2.7). The servers in the middle play the role of ForwardingServer, as each one forwards the request to the next server in the chain after doing their part of service. The last server in the chain plays the role of ReplyingServer as it sends the reply back to the client. In this thesis we show how these two patterns are converted into LQN models. More architectural patterns and the corresponding rules for translating them into LQN are described in [Petrium+2000].

Figure 2-7 structure and behaviour aspects of Forwarding Pattern

The architectural patterns are important for the UML to LQN transformation because the generated LQN model reflects both the structure and the behaviour of the software model, as explained in more detail in Chapter 3.

3 Transformation from UML Models to LQN Models at diagram level

The chapter is organized as follows. In the first section we discuss the notation for the UML diagrams used in our thesis: deployment diagram, collaboration diagram, and activity diagrams. Next we propose transformation rules represented at the notation level. The metamodel representations behind the transformation rules will be described in next chapter.

3.1 Conceptual Description

The Unified Modeling Language (UML) is a graphical modeling language that is used for visualizing, specifying, constructing and documenting software system. A software system is not a one-dimensional thing, it consists of concurrent multiple view as stated in [Booth+99]: “Software architecture is not only concerned with structure and behaviour, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and tradeoffs and aesthetic concerns”. Therefore, UML describes the following five complementary views: the use case view, the design view, the process view, the implementation view and the deployment view. Each of these views involves structural modeling, as well as behavioral modeling[Booch+99].

The following four diagrams are typically used to describe the static parts of a system: class diagram, object diagram, component diagram, and deployment diagram. UML provides other additional diagrams to view the dynamic parts: use case diagram,

interaction diagram (sequence and collaboration diagram), activity diagram and statechart diagram.

Each kind of diagrams focuses on a certain perspective of the system. We will review in the following three sections the main aspects of the notation for the deployment, collaboration diagram and activity diagrams. In Chapter 4, the detailed Semantics (Meta Model) of these three diagrams will be introduced.

3.1.1 Deployment Diagram

A deployment diagram is a graph of nodes connected by communication associations. Nodes may contain component instances, which indicate the software component that lives or runs on the node. Deployment diagrams are used in modeling the physical aspects of an object-oriented system. It is a diagram that shows the configuration of run time processing nodes and the components that live on them. Figure 3.1 shows a simple example of deployment diagram. This example shows three nodes containing three components: client, server and database, each on its own node. These components are connected by wide area network from client to server and local area network from server to database.

Figure 3-1 Simple Example of Deployment Diagram

3.1.2 High Level Collaboration Diagram

According to UML notation, a collaboration can be used to specify the implementation of design constructs such as design patterns. In section 2.7, it was shown that a collaboration has two aspects: structural and behavioral. Figure 2.6 defines the collaboration for the Client/Server pattern, and Figure 2.7 the collaboration for the Forwarding Pattern. In Figure 3.2 is shown the use of Client/Server collaboration (pattern) in the collaboration diagram that illustrates the high-level architecture of a simple system. The use of a Collaboration is shown as a dashed ellipse containing the name of the Collaboration. A dashed line is drawn from the collaboration symbol to each of the symbols denoting Objects or Classes that participate in the Collaboration. Each line is labeled by the role of the participant. Therefore, a collaboration symbol can show the use of a design pattern together with the actual Classes that occur in that particular use of the pattern [UML1.3]. Figure 3.2 shows a simple example of high-level collaboration diagram. This example shows two uses of Client/Server pattern, one between *client* and *server*, and the other between *server* and *database*. The *client* component plays the role of "Client" in the Client/Server collaboration with *server*, and the *server* component plays the role of "Server". In the second collaboration between *server* and *database*, *server* plays the role of "Client", and the *database* plays the role of "Server".

Figure 3-2 Simple Example of high level collaboration diagram

3.1.3 Activity Diagram

Based on the definition of [UML1.3], an activity diagram shows the flow of control from activity to activity. It is a special form of a state diagram in which most of the states are actions and in which most of the transitions are triggered by the completion of the actions. An activity is an ongoing nonatomic execution within a state machine. An activity ultimately results in some actions, which is made up of executable atomic computations that result in a change of system state or the return of a value.

Figure3.3 presents an activity diagram and includes the most common model elements. Next, the semantics of some key elements is described.

Figure 3-3 Simple Example of activity diagram

3.1.3.1 Basic elements

Activity states and action states

- Action states are executable computation and can't be decomposed. The work of an action state is generally considered to take a small execution time. An action state in graphical syntax is shown as shape with straight top and bottom and with convex arcs on the two sides.
- In contrast, activity states can be further decomposed, they are not atomic and in general, these states take some duration to complete. Each activity state might be composed of other action or activity states and can have its own activity diagram.

Transition

When the action or activity of a state completes, the flow of control passes immediately to the next action or activity state. Transitions will be used to specify this flow which shows the path from one action or activity state to the next. Sequential transitions are the most common transitions in activity diagrams, but there are also transitions associated with branching and forking the control flow. In order to model these, UML Pseudo-states are also needed.

Pseudo-state

- In the flowchart, a branch can be used to specify alternate paths which are taken based on some Boolean expression. It is possible to converge multiple incoming transitions into a single outgoing transition and conversely, it is also

possible to be used to split an incoming transition into multiple outgoing transition segments with different guard conditions [UML 1.3].

- UML provides fork/join vertices to represent the parallel flows of control. A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control and the join represents the synchronization of two or more concurrent flows of control.

Swimlanes

Actions and subactivities may be organized into swimlanes. A swimlane is used to organize the responsibility for the actions and subactivities it contains. Each swimlane represents the responsibility for a part of the overall activity, and may eventually be implemented by one or more objects [UML 1.3]. In our case, we are concerned with the concurrency in the system, so we choose to build the activity diagram in such a way that each swimlane corresponds to a single execution flow. This means that a swimlane will contain the activities carried out by a certain active object and any number of associated passive objects. We choose to name the swimlane with name of the active object. A swimlane is graphically separated from its neighbor by a vertical solid line, as shown in Figure3.3. In the metamodel, a swimlane maps into a Partition of States in the ActivityGraph.

3.2 UML to LQN Transformation Rules at UML diagram level

Similar to the SPE methodology from [Smith90], [Smith2001], the starting point for our algorithm is a set of key performance scenarios annotated with performance information. For each scenario we derive a LQN submodel.

As mentioned in chapter 2, a pattern introduces a higher-level of abstraction design artifact by describing a specific type of collaboration between a set of prototypical components playing well-defined roles, and helps our understanding of complex systems. Each architectural pattern has two aspects: structure and behavior. Our transformation target model, the LQN model, is represented as an acyclic graph whose nodes (named also tasks) are software entities and hardware devices, and arcs denote service requests. As shown in section 2.2.1, the LQN model consists of the following elements: tasks, entries, phases of the specific entry, activities (if any) and arcs (requests). When we do the transformation, we use high-level collaboration diagrams and deployment diagrams, which show the structural aspect of the system to extract the information for creating the tasks and the arcs of the LQN model. In the second step, we take as input activity diagrams with annotated performance information, which shows the behavioral aspect of the system and use it to generate lower-level LQN elements (entries, phases, activities and their parameters). The generated LQN model matches the kind of interaction between the high-level software components defined by the design pattern.

According to the above analysis, our transformation from UML to LQN will be divided into two parts. The first part is to create the LQN high-level structure (i.e., tasks and arcs) and the other part is to add more detailed parameterized information into the structural model so that the entire LQN model will be created.

In the next two sections, transformation rules from a UML model to a LQN model are illustrated at notational level. This provides a “bird’s-eye view” of the transformation approach. In chapter 4, these transformations will be described in detail by making use of the UML metamodel which is more abstract and less readable.

3.2.1 UML to LQN structure Transformation – Step1

The submodel structure (i.e., the processor information, the software and hardware tasks and their connecting arcs) is obtained from the high-level architecture of the UML model and from the deployment of software components to hardware devices. Two kinds of UML diagrams are taken into account in this step: a high-level collaboration diagram that shows the concurrent/distributed high-level component instances and the patterns in which they participate, and the deployment diagram. Each high-level software component is mapped to a LQN software task, and each hardware device (processor, disk, communication network, etc.) is mapped to a LQN hardware task. The type of the arcs between LQN nodes will be obtained from the design pattern described in the high-level collaboration diagram. It is important to mention that in the first transformation step from UML to LQN, we take into account only the structural aspects of the architectural patterns; their behavioral aspect will be considered in the next step. The following part in this section will show how to apply algorithm step1 to the Client/Server pattern and Forwarding pattern at UML diagram level.

Client/Server pattern is very frequently used in today's distributed systems. As described in section 2.7(Figure2.6 and Figure 2.7), We will consider two realizations of the patterns. In the first case, the client/server communication takes place through a synchronous message, where the client sends a request to the server and blocks until the reply from the server comes back. We will name this "Client/Server with blocking client". In the second case, the client/server communication happens through 2 asynchronous messages. The client sends the request to the server, but it does not block immediately.

The Client will continue its work, and only later will try to receive the reply, waiting for it if necessary. The reply is sent by the server through another asynchronous request. We will name this case “Client /Server with non-blocking client”.

Client/Server pattern with blocking client:

Figure 3-4 Step 1 applied to the Client/Server with blocking client

Figure 3.4 shows an example of the high-level collaboration diagram using the Client/Server pattern and the corresponding deployment diagram that are used to derive the structure of the LQN submodel on the right side.

The mapping process is as follows:

- Each high-level software component is mapped to a LQN software task. From the deployment diagram shown in Figure 3.4, it's very easy to get the software components information: *client* and *server*. These two components will be mapped to

the corresponding LQN software tasks with the same name as client and server.

- Each hardware device (processor, disk, communication network, etc) is mapped to a LQN hardware task. From the deployment diagram, we have two processors: ProcC and ProcS. They will be mapped to two hardware tasks named ProcC and ProcS in the LQN submodel.
- The high-level collaboration diagram provides the information that the design pattern (Client/Server) is involved between *client* and *server*. We will use this information to check the consistence between the related activity diagram and the architectural pattern. This information will also be used to generate the type of request between the corresponding tasks in the LQN model. In this case, the request arc between client task and server task is a synchronous message.

Client/Server pattern without non-blocking client :

In this case, the structural part of the generated LQN submodel is the same as for the Client/Server with blocking client shown in Figure 3.4. So we will not repeat the explanation here. However, the internal structure of the Client will be different, as explained in Section3.2.2.

Forwarding pattern is an extension of the Client/Server pattern, where the client's request is served by a series of servers instead of a single one. There may be more than two servers in the chain. The servers in the middle play the role of ForwardingServer as each one forwards the request to the next server in the chain after doing their part of service. The last server in the chain plays the role of ReplyingServer as it sends the reply back to the client.

Figure 3-5 Step 1 applied in Forwarding pattern

In Figure 3.5 is shown a simple example with one forwarding server. After accepting a request from a client, the first server task in the chain will do the processing, then it will forward the request to the replying server. The forwarder is free to continue its activity, while the client remains blocked, waiting for the reply. The replying server that continues to serve the request will eventually complete it and send the reply back directly to the client.

The mapping is as follows:

- Each high-level software component is mapped to a LQN software task. The deployment diagram shows that three software components exist in our system: *client*, *server1*, and *server2*. These software components will be mapped to the corresponding client task, server1 task, and server2 task in the LQN model.
- Each hardware device (processor, disk, communication network, etc) is mapped to a

LQN hardware task. From the deployment diagram, we have three processors: ProcC, ProcS1 and ProcS2 which are mapped to three hardware tasks with the same name as ProcC, ProcS1, and ProcS2 in the LQN model showed in the above Figure 3.5.

- The high-level collaboration diagram provides the information that the design pattern between the components *client*, *server1*, and *server2* is Forwarding pattern. We will use this information to do the consistency checking between the related activity diagram and the architectural pattern. This information will also be used to identify the type of arcs between the tasks in the LQN model. In the Forwarding case, the arc between client task, and server1 task is synchronous, and the arc between server1 task and server2 task is forwarding.

3.2.2 UML to LQN behavior Transformation –Step2

LQN task details are obtained from UML scenario models represented as activity diagrams, over which we overlay the behavioral aspect of the architectural pattern, making sure that the scenario is consistent with the patterns. By “LQN details” we understand the following elements of each task: entries, phases, activities (if any) and their execution time demands and visit ratio parameters, as described in section 2.2. A task entry is generated for each kind of service offered by the corresponding software component instance. The services offered by each instance are identified by looking at the messages received by it in every scenario taken into account for performance analysis.

Scenarios can be represented in UML by sequence, collaboration or activity diagrams. (The first two are very close as descriptive power and have similar metamodel representation). UML statecharts are another kind of diagrams for behavior description,

but are not appropriate for describing scenarios. A statechart describes the behavior of an object, not the cooperation between several objects, as needed in a scenario.

In the proposed approach, we decided to use activity diagrams for the translation to LQN. The main reason is that sequence (collaboration) diagrams are not well defined in UML yet, as they are lacking convenient features for representing loops, branches and fork/join structures. Other authors who are building performance models from UML designs have pointed out this deficiency of the present UML standard, and are using instead extended sequence diagrams that look like the Message Sequence Chart standard (see [Smith90], [Smith+01] for well known examples). We did not take the approach of extending the sequence diagrams with the missing features because our algorithm takes in XML files generated by UML tools, and parses graphs of UML metaobjects. Our implementation is consistent with the present UML metamodel and XMI interface; moreover, it uses a free-source library that implements the UML metamodel as defined in [UML1.3]. Therefore, our choice was to use activity diagrams, which are able to represent branch/merge, fork/join and activity composition.

However, the activity diagrams have a disadvantage with respect to sequence (collaboration) diagrams: they do not show what objects are responsible for different actions. An attempt to counterbalance this weakness was the introduction of “swimlanes” (or partitions) in the UML standard as we discussed in section 3.1.3. A swimlane contains actions that are performed by a certain instance or set of instances (for example, a swimlane can be associated to a whole department when modeling a workflow problem). In our approach, we associate a swimlane with each concurrent (distributed) component, which will be translated into a LQN task.

Since many UML modellers prefer sequence (collaboration) diagrams for expressing the cooperation between objects, other students working with Prof. D.Petriu have proposed an algorithm based on graph transformations for converting automatically sequence diagrams into activity diagrams[Amer01][Wong02]. The transformation associates a swimlane to all the objects that belong to a concurrent (distributed) component, and therefore adjusts the level of abstraction of the model to the needs of the translation to LQN.

In the algorithm step2, we provide a graph grammar-based transformation of activity diagrams into LQN detailed features. The graph-grammar formalism is appropriate in this case because both UML and LQN models are described by graph.

The essential idea of all graph grammars is that they are a generalization of the string grammars that are used in compilers. A string grammar (also named a formal grammar) for a language prescribes which strings of symbols are allowed in the language. It has four components:

- A set of terminal symbols – the symbols actually used in the language.
- A set of nonterminal symbols- these are used to describe parts of the language.
- A set of rules that define each nonterminal as string of symbols, each of which must be either a terminal or a nonterminal.
- A start symbol, that is, a special nonterminal whose definition is applied first whenever we want to generate an acceptable string in the language.

Similarly, the graph grammars have four main components: terminal symbols, nonterminal symbols, rules and a start symbol. The difference with respect to string

grammars is that all the terminal symbols are graph nodes and the nonterminal symbols are subgraph. In our case, the start symbol which represents the initial host graph is the set of metaobjects that represents a given activity diagram. The metaobjects representing the nodes(StateVertex and Transition)are the terminal symbols of the grammar. We will discuss later the nonterminal symbols and the rules.

Our purpose in this step is to parse the activity diagram to check first whether it is correct, then to divide it into subgraphs that correspond to various LQN elements(entries, phases, etc.). In general, parsing graph grammars is quite difficult, and in some cases even impossible[Rozenberg97]. In this case, we have found a shortcut, as explained below, that decomposes the original host graph into subgraphs described by simpler graph-grammars, very similar to string grammars. Each such subgraph corresponds to a partition and describes the behavior of a single component, dealing with sequences of scenario steps, alternative branches and fork/join structures. We were able to define and implement a top-down parser with recursive methods to parse these subgraphs.

3.2.2.1 Grammar

Before giving the description of the transformation algorithm, we introduce the graph-grammar used for parsing every subgraph. The shortcut we have found to simplify the parsing of the graph that represents an activity diagrams consists in detaching the swimlanes one from the other. This is done by traversing the graph, identifying the cross-transitions, checking if they are consistent with the communication style implied by the pattern and “cutting ” each cross-transition in two. Since every cross-transition represents a message from one concurrent component ina swimlane to another concurrent component in another swimlane, we will attach a separate nonterminal label to both ends

of the cross-transition. Depending on the type of message represented by the cross-transition, and the pattern involved, we are using the following nonterminals for the “sending” and “receiving” end (i.e., sending role and receiving role).

<i>Pattern</i>	<i>Type of message</i>	<i>Sending role</i>	<i>Receiving role</i>
<i>Client/Server</i>	<i>Request</i>	<i>SendReq</i>	<i>RecReq</i>
	<i>Reply</i>	<i>SendReply</i>	<i>RecReply</i>
<i>Forwarding</i>	<i>Request</i>	<i>SendFReq</i>	<i>RecFReq1</i>
	<i>Forward</i>	<i>FwdReq</i>	<i>RecFReq2</i>
	<i>Reply</i>	<i>RepFReq</i>	<i>RecFReply</i>

Table 1 Nonterminal labels for Client/Server and Forwarding Patterns

Our thesis deals only with Client/Server and Forwarding pattern, however, the grammars defined here can be extended easily. Other design patterns can be added as needed in further work.

1. **<Activity Diagram>:= { <Partition > }₁ⁿ**
2. **<Partition>:= <Sequence>**
3. **<Sequence>:= <Basic Block><Sequence>**
4. **<Basic Block>:= <StateVertex> | <Transition> | <RVreq> |
<RecProcReply> | <RVFreq> | <RecFProcFwd> | <RecFProcRly> |
<Branch/Merge> | <Fork/Join> | <Empty> |**

5. $\langle RVreq \rangle := \langle SendReq \rangle \langle RecReply \rangle$
6. $\langle SendReq \rangle := CrossTransition$
7. $\langle RecReply \rangle := CrossTransition$
8. $\langle RecProcReply \rangle := \langle RecReq \rangle \langle Sequence \rangle \langle SendReply \rangle$
9. $\langle RecReq \rangle := CrossTransition$
10. $\langle SendReply \rangle := CrossTransition$
11. $\langle RVFReq \rangle := \langle SendFReq \rangle \langle RecFReply \rangle$
12. $\langle SendFReq \rangle := CrossTransition$
13. $\langle RecFReply \rangle := CrossTransition$
14. $\langle RecFProcFwd \rangle := \langle RecFReq1 \rangle \langle Sequence \rangle \langle FwdReq \rangle$
15. $\langle RecFReq1 \rangle := CrossTransition$
16. $\langle FwdReq \rangle := CrossTransition.$
17. $\langle RecFProcRly \rangle := \langle RecFReq2 \rangle \langle Sequence \rangle \langle RepFReq \rangle$
18. $\langle RecFReq2 \rangle := CrossTransition$
19. $\langle RepFReq \rangle := CrossTransition$
20. $\langle Branch/Merge \rangle := \langle Branching \rangle \{ \langle Sequence \rangle \}_1^n \langle Merging \rangle$
21. $\langle Fork/Join \rangle := \langle Forking \rangle \{ \langle Sequence \rangle \}_1^n \langle Joining \rangle$

The grammar rules are, in general, rather simple, as most of them contain sequential subgraphs in the right-hand side. The most interesting ones are the rules 18 for the branch-merge structure and 19 for the fork-join structure. The subgraphs corresponding to the left-hand side nonterminals are represented in Figure 3.6 and Figure 3.7, respectively.

Figure 3-6 <Branch/Merge> structure

Figure 3-7 <Fork/Join> structure

3.2.2.2 Transformation Step 2.1

The goal of step 2.1 is to overlay the behavior of the architectural patterns extracted in step1 from the high-level collaboration diagram over the activity diagram, and verify whether the communication between concurrent components is consistent with the pattern. This is done by traversing the graph, by identifying the cross-transitions between swimlanes, and by checking if they follow the protocol defined by the pattern. Then the

Step 2.1 for the Client/Server with blocking client is as follows:

- Based on algorithm step1, we know that the design pattern involved between the active objects *client* and *server* is the Client/Server pattern. This means that we expect the following behaviours: the Client sends a request to the server, which process it and replies later.
- Traverse the activity diagram. If the client blocks after sending the message to the server, it belongs to case1 (i.e., blocking client).
- According to the protocol defined by the Client/Server pattern, a request message and a replying message are expected between *client* and *server* components.
- If the activity diagram follows the protocol defined by the pattern, during the traversing process, we attach appropriate nonterminal symbols to the elements identified. In Figure 3.6, we attach SendReq and RecReq to the cross-transition 1 that represents the request from *client* to *server*. Similarly, we attach SendReply and RecReply to the cross-transition 2 that represents the reply from *server* to *client*.
- If the activity diagram does not follow the protocol defined by the pattern, an error message will be displayed to the users.

Case2: Client/Server pattern with non-blocking client:

Figure 3-9 Step 2.1 applied to Client/Server with non-blocking client

Figure 3.9 illustrates algorithm step 2.1 applied to the Client/Server pattern with non-blocking client. The only difference from the case with blocking client is on the client side. The traversal of the activity diagram will check whether the client follows the communication protocol from the pattern: after sending the request, does some work and then receives the reply. The same nonterminal symbols are attached to the two cross-transitions as in the previous case.

Figure 3-10 Step 2.1 applied to Forwarding pattern

Forwarding pattern: According to Figure 3.10, the active object *client*, *server1* and *server2* participate in Forwarding pattern. In this pattern, *client* sends a request to *server1*, which plays the role of *ForwardingServer* and forwards the request to *server2*, which in turn plays the role of *ReplyingServer* and sends the reply back to the *client* object.

The verifying process:

- Based on the algorithm step2.1, we know that the design pattern among the active objects *client*, *server1* and *server2* is Forwarding pattern.
- Three messages are expected according to protocol defined by the Forwarding pattern: *client* should send request to *server1*, *server1* should forward the request to *server2* and finally, a reply message should be sent from *server2* to *client*.
- Traverse the activity diagram and check whether it contains these 3 messages.
- If the activity diagram follows the protocol defined by the pattern, we attach appropriate labels to the subgraphs identified. In the Figure3.10, we attach sendFReq and RecFReq1 to cross-transition 1, representing the request from *client* to *server*. Similarly, we attach ForFReq and RecFReq2 to the cross-transition 2 representing the forwarding message from *server1* to *server2* and finally we attach RepFReq and RecFReply to cross-transition 3 which identify the reply message from *server2* to *client*.
- If the activity diagram does not follow the protocol defined by the pattern, an error message will be displayed to the users.

3.2.2.3 Transformation step 2.2

Parse separately the subgraph within each swimlane. This will deal with sequence, alternative branches and fork/join structures. Considering the entries of the specific task, each task starts with zero entries. A new entry is added to this task if a new communication request is received from other task. When the new entry is created, it enters its Phase1 scope. When the server sends a reply back to the client, or forwards it to

another server, it moves to the second phase within the entry.

The graph transformation rule and parsing tree for a Client/Server pattern with blocking client are illustrated in Figure3.11, whereas those for a Client/Server pattern with non-blocking client are illustrated in Figure3.12.

Figure 3-11 Algorithm Step 2.2 applied to Client/Server with blocking client

In our case, swimlanes in the activity diagram correspond to the active software objects in the deployment diagram and high-level collaboration diagram. According to Figure3.11a,

there are two swimlanes *Client* and *Server*, which correspond to the software components with the same name given in the high-level collaboration diagram and deployment diagram. Our algorithm will parse the swimlane separately from left to right. So the *client* swimlane is processed first. We draw the parsing tree according to the graph grammar. This parsing tree consists of three parts: <Sequence>, <RVreq> and <Sequence>. The rule for the leftmost nonterminal <Sequence> is applied first until the terminal symbol StateVertex A is reached. Then the nonterminal <RVreq> will be processed by applying the appropriate rules and finally the right most nonterminal <Sequence> will be parsed. When processing the server partition, it can be seen that four nonterminal parts, namely <Sequence>, <RecProcReply>, <PostService>, <Sequence> are identified on this side. These four parts will be parsed one by one from leftmost to rightmost. The <Sequence> within <RecProcReply> corresponds to the phase 1 of service and is found between the “join” Pseudostate marking the receiving of the client request and the “fork” Pseudostate marking the sending of the reply. The other subgraph <PostService> corresponds to the second phase of service, and is found between the “join” marking the sending of the reply and either a “waiting” state for a new request or the “undefined” state we use by default to mark the end of the respective component behaviour on behalf of the current scenario. (This subgraph may very well be empty). During algorithm step2.1, RecReq and SendReply have already been labeled in cross-transition 1 and 2. According to these labels, we can identify the type of the subgraphs and then choose the appropriate rule to apply. Their partial parsing tree corresponding to the Client and Server partitions are shown in Figure 3.11 b and Figure3.11c. Figure3.11d shows the generated LQN model elements.

Figure 3-12 Algorithm Step 2.2 applied to Client/Server with non-blocking client

Figure 3.12 shows the Graph transformation rule, the corresponding parsing tree and LQN model for the client/server pattern with non-client blocking. Two partitions exist in this pattern: *Client* partition and *Server* partition. In terms of our algorithm, the *Client* partition will be processed first and then the *Server* partition. Client parsing tree shows

three nonterminal symbols, <Sequence>, <Fork/Join> and <Sequence> on the right. These nonterminals will be processed one after another. When the <Fork/Join> nonterminal is parsed, the left side branch<Sequence> will be parsed first and then the right side nonterminal <Sequence>. For simplifying the figure, we have not shown in the parsing tree the terminal symbols <Forking> and <Joining> Pseudostate. When creating the corresponding LQN model shown in Fig3.12.c, the Fork/Join part is mapped to fork/join activity part Branch1 and Branch2. The RVreq will be mapped to the synchronous call from one of the activities within Branch2 part to serverTask. In the *Server* partition, there are four nonterminal symbols: <Sequence>, <RecProcReply>, <PostService> and <Sequence> on the right side. Among these nonterminals, <Sequence> part of <RecProcReply> corresponds to the first phase of service and <PostService> corresponds to the second phase of service.

The graph transformation rule and parsing tree for a Forwarding pattern with one forwarding server are illustrated in Figure.3.13.

According to this Figure, three partitions are involved into this pattern: *client* partition, *server1* partition and *server2* partition. And parsing order is from left to right. When the *client* partition is parsed, it has three parts: <Sequence>, <RVFreq> and <Sequence> which will be processed from left to right one after another as we analyzed in the previous pattern. During the process of *server1* partition, it has four nonterminals <Sequence>, <RecFProcFwd>, <PostService> and <Sequence>. When receiving the request from client, *server1* creates a new entry and enters the first phase of the entry. After forwarding the request to *server2*, this entry enters its 2nd phase. In other words, <Sequence> part within <RecFProcFwd> corresponds to the first phase of service and <PostService>

corresponds to the second phase. The server2 partition has three parts: <Sequence>, <RecFProcRly> and <Sequence>. Similarly, when receiving the forwarding request from server1, a new entry is created for server2, and the scope between the receiving request to replying back to client marks the first phase of the newly created entry.

Figure 3.13 c gives the constructed LQN model.

a) Graph transformation rule for the Forwarding pattern

b). Parsing tree for Server1 partition

c.) Constructed LQN model

Figure 3-13 Algorithm Step 2.2 applied to Forwarding pattern

4 Transformation from UML to LQN Model at metamodel level

Chapter 3 has discussed the algorithm for the automatic transformation from UML model to LQN Model at the UML notation level. Chapter 4 will describe the transformation process from the UML metamodel point of view and give the concrete pseudocode of the main methods used in the implementation.

4.1 Conceptual Description

4.1.1 UML metamodel and Novosoft UML API

As described in section 2.3.1, the UML metamodel is defined as one of the layers of the four-layer metamodeling architecture. It is regarded as being an instance-of the MOF residing at the M2 level. The official version of the UML metamodel at the time of the thesis research was UML 1.3 (OMG released the latest version of UML 1.4 in September 2001). The metamodel concepts and semantic constructions are described in chapter 2. The metamodel referred in the rest of the thesis is the version UML 1.3.

4.1.1.1 UML 1.3 metamodel

The UML metamodel is managed by organizing it into logical packages. These packages group metaclasses that show strong cohesion with each other and loose coupling with metaclasses in other packages.

The top-level packages are shown in Figure 4.1:

Figure 4-1 Top-Level Packages

The foundation and Behavioral Elements packages are further decomposed as shown in Figure 4.2 and Figure 4.3

Figure 4-2 Foundation Package

Figure 4-3 Behavioral Elements Package

4.1.1.2 UML 1.3 physical metamodel

In addition to UML metamodel mentioned above, OMG proposed the UML physical metamodel, which is more clear for realization and more practical. The physical metamodel is the representation of the UML semantics abstract syntax, but it simplifies the UML metamodel. Minor modifications and changes are introduced as needed to support the generation of an XMI DTD in the UML physical metamodel.

4.1.1.3 Novosoft UML metamodel

Novosoft UML metamodel is a base for creating Novosoft UML API. It has minor differences from UML 1.3 physical metamodel. These modifications are adopted to allow the model to be mapped unambiguously to Java language, but it follows UML physical metamodel very closely [NSUML99].

4.1.1.4 Novosoft UML API.

Novosoft UML API is a modern, high-quality Java-based software, supporting user's UML models, that is, it is an implementation of UML metamodel. It consists of interface, classes, attributes and methods, which supports UML and XMI standards. Two important classes are `XMIReader` and `XMIWriter` which are used to read and write model according to the XMI format. The version of Novosoft UML API used in our transformation application is 0_4_19 that can be download from [NSUML99]. The API mentioned in the rest of thesis will refer to Novosoft UML API unless otherwise specified.

4.2 UML Metamodel Representation for the Related Diagrams

It has been mentioned that UML metamodel is the base for the UML physical metamodel, Novosoft UML metamodel and Novosoft UML API. This section summarizes the metamodel elements used to represent deployment diagram, collaboration diagram and activity diagram [UML 1.3] by using the NSUML API [NSUML99]. These elements correspond to the code used in our implementation.

4.2.1 Deployment Diagram

Deployment diagrams show the configuration of run-time processing elements and the software components, processes and objects that live on them. Software component instances represent run-time manifestations of code units.

Fig 4.4 shows the main model elements for the deployment diagram, which correspond to the code used in our implementation.

Figure 4-4 Part of MetaModel Representation for a Deployment Diagram

ModelElement:

A model element is an element that is an abstraction drawn from the system being modeled.

In the metamodel, a ModelElement is a named entity in a Model. It is the base for all modeling metaclasses in the UML. All other modeling metaclasses are either direct or indirect subclasses of ModelElement.

NodeInstance:

A node instance is an instance of a node. A collection of component instances may reside on the node instance.

In the metamodel, NodeInstance is an Instance that originates from a Node. Each ComponentInstance that resides on a NodeInstance must be an instance of a Component that resides on the corresponding Node.

ComponentInstance:

A component instance is an instance of a component that resides on a node instance.

In the metamodel, a ComponentInstance is an Instance that originates from a Component. It may reside on a NodeInstance.

4.2.2 Collaboration Diagram

A collaboration can be used to specify the implementation of design constructs. The collaboration maps into a Collaboration. For each class symbol attached by an arrow to the pattern occurrence symbol, the corresponding Class is bound to the base association target of the ClassifierRole in the Pattern with the name equal to the name on the arrow.

Fig 4.5 shows the main model elements for the collaboration diagram that correspond to the code used in our implementation.

Figure 4-5 Part of MetaModel Representation for high level collaboration diagram

Collaboration:

A Collaboration can be used to specify the high-level architectural patterns. In this case, it can contain a set of classifierRoles with the name equal to the name on the arrow from Collaboration to the actual Classes that occur in the specific use of the pattern.

ClassifierRole:

A classifier role is a specific role played by a participant in a collaboration. It specifies a restricted view of a classifier.

In the metamodel, a ClassifierRole specifies one participant of a Collaboration.

Classifier:

A classifier here is the view of the related ClassifierRole. It is the actual Classes that occur in the particular use of the pattern.

4.2.3 Activity Diagram

An activity graph is a special case of a state machine that is used to model processes involving one or more classifiers. It shares many metamodel elements with state machine. The metamodel Representation for Activity diagram is shown in Figure4.6.

Figure 4-6 Part of MetaModel Representation for activity diagram

ActivityGraph:

An activity graph is a special case of a state machine that defines a computational process in terms of the control – flow and object – flow among its constituent actions. In the activity diagram, all (or at least most) of the states are action or subactivity states and all (or at least most) of the transitions are triggered by completion of the actions or subactivities in the source state[UML1.3].

State:

A state is an abstract metaclass that models a situation during which some invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. It can also model dynamic conditions such as the process of performing some activity [UML1.3].

CompositeState:

A composite state is a state that contains other state vertices (states, pseudostates, etc.). Any state enclosed within a composite state is called a substate of that composite state. A composite state can be decomposed into concurrent substates or into mutually exclusive substates [UML1.3].

StateVertex:

A StateVertex is an abstraction of a node in a statechart graph. In general, it can be the source or destination of any number of transitions.

Transition:

A transition is a directed relationship between a source state vertex and a target state vertex.

Pseudostate:

A pseudostate is an abstraction that encompasses different types of transient vertices in the state machine graph. Typically, they are used to connect multiple transitions into more complex state transitions paths.

The following pseudostate kinds are defined:

- An initial pseudostate represents a default vertex that is the source for a single transition to the default state of a composite state. There can be at most one initial vertex in composite state.
- Fork vertices serve to split an incoming transitions into two or more transitions terminating on orthogonal target vertices. Join vertex serves to merge several transitions emanating from source vertices in different orthogonal regions.
- Branch vertices can be used to split an incoming transition into multiple outgoing transition segments with different guard conditions. Conversely, Merge vertices can be used to converge multiple incoming transitions into a single outgoing transition representing a shared transition path.

Partition:

A partition is a mechanism for dividing the states of an activity graph into groups.

FinalState:

A final state is a special kind of state signifying that the enclosing composite state is completed. A final state cannot have any outgoing transitions.

4.3 Reading XML files

Figure 4.7 shows that the input of our implementation is the XMI files containing UML models. The first step in our implementation is to read the XMI files. These files will be parsed by the XMI reader provided by Novosoft UML API. Novosoft UML API supports XMI standard. It can read and write model according to XMI format. The `XMIReader` provided by Novosoft UML API makes use of JavaSoft's Java API for XML Parsing (JAXP) which can be available for download at [JAXP]. The `XMIReader` class is a huge java file consisting of more 20,000 lines of source code. Its responsibility is to create Java objects (NSUML objects) from an XMI file. The two main standards that have emerged for creating XML parsers are the Simple API for XML (SAX) and the Document Object Model (DOM). JavaSoft's JAXP download contains implementation of both SAX and DOM parser. `XMIReader` class makes use of SAX standard.

SAX was created as a result of a mailing list discussion, and provides sequential, read-only access to the document's contents. That means, SAX does not provide any facility for modifying a document, and it does not allow you to examine an arbitrary portion of the document. The SAX Implementation in JAXP provides four interfaces as follows: `EntityResolver` interface, `DTDHandler` interface, `DocumentHandler` interface and `ErrorHandler` interfaces. It also provides the `HandlerBase` class which is used to implement the above interfaces. Since the `HandlerBase` class provides "dummy" implementations for each of the methods defined in the four interfaces, you can create a `HandlerBase` subclass that includes only implementations of the methods that you are interested in using. The `XMIReader` provided by NovoSoft UML is an example of the

HandlerBase subclass.

After parsing through the XMIRReader, the output is the internal data structure corresponding to the UML metamodel which actually is the input of our transformation algorithm discussed in the following section.

Figure 4-7 Flow of thesis implementation

4.4 Generate LQN structure

In chapter 3 was shown that the LQN structure is obtained from the high-level collaboration and deployment diagrams. Since we've already introduced the meta-model of these two diagrams in Section 4.2, the following part will present the global picture of how these internal meta objects can be used to perform transformation algorithm step1 from UML model to LQN structure.

4.4.1 Algorithm for structure transformation

Generating the LQN structure is done in two main substeps: step1.1 is to extract information from the deployment diagram and step1.2 is to obtain information from the

high-level collaboration diagram.

4.4.1.1 Step 1.1: Obtain information from the Deployment diagram

In algorithm step 1.1, the processors (the disks, the network), and the software components residing in the related processor are extracted from the meta-objects of the deployment diagram. Then the transformation method `extractFromDeployment()` will be applied to transform the above information into the software and hardware tasks and the arcs between different tasks.

Here is the algorithm for obtaining information from deployment diagram.

```
extractFromDeployment(Mmodel mmodel)
{
    Vector processorInfor;
    Vector taskInfor;

    While(mmodel has elements e)
    {
        if(e instanceof MNodeInstance)
        {
            create Processor object p;
            handle ScheDispline;
            handle Multiplicity
            add object p to Vector processorInfor;
        }
        if(e instanceof MComponentInstance)
        {
            create Task object t;
            handle RefFlag;
            handle Multiplicity;
            handle Processor;
            set initial value to EntryList;
            add object t to Vector taskInfor;
        }
    }
}
```

When extracting the information from the deployment diagram, the purpose is to get the information about Processor , the software task running on the related processor, and the

detailed information describing processor scheduling discipline, multiplicity, reference flag for the software, and hardware task etc.

So when traversing the meta objects created by the `XMIRReader`, our algorithm will create a related processor object each time a `NodeInstance` element occurs. After extracting the detailed performance information, the scheduling discipline and multiplicity for the newly created processor object can be set. The processor object will be mapped to the corresponding hardware task in the LQN model.

In contrast, a Task object will be created if a component instance is met when traversing the metaobjects. In terms of additional information, the Reference Flag, Processor, and multiplicity for this created Task object will be set. This task will be mapped to the software task when the LQN model is constructed.

4.4.1.2 Step 1.2: Obtain information from the high level collaboration diagram

When extracting the information from the high-level collaboration diagram, the purpose is to identify the concurrent/distributed high-level components and keep track of the design patterns applied to them. The design pattern information will also be used to identify the message type (synchronous, asynchronous, or forwarding) among the related active objects.

This part is done using the following algorithm:

```
extractFromCollaboration(MModel mmodel)
{
while(mmodel has elements e1)
{
    if (e1 instanceof MCollaboration)
```

```

{
    create a CollaborationObject o;

    getOwnedElements    from e1
    while(ownedElements has elements e2)
    {
        create (classifierRole, base) pair of e2 as one
        element of the hashtable for o;
    }
}
store CollaborationObject to the related Vector.
}

```

While traversing the high-level collaboration diagram, we keep track of the concurrent/distribute objects and identify the pattern applied to them. Each time a collaboration occurs, collaborationObject is created with the same name as the Pattern name. Then we store the object which is actually involved into the specific pattern and its role played in the pattern into the hashtable of the newly created collaborationObject. This information will be used to verify whether the activity diagram complies to the design pattern in algorithm step 2 and it will also be used to identify the message type between different active objects.

4.5 Generate LQN detailed elements

Section3.2.2 has illustrated the behaviour transformation algorithm from an activity diagram to LQN model at notation level. This section will describe these behaviour transformations in detail by making use of the UML metamodel.

From Chapter3, we already know that UML scenario models represented as activity

diagrams will be used to extract the LQN task detailed information such as the entry list for each of the task, the phases, the activities (if any) of the task and their execution time demands and visit ratio parameters from one phase of an entry to another entry or from activity to other entry. At the same time, the scenarios represented by the activity diagrams will be verified whether they are consistent with the high-level architectural pattern used in the system or not. So this part of the algorithm consists of mainly two steps: step2.1 is to verify the consistency between the activity diagram and the high-level design pattern, step2.2 is to obtain the detailed information about the LQN task.

The next two sub sections will give more details on algorithm step 2.1 and step2.2.

4.5.1.1 Transformation step 2.1

The goal of step 2.1 is to verify whether the scenarios represented by the activity diagrams are consistent with the high-level collaboration diagram. Algorithm step2.1 will make use of the information obtained algorithm step1.2: extract information from high level collaboration diagram.

4.5.1.1.1 Conceptual Description

The idea of algorithm step2.1 is to overlay the behavior of the architectural patterns extracted in algorithm step1.2 and verify the consistency between the activity diagram and the design pattern among the active objects.

While traversing the activity diagram, we create a CrossTransitionObject if a transition crosses the boundary of two partitions. This object contains the information about the cross transition element, the element's source partition name and target partition name .

It will be stored in an intermediate Vector. Then by use of the design pattern information obtained from algorithm step1.2, we will check if the crossTransition objects, which actually identify the message between swimlanes, follow the protocols expected by the specific design pattern. During the traversing, the appropriate labels will be attached to cross-transition ends(see table3.1). If the activity diagram satisfies all the design pattern protocols, the algorithm will go to the next part, which will obtain the detailed information. Otherwise, an error message will be displayed to the users. So far, our implementation only deals with two design patterns: Client/Server pattern and Forwarding pattern. Other design pattern such as Pipeline & Filters, Master/Slave, BlackBoard, etc will be extended in the future work.

4.5.1.1.2 Pseudocode

The following is the main part of algorithm step2.1 that is concerned with checking the consistence and how to identify the label to the cross-transition object:

```

while(collaborations)
{
    if(collaborations name is "client/server")
    {
        identify the clientName and serverName for this pattern;

        while(crossTransition objects  cto)
        {
            if (cto's source is clientName and target is serverName)
            {
                sendReq=true;
                Attach appropriate labels on the crossTranstion object.
            }
            else if(cto's target is clientName and source is serverName)
            {
                sendRep=true;
                Attach appropriate labels on the crossTranstion object.
            }
        }
    }
    else if(collaboration name is "forwarding")
    {
        identify the clientName, forwardServerName and replyServerName;
    }
}

```

```

while(crossTransition objects)
{
    get one object named cto;
    if(sourceName.equals(clientName))
    {
        if(targetName.equals(forwardingServerName))
        {
            sendFReq=true;
            Attach appropriate labels on the crossTranstion object.
        }
    }
    else if(sourceName.equals(forwardingServerName))
    {
        if (targetName.equals(replyServerName))
        {
            forReq= true;
            Attach appropriate labels on the crossTranstion object.
        }
    }
    else if(sourceName.equals(replyServerName))
    {
        if(targetName.equals(clientName))
        {
            sendFRep=true;
            Attach appropriate labels on the crossTranstion
            object.
        }
    }
}

}

if(all the flags set true )
{
    go to next phase2 algorithm;
}
else
{
    display error message.
}

```

4.5.1.2 Transformation step 2.2

This step parses the activity diagram and generates the detailed information about the LQN model such as the entries, phases and activities (if any), and their parameters (CPU demands and visit ratios).

The transformation in this step follows the rules listed here:

- The hardware tasks, and software tasks of the LQN model have been identified in the algorithm step1. Considering the entries of the specific task, each task starts with zero entries. A new entry is added to this task if a new communication request is received by this task from another task.
- When a new entry is created, it enters its Phase1 scope. When the server sends a reply back to the client, or forwards it to another server, it moves to the second phase within the entry.
- Activities are detected if the nonterminal <Fork/Join> or <Branch/Merge> is encountered. The related information about activities in the LQN model will be obtained by parsing <Fork/Join> or <Branch/Merge> nonterminals.
- Naming convention for the LQN generated elements:
 - Software tasks will have the same name as the UML software components.
 - Processors will have the same name as the node instance.
 - The LQN entries for every task is <task name>E_i (i is the index of entry list).
 - LQN activities will have the same name as the first action state.
- According to [Profile01], we know that the UML Performance Profile provides the facility of specifying execution parameters which can be used by modeling tools to compute predicted performance characteristics. But in our implementation, we process XML files produced by current UML tools, which obviously do not support the Performance Profile yet. Therefore, we have attached the tagged value by hand identifying the demand for each of the activity, the execution probability for the

activity (in the case of branching), the visit ratio.

4.5.1.2.1 Conceptual Description

The purpose of algorithm step2.2 is to decompose the original host activity graph into subgraphs described by simpler graph grammar which was discussed in section 3.2.2.1. Each subgraph corresponds to a partition and describes the behavior of a single component, dealing with sequence of scenario steps, alternative branches and fork/join structure. It is done by implementing a top-down parser with recursive methods.

The “Top-down parsing” indicates that we start at the root of the parsing tree and work our way down to the leaves. At each stage in the partially completed parsing tree, our algorithm applies a rule to the leftmost nonterminal which is currently a leaf.

Our parsing begins with the activity diagram representing the scenario. That means, the root of the parsing tree is the nonterminal symbol <activity diagram>. Each activity diagram consists of more than one partition. We will process the partitions from left to right, one by one. Each partition may contain sequence of scenarios, alternative branches and fork/join structures. These parts will be processed from top to bottom, and if existing in the same level, they will be processed from leftmost to rightmost.

4.5.1.2.2 Pseudocode

This section will show the main tree and most of the subtrees of the recursive descent parser and will give the respective Pseudocode and the rules that are generating the subtrees.

The main tree shown in Figure 4.8 is fairly simple, it begins with the <activity diagram>

and processes the contained <partition> one by one from left to right.

Figure 4-8 The main tree for the recursive descent parser

Here is the algorithm for parsing the Activity Graph

- initialize new model and activityGraph;
- Get all partitions
- Loop For all partitions
 - processPartition(MPartition mp)
 - postProcess
- finalize activityGraph and model;

The <partition > subtree starts with <Sequence> which further decomposes into two parts:

<Basic Block> and <Sequence> again.

Figure 4-9 The subtree <partition>

When processing <partition>, we will apply the appropriate rule for the <Basic Block> in terms of the type of nonterminals it contains. In the implementation, the choice of the rule to be used is determined by the label on the crossTransition object and /or the pseudostate kind. According to the grammar, the possible labels for the crossTransition Object are: SendReq, RecReply, RecReq, SendReply, SendFReq, RecFReply, RecFReq1, FwdReq, RecFReq2, and RepFReq. The possible labels for the pseudostate object are : branch, merge, fork, and join.

According to rule 4 from the grammar, a <Basic Block> may contain the following nonterminals: <RVreq>, <RecProcReply>, <RVFReq>, <RecFProcFwd>, <RecFProcRly>, <Branch>, <Fork/Join>. These nonterminals are discussed in the following one by one.

If the label of sendRoleName in the crossTransition object is SendReq, that means the <RVreq> is encountered, which represents the sending of a request by a client and the receiving of the reply. The rule for <RVreq> is straightforward:

$$\langle RVreq \rangle := \langle SendReq \rangle \langle RecReply \rangle$$

In this case, we only need to keep track of the two crossTransition objects: <SendReq>, <RecReply> which will be used to identify the phase scope for the entry.

There is another Client/Server case, where the client is non-blocking (see Figure 3.10). This case is represented by a <Fork/Join> nonterminal that represents a subgraph with two branches contained between a fork and a join pseudostate. As shown in Figure 3.10, one branch represents the activity of the client after sending the request and before accepting the reply. The second branch represents a <RVreq> nonterminal which represents the

sending of a synchronous request and receiving of the reply. Such a structure will generate a LQN task with activities, as shown in Figure 3.10c

On the server side, if the receiving label is <RecReq>, the following rule should be applied :

$$\langle \text{RecProcReply} \rangle := \langle \text{RecReq} \rangle \langle \text{Sequence} \rangle \langle \text{SendReply} \rangle$$

We have mentioned earlier that when receiving a new request from other task, a new entry is created for the task. This entry starts with its first phase. The scope of phase1 is in between receiving the request from the client and sending the reply to the client as shown in Figure 3.10a (or forwarding request to next server as in Figure 3.11a). So in the above rule, <RecReq> and <SendReply> will be used mark the beginning and ending of the first phase.

When parsing the nonterminal <Sequence>, it will recursively apply the rule for <Sequence>. The <Sequence> will be eventually decomposed into subtrees which will be processed one by one from left to right.

Another design pattern processed by the parsing algorithm is the Forwarding pattern. On the client side, if the sending label is SendFReq, <RVFReq> will be processed. From the client point of view, this case is the same as sending a synchronous call, so its rule is similar to <RVreq>.

$$\langle \text{RVFReq} \rangle := \langle \text{SendFReq} \rangle \langle \text{RecFReply} \rangle$$

On the server side, if the label of the receiving role is <RecFReq1> or <RecFReq2>, the next nonterminal being parsed is <RecFProcFwd> or <RecProcReply>. Because these

two cases are very similar to <RecProcReply>, we will not repeat the explanation here.

When processing the nonterminal <Sequence> for either client or server, we may find structures such as <Fork>, <Join>, <Branch>, and <Merge>. We need to identify exactly which one of these cases was encountered, so we will check if the current statevertex is one of the above pseudostates. If it 's the <Fork> state that means a <Fork/Join> nonterminal is waiting to be processed. We parse this <Fork/Join> until reaching the corresponding <Join>. Similarly, if the Branch state was found, then the next nonterminal that should be processed is <Branch/Merge>. The processing will end when the <Merge> subvertex is reached.

This approach deals with nested structures, but does not handle structures that are not strictly nested (such as forking in a swimlane and joining in another). This is a limitation of the current approach that needs future work. In our implementation, all the recursive functions needed for the descent parser are methods of a Java class named

Transformation. The pseudocode for processPartition is the following:

```
processPartition(MPartition mp)
{ get the start vertex;
//process the client side
if(start vertex is "initial")
{
    process vertexChain till find the first CrossTransition
    if(CrossTransition 's role is "SendReq.")
    {
        processRVreq();
    }
    else if (CrossTransition's role is "SendFReq")
    {
        processRVFReq();
    }
}
//process the server side except the case of creating the new thread.
else if (start vertex is "waiting")
{
    process vertex chain till find the first CrossTransition
    if(CrossTransition' s receiving role is "RecReq")
    {
```

```

        RecProcReply();
    }
    else if (CrossTransition' receiving role is "RecFReq1")
    {
        RecFProcFwd();
    }
    else if (CrossTransition' s receiving role is "RecFReq2")
    {
        RecFProcRly();
    }
}

```

Figure 4-10 subtree for <RVreq>

The pseudocode for processRVreq() is the following:

```

processRVreq()
{
    while(crossTransitionObject)
    {
        get one object c ;
        try to find the corresponding reply from the server to
        client;
    }
    return the reply transition object;
}

```

Figure 4-11 subtree for <RecProcReply>

The pseudocode for RecProcReply() is the following:

```

CrossTransitionObject RecProcReply()
{
    while(crossTransitionObject)
    {
        get one object c ;
        try to find the corresponding reply from the
        Replyingserver to client;
    }
    sequence();

    return the reply;
}
  
```

Algorithm for RecFProcFwd() and RecFProcRly() are similar to RecProcReply, so we will not give it here.

The pseudocode for Sequence() is the following:

```

Sequence()
{
    while(end of chain not reached)
    {
        get the first vertex v1;
    }
}
  
```

```

if( v1's name is " branch ")
{
    while(branches)
    {
        processBranches();
    }
    modify v1;
}
else if( v1's name is "Fork")
{
    while(forks)
    {
        processForks();
    }
    modify v1;
}
else
{
    get the next vertex assign it to v1.
}
}

```

The following is the pseudocode for processing branches. We will have two major cases here. One is that the branch has the request to other objects. If in this case, the processRVreq(), or processRVFreq. will be recursively called according to the lables we've met. Another case is that during branch processing, no request is encountered, in such case, we only need to call recursively Sequence () method.

```

processBranches()
{
    if this branch has request to other object
    {
        if(the request is RVreq)
        {
            processRVreq();
        }
        else if(the request is RVFreq)
        {
            processRVFreq();
        }
    }
    else
    {
        Sequence();
    }
}

```

The concept of `processFork()` method is similar to `processBranch()`, so we will not give its pseudocode here.

4.6 Practical issues related to the XML input files

Unfortunately, the UML tools used in this research do not support all the UML features provide in the standard. For example, Rational Rose does not support the following:

- Collaboration symbol (dashed ellipse).
- Software component in deployment diagrams.
- Object flow in activity diagrams.
- Tagged values.

The other UML tool used for the thesis, ArgoUml does not support the following:

- Collaboration symbol (dashed ellipse).
- Activity diagram swimlanes and object flow.

Due to these reasons, the XML files containing the UML models that are the input to our transformation program were generated by UML tools but had to be modified by hand in order to add the missing features. After modification, the files are still correct according to the XMI standard, and can be processed without problems by the `XMIReader` which is a part of the Novosoft UML library [NSUML99].

5 Case Study

The previous chapters have described the concept and algorithm of performing the transformation from UML model to LQN model. This chapter presents the application of the proposed UML to LQN transformation algorithm to a more substantial example: a Group Communication Server [Scratchley00].

5.1 Description of the Group Communication Server application

The Group Communications Server (GCS) application is to manage a set of shared documents and give users access to them. Each user can subscribe to a set of documents, unsubscribe from the specific documents, submit new documents, and retrieve the documents they are interested in.

In this chapter we will consider only the following four use cases.

1. Submitting a new document.
2. Subscribing to a document.
3. Unsubscribing from a document.
4. Retrieving the most recent version of a document.

Figure 5-1 Use Cases in the GCS application

5.2 Apply transformation algorithm to GCS application

5.2.1 Case 1: Subscribe Document & Unsubscribe Document

5.2.1.1 Use case 1: subscribing to a document

From Figure 5.2a, we know that two collaborations exist in the system design for this use case: Client/Server pattern between *UserT* and *MainProc* and also the Client/Server pattern between *MainProc* and *DocInf*. This information is obtained in algorithm step1.2, which extracts information from collaboration diagram and is used later to check if the activity diagram for this use case is consistent with its design patterns.

Figure 5.2b shows the deployment diagram for “Subscribe Document” use case. Only two nodes are of interest in the deployment diagram: ClientNode and the ServerNode. ClientNode contains the software component with the name of *UserT*, and ServerNode contains two software components named *MainProc*, and *DocInf*.

The process of subscribing to a document is very straightforward. After receiving a “subscribing document” request from the user, the *MainProc* residing on ServerNode does some processing as follows: first it retrieves the information about the document, then adds the requesting user to the list of subscribers for the specific document, finally after preparing an acknowledgement for the user, *MainProc* will send the acknowledgement back to the user.

Figure 5.2c, shows the activity diagram for this use case. It's easy to see that there are three partitions here which correspond to the three software components in the Deployment diagram. In algorithm step2, we check first the consistency between the activity diagram and the high level collaboration diagram and then decompose the activity

diagram into swimlane subgraphs and parse the subgraphs one by one using our parsing algorithm discussed in Chapter 3, 4.

Figure 5.2a High-level architecture for the "Subscribe" scenario

Figure 5.2b High-level architecture for the "Subscribe" scenario

Figure 5.2c Activity Diagram annotated with performance info for the Subscribe Document scenario

Figure 5-2 Group Communication Server: high-level architecture, deployment and activity diagram for "Subscribe" scenario

According to the design pattern information obtained from the high-level collaboration diagram, the Client/Server pattern occurs between *UserT* and *MainProc* components. That means a request is expected from *UserT* to *MainProc* , and a reply message is expected from Partition *MainProc* to *UserT* after some period of processing procedure. Traversing the related activity diagram, if the diagram follows the protocol expectation, the appropriate labels will be attached to the cross-transition ends. These labels as well as the terminal StateVertex (such as branch, merge, fork, join) will be used in the parsing algorithm as explained in section3.2.2.2 and section4.5.1.2. The same procedure is applied to the consistency check between *MainProc* and *DocInf* partition.

Figure 5.3, Figure 5.4, and Figure5.5 show the parsing tree for *UserT* partition, *MainProc* partition, and *DocInf* partition separately.

Figure 5-3 parsing tree for UserT partition

The subgraph for the *UserT* partition consists of three parts:

- **<Sequence>** which will be processed as StateVertex with the name of RequestInitial.
- **<RVreq>** has two children: **<SendReq>** identifying the request from *UserT* to

MainProc. <RecReply> identifying the reply from *MainProc* to *UserT*.

- <Sequence> on the right will be parsed as StateVertex named ReceiveAck.

These three nonterminals will be processed from left to right one by one by the parsing algorithm.

Figure 5.4 shows that four nonterminals will be processed one by one from left to right when parsing *MainProc* partition: after <Sequence> is processed as StateVertex named WaitingMainProc, the parser will process <RecProcRly> which will be decomposed further into <RecReq>, <Sequence> and <SendReply> on the right side. <PostService> will be parsed next, but in most cases, it may be an empty part. Finally, the rightmost nonterminal <Sequence> will be processed as the StateVertex named undefinedMainProc.

A new entry is created for the *MainProc* task when receiving a request from *UserT* and firstly this entry stays in its phase1 scope. After *MainProc* sends the reply to the *UserT*, this entry enters its second phase. The service time for each of the activity/state states, the visit ratio from *UserT* to the newly created entry named "Subscribe to a document" and other LQN parameters are attached as tagged values "by hand" because the current UML tools we used do not support the performance profile which is in the process of being defined by OMG.

The parsing tree for DocInf partition is very similar to that for *MainProc*. The *DocInf* is in the waiting state until it receives the synchronous call from *MainProc*. After processing the request, it sends the reply to the *MainProc*. The <RecProcRly> will be processed from <RecReq>, <Sequence> to <SendReply>. <Sequence> here will be mapped to

phase1 scope of the new entry created for task *DocInf*.

Figure 5-4 parsing tree for MainProc partition

Figure 5-5 parsing tree for DocInf Partition

Figure 5.6 gives the generated LQN submodel for “subscribing a new document”. The corresponding textual description file in the format expected by the LQN solver is given in Appendix “A”.

Figure 5-6 LQN model for Subscribe scenario

5.2.1.2 Use Case2: unsubscribe from the specific Document

The process of Unsubscribing from the specific document is very similar to subscribe to a specific document, so it is not described here. The Appendix “B” gives its LQN description file.

5.2.2 Case 2: Retrieve Documents & Submit Document

5.2.2.1 Use case 3: retrieving the most recent version of a document

The Group Communication Server accepts two classes of documents from its subscribers, private and public, each with its own access right. The documents are kept in two different files: the private documents on disk1 and the public ones on disk2.

Figure 5. 7a. shows the architectural patterns in which are the participating components involved in this use case. By applied algorithm step 1.2, information about collaborations can be obtained as follows: Forwarding collaboration between *UserT*, *MainProc* and *RetrieveProc*. Client/Server collaboration between *RetreiveProc* and *Disk1Proc* (*Disk2Proc*). This information will be used to do the consistency checking as we discussed earlier.

Figure 5. 7b gives the deployment diagram for the GCS application. This figure shows three processing nodes *UserT Node*, *Server1Node* and *Server2Node*. *UserT Node* is the processor for *UserT*, *Server1Node* for *MainProc* and *RetrieveProc*, and *Server2Node* for *Disk1Proc* and *Disk2Proc*. As described in section 4.2.1, there are two kinds of meta-objects which will be used in our transformation: *NodeInstance* and the related *ComponentInstance*. Each *NodeInstance* is mapped to a hardware task such as processor, disk or networking device, and the related *ComponentInstance* will be mapped to the software task in the LQN submodel. By applying algorithm step 1.1, the hardware devices, the software entity and the request arcs between different tasks can be obtained.

Figure 5.7a High-level architecture for the “Retrieve” scenario

Figure 5.7b Deployment diagram for the “Retrieve” scenario

Figure 5-7 Group Communication Server: high-level architecture, deployment diagram

The processing of a retrieve-document request occurs as follows

- A User process *UserT* sends a request for a specific document to the *MainPro* of *Server1Node*, which determines the type of request.
- *MainProc* will forward the request to another process named *RetrieveProc*. This process is responsible for retrieving the documents.
- 20% will be requests for private documents and 80% for public documents. In each case, *RetrieveProc* delegates the responsibility of reading the document to the corresponding disk process in *Server2Node*.
- After getting the document either from *Disk1Proc* or from *Disk2Proc*, *RetrieveProc* sends it back to the user.

The activity diagram with performance annotations is given in Figure 5.3c. Activity Diagram is used to extract the detailed information which will be transformed to corresponding parameters in the LQN model

Figure 5.8 shows that five swimlanes exist in the activity diagram for the Retrieve Document use case. The five swimlanes are: *UserT*, *MainProc*, *RetrieveProc*, *Disk1Proc* and *Disk2Proc*. Each of them will correspond to a LQN software task.

In the following part, we will describe how step2.2 is used to extract the useful information for constructing the LQN model.

We traverse the activity diagram and create a CrossTransition Object each time we met a transition crossing the boundary between two swimlanes. Then, from the information already obtained from the high-level collaboration diagram, we know that 3 collaborations are involved in this use case:

- Forwarding design pattern among *UserT*, *MainProc* and *RetrieveProc*, which means a request from *UserT* to *MainProc*, forwarding message from *MainProc* to *RetrieveProc* and replying message from *RetrieveProc* to the *UserT*.
- Client/Server pattern between *RetrieveProc* and *Disk1Proc*, which expects the sending message from *RetrieveProc* to *Disk1Proc* and finally a reply from *Disk1Proc* back to *RetrieveProc*.
- Client/Server pattern between *RetrieveProc* and *Disk2Proc*. The protocol is the same as Client/Server pattern between *RetrieveProc* and *Disk1Proc*.

If the activity diagrams for the Retrieve Document follow all the protocols, the activity diagram is correct from the architectural point of view. We can apply the parsing algorithm in the next step. Otherwise, an error message will be displayed to the users.

The purpose of step 2.2 is to divide the activity diagram into subgraphs in terms of partitions.

Fig.5.9 shows the parsing tree for partition *UserT*.

Figure 5-9 Parsing tree for the UserT partition

The parsing tree structure of the *UserT* partition is the same as for the use case “Subscribe to documents”. We will not repeat the description.

Figure 5.10 shows the parsing tree for the MainProcess partition:

Figure 5-10 Parsing tree for MainProcess

According to Figure 5.10, four parts exists in the parsing tree for the *MainProcess* Partition:

- <Sequence> on the leftmost which will be parsed as the terminal StateVertex named *WaitingMainProc*.
- Nonterminal <RecProcFwd> is identified and decomposed into <RecFReq1>, <Sequence>, and <SendFReq>.
 - <RecReq> will be parsed as the receiving request from *UserT* to *MainProc* partition.
 - <Sequence> will correspond to the phase1 of the newly created entry for *MainProc*.
 - <SendRep> is the forwarding message from *MainProc* to *RetriveProc* partition.
- <Sequence> on the right corresponds to the second phase of the newly created entry for *MainProcess* (This subgraph may very often be empty.)

As we mentioned before, a new entry is added to the task if a new communication request is received, and this entry starts in Phase1. When a server forwards the request to another server, the entry moves to the second phase within the entry. In our *MainProcess* task, *MainProcess* receives the communication request from the *UserT* task and so a new entry named "MainProcessE2" is created in the LQN model. The scope between receiving the request until forwarding the request to *RetriveProcess* forms the phase1 part of this newly created entry.

Figure 5-11 Parsing tree for RetriveProcess

Figure 5.11 shows the parsing tree for the *RetrieveProcess*. The high level structure of this parsing tree is similar to *MainProcess*. Both of them contain: <Sequence>, <RecProcFwd> or <RecProcRly>, <PostService>, and <Sequence> nonterminals. The most interesting part of this parsing tree which is also the emphasis of our analysis here is the <Branch/Merge> nonterminal. Two <Sequence> nested in this nonterminal, each of which follows a Rendezvous request to Disk1Proc or Disk2Proc. As we know from the description of the LQN model, activities are components that represent the lowest level of details in LQN. Activities are connected together to form a directed graph. Execution may branch into parallel concurrent threads of control, or choose randomly between different paths. Activities are detected if a conditional or non-conditional branching state is encountered. Here in *RetrieveProcess* partition, two conditional branches are detected, which will be mapped into the related activities when generating the LQN model.

The parsing trees for Disk1Process and Disk2Process are very simple, so we will not describe here.

The LQN submodel obtained by applying the algorithm is shown in Figure5.12.

corresponding LQN textural file can be found in Appendix “C”.

Figure 5-12 LQN submodel for Retrieve Document use case

5.2.2.2 Use case 4: submitting a new document

The process of Submitting a new document is similar to retrieve document use case, so it is not described here. The Appendix “D” gives its LQN description file.

6 Conclusions

6.1 What was done

This thesis proposes a graph-grammar based method for transforming automatically a UML model annotated with performance information into a Layered Queueing Network(LQN) performance model. The input to the transformation algorithm is an XML file that contains the UML model in XML format according to the standard XMI interface. The output is the corresponding LQN model description file, which can be read directly by the existing LQN solvers. The main part of the algorithm is a graph-grammar based method for generating LQN elements from activity diagrams. An important contribution of the thesis is the development of the graph-grammar and of its parsing algorithm.

The thesis implements the proposed method in Java to transform UML models to LQN model. The implementation is consistent with version 1.3 of the UML standard, and uses a free-source library named NovoSoft UML that implements the UML metamodel and its API.

Based on the main results of the thesis, a conference paper has been accepted : D.C.Petriu, H.Shen, "Applying the UML Performance Profile: Graph Grammar based derivation of LQN models from UML specifications", accepted at Performance TOOLS'2002, London, UK, April 2002 (to appear as a LNCS volume, Springer Verlag).

Our work is one step in a larger research project aiming at deriving performance models from UML models and integrating the results of performance analysis back to the UML

models. The thesis does the forward UML to LQN transformation , but does not attempt the backward LQN to UML transformation.

6.2 Future work

The following sections present some possibilities for future research.

6.2.1 Forward translation from UML to LQN

- Expand the set of architectural patterns that can be recognized in the software architecture and extend the algorithm for their translation to a performance model.

This implies the extension of the grammar used to verify if the software specification corresponds to the respective patterns.

- Deal with multiple UML diagrams describing behaviours for different scenarios. For now, only activity diagrams were considered. In the future work, the sequence and collaboration should also be accepted. This implies that a larger subset of the UML metamodel will be analyzed.
- Develop a better mechanism for dealing with exceptions, which will produce more uses-friendly error messages. For now, the thesis detects if the scenarios representing by activity diagrams do not correspond to the respective patterns, but do not help the users much to diagnose the causes of the error.

6.2.2 Backward translation from LQN to UML

As we mentioned earlier, this thesis is one step in a larger research project aiming at

deriving performance models from UML models and integrating the results of performance analysis back to the UML models. The thesis does the forwarded UML to LQN transformation , but does not attempt the backward LQN to UML transformation. The OMG performance profile provides for performance results (such as response time, throughput, etc.) to be displayed in the UML model. In order to realize the backward translation, future work should build an interchange mechanism that will take the results from the performance model solver and plug them back into the UML model.

APPENDIX "A"

#The constructed LQN submodel for "Subscribe Documents"
scenario

```
G "subscribe.lqn"

P 0

  p ClientNode f i
  p ServerNode f

-1

T 0

  t UserT r UserTE1 -1 ClientNode m 20
  t MainProcess n MainProcessE1 -1 ServerNode
  t DocInfProcess n DocInfProcessE1 -1 ServerNode

-1

E 0

s UserTE1 0 12.5 0 -1

Z UserTE1 0 1.5 0 -1 # client "think time"

s MainProcessE1 4.564E-5 0 0 -1

s DocInfProcessE1 1.0E-5 0 0 -1

y UserTE1 MainProcessE1 0 1 0 -1

y MainProcessE1 DocInfProcessE1 1 0 0 -1

-1
```

APPENDIX "B"

#The constructed LQN submodel for "unsubscribe Documents"
scenario

G "unsubscribe.lqn"

P 0

p ClientNode f i

p ServerNode f

-1

T 0

t UserT r UserTE2 -1 ClientNode m 20

t MainProcess n MainProcessE2 -1 ServerNode

t DocInfProcess n DocInfProcessE2 -1 ServerNode

-1

E 0

s UserTE2 0 12.5 0 -1

Z UserTE2 0 1.5 0 -1 # client "think time"

s MainProcessE2 4.564E-5 0 0 -1

s DocInfProcessE2 1.2E-5 0 0 -1

y UserTE2 MainProcessE2 0 1 0 -1

y MainProcessE2 DocInfProcessE2 1 0 0 -1

-1

APPENDIX "C"

#The constructed LQN submodel for "Retrieve Documents"
scenario

```
G "retrieve.lqn"
P 0
  p ClientNode f i
  p Server1Node f
  p Server2Node f
-1
T 0
  t UserT r UserTE3 -1 ClientNode m 20
  t MainProcess n MainProcessE3 -1 Server1Node
  t RetrieveProcess n RetrieveProcessE1 -1 Server1Node
  t Disk1Process n Disk1ProcessE1 -1 Server2Node
  t Disk2Process n Disk2ProcessE1 -1 Server2Node
-1
E 0
  s UserTE3 0 26 0 -1
  Z UserTE3 0 1.5 0 -1 # client "think time"
  s MainProcessE3 3.41E-5 0 0 -1
  A RetrieveProcessE1 readRequest
  s Disk1ProcessE1 2.58E-4 0 0 -1
  s Disk2ProcessE2 2.58E-4 0 0 -1
  y UserTE3 MainProcessE3 0 1 0 -1
  F MainProcessE3 retrieveProcessE1 1 0 0 -1
-1
A RetrieveProcess
  s readRequest 2.0E-7
  s RFD1Initial 1.0E-7
  s RFD2Initial 1.0E-7
```

```
s  sendFile 1.5E-5
y RFD1Initial Disk1ProcessE1 0.2
y RFD2Initial Disk2ProcessE1 0.8
:
readRequest->(0.2)RFD1Initial + (0.8)RFD2Initial
RFD1Initial + RFD2Initial ->sendFile
sendFile[RetrieveProcessE1]
```

APPENDIX "D"

#The constructed LQN submodel for "Submit Documents"
scenario

```
G "submit.lqn"

P 0

  p ClientNode f i
  p Server1Node f
  p Server2Node f

-1

T 0

  t UserT r UserTE4 -1 ClientNode m 20
  t MainProcess n MainProcessE4 -1 Server1Node
  t SaveProcess n SaveProcessE1 -1 Server1Node
  t Disk1Process n Disk1ProcessE2 -1 Server2Node
  t Disk2Process n Disk2ProcessE2 -1 Server2Node

-1

E 0

  s UserTE4 0 26 0 -1
  Z UserTE4 0 1.5 0 -1 # client "think time"
  s MainProcessE4 9.666E-5 0 0 -1
  A SaveProcessE1 writeRequest
```



```

s Disk1ProcessE2 2.00E-4  0 0  -1
s Disk2ProcessE2 2.00E-4  0 0  -1
y UserTE4 MainProcessE4 0 1 0 -1
F MainProcessE4 SaveProcessE1 1 0 0 -1
-1

A SaveProcessE1

s writeRequest  2.0E-7
s WTD1Initial  1.0E-7
s WTD2Initial  1.0E-7
s sendAck  1.8E-5
y WTD1Initial Disk1Process_submit 0.4
y WTD2Initial Disk2Process_submit 0.6
:
writeRequest->(0.4)WTD1Initial + (0.6)WTD2Initial
WTD1Initial + WTD2Initial ->sendAck
SendAck[SaveProcessE1]

```

References

- [Armstrong01] Armstrong, E., "Working with XML: The Java API for XML Parsing (JAXP) Tutorial", See <http://java.sun.com/xml/jaxp-1.1/docs/tutorial/>.
- [ArgoUML] See <http://argouml.tigris.org/>.
- [Amer01] Amer, H., "Automatic transformation of UML software specification into LQN performance models using graph grammar techniques", Ottawa. Thesis (M.Eng.), Carleton University, 2001.
- [Booch94] Booch, G., Object-Oriented Analysis and Design with Applications (2nd ed.), Benjamin/Cummings, Redwood City, 1994.
- [Cook+94] Cook, S. & Daniels, J., Designing Object Systems: Object-Oriented Modeling with Syntropy, Prentice-Hall, Hemel Hempstead, 1994
- [DuCharme99] DuCharme, B., XML:the annotated specification, Upper Saddle River, N.J., Prentice Hall. 1999.
- [Wong02] Eric C. Wong : "XMI-based Transformation of UML Interaction Diagrams to Activity Diagrams". Master thesis, school of Computer Science, Carleton University, January, 2002.
- [Franks+95] G.Franks, A. Hubbard, S. Majumdar, D.C. Petriu, J. Rolia, C.M. Woodside, "A toolset for Performance Engineering and Software Design of Client-Server Systems", Performance Evaluation, Vol. 24, Nb. 1-2, pp 117-135, November 1995.
- [Franks99] Grag Franks. Performance Analysis of Distributed Server Systems, PhD thesis, Department of Systems and Computer Engineering, Carleton Univeristy, 1999.
- [Rozenberg97] G.Rozenberg (ed), Hanbook of Graph Grammars and Computing by Graph Transformation, World Scientific, 1997.
- [Hess00] Hess, D. A., "Rational Rose Enterprise Edition", Tech update product info. See <http://techupdate.cnet.com/enterprise/0-6119586-723-3121217.html>
- [Jacobson+92] I.Jacobson. M. Christerson, P.Johnsson and G Overgaard. Object-ORIENTED software Engineering: A Use Case Driven Approach. Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [JAXP] JavaSoft Java API for XML Parsing, see <http://java.sun.com/xml>.
- [Leventhal+98] Leventhal, M., Lewis, D. & Fuchs, M., Designing XML Internet applications. Upper Saddle River, N.J., Prentice Hall PTR, 1998.
- [Maruyama99] Maruyama, H., XML and Java; developing Web applications. Reading, Mass.. Addison Wesley, 1999.

- [Martin99] Martin, T.A., Project cool guide to XML for Web designers. New York, John Wiley, 1999.
- [NSUML99] Novosoft UML API, see <http://www.novosoft-us.com/>.
- [Neilson+95] J.E.Neilson, C.M. Woodside, D.C. Petriu and S. Majumdar, "Software Bottlenecking in Client-Server Systems and Rendezvous Networks", IEEE Trans. On Software Engineering, Vol. 21, No. 9, pp.776-782, Sep. 1995.
- [OMG] Object Management Group : <http://www.omg.org>.
- [Profile01] OMG: "Response to the OMG RFP for Schedulability, Performance, and Time", OMG document number ad/2001-06-14, <http://www.omg.org/cgi-bin/doc?ad/2001-06-14>, June 2001.
- [Petriu2000] Petriu, D.C., Wang, X, "From UML description of high-level software architecture to LQN performance models", in Applications of Graph Transformations with Industrial Relevance AGTIVE'99 (eds. M.Nagl, A. Schuerr, M. Muench), LNCS 1779, pp. 47-62, Springer, 2000.
- [Kähkipuro01] "UML-Based Performance Modeling Framework for Component-Based Distributed Systems", in R. Dumke et al.(Eds): Performance Engineering, LNCS 2047, Springer, pp.167-184, 2001.
- [Petriu+2000] Dorina Petriu, Christiane Shousha, and Anant Jainapurkar: "Architecture-Based Performance Analysis Applied to a Telecommunication System", 2000.
- [Rolia+95] J.A. Rolia, K.C. Sevcik, "The Method of Layers", IEEE Trans. On Software Engineering, Vol. 21, Nb. 8, pp 689-700, August 1995.
- [Ramesh+2000] A Multilayer Client-Server Queueing Network Model with Synchronous and Asynchronous Messages", IEEE Trans. on Software Engineering, Vol. 26, No. 11, pp. 1086-1100, 2000.
- [Rumbaugh+91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W., Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs. 1991.
- [Reenskaug96] Reenskaug, T., Working with Objects, Manning, Greenwich, 1996.
- [Rose] Rational Software Corportion, See <http://www.rational.com/products/rose/index.jsp>.
- [Schema] XML alternative schema: <http://www.oasis-open.org/cover/schemas.html#W3CWorkingGroup>.
- [Scratchley00] Evaluation and Diagnosis of Concurrency Architectures by W.Grag Scratchley (Ph.D thesis), Carleton University, 2000.
- [Smith90] Smith, C.U., "Performance Engineering of Software Systems", Reading Mass., Addison Wesley, 1990.
- [Smith+01] C.U. Smith, L.G Williams, Performance Solutions: A Practical Guide to Creating responsive, Scalable Software, Addison Wesley, 2001.

- [Schurr90] Schurr, A., "Introduction to PROGRES, an Attributed Graph Grammar-based Specification Language", In: Nagl, M., (ed): Graph-Theoretic Concepts in Computer Science, Lecture Notes in Computer Science, Vol. 411, pp. 151-165, 1990.
- [UML1.3] OMG: Unified Modeling Language Specification, version 1.3, See <http://www.omg.org/cgi-bin/doc?formal/00-03-02>, March 2000.
- [UML1.4] OMG: Unified Modeling Language Specification, version 1.4, See <http://www.omg.org/cgi-bin/doc?formal/01-09-67>, Sep 2001.
- [Booch+99] Grady Booch, James Rumbaugh, Ivar Jacobson, "The Unified Modeling Language User Guide", Addison Wesley, 1999.
- [Woodside89] C.M. Woodside, "Throughput calculation for basic stochastic rendezvous networks", Performance Evaluation. Vol.9, Nb.2, pp. 143-160, 1989
- [Woodside+95] 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 on Computers, Vol. 44, Nb. 1, January 1995.
- [W3C] World Wide Web Consortium, See <http://www.w3c.org>.
- [Wang99] Wang, X., Deriving Software Performance Models From Architectural Patterns By Graph Transformation, M.Eng. thesis, Department of Systems and Computer Engineering, Carleton University, 1999.
- [XML1.0] eXtensible Markup Language (XML) version 1.0, Tim Bray, et al, W3C, 10 Feb 1998. See <http://www.w3c.org/TR/REC-xml>.
- [XMI1.0] OMG: XML Metadata Interchange specification, version 1.0, See <http://cgi.omg.org/cgi-bin/doc?formal/00-06-01>, 2000.
- [XMI1.1] OMG: XML Metadata Interchange specification, version 1.1, See <http://www.omg.org/cgi-bin/doc?formal/2000-11-02>, 2000.