FROM UML TO PERFORMANCE MODELS BY XML TRANSFORMATIONS

by

Ping Gu

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Ottawa-Carleton Institute for Electrical and Computer Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6

October 2006

Your file Votre référence
ISBN: 978-0-494-27098-1
Our file Notre référence
ISBN: 978-0-494-27098-1

ABSTRACT

Model Driven Architecture (MDA), the new approach to software development proposed by OMG, promotes the idea that software development should be based on models throughout the entire software lifecycle. In the context of MDA, it becomes important to have the ability to evaluate at an early stage the performance characteristics of UML models, in order ensure that the systems under development will meet their performance requirements.

This work proposes an XML-based transformation method of an annotated UML model into different performance models, which can be further analyzed with existing tools. The mapping between the input model and the output model is defined at a higher level of abstraction based on graph transformation concepts; whereas the definition of the transformation rules and algorithm uses lower level XML trees manipulations techniques, such as XMLgebra. The actual implementation of the proposed model transformation is based on eXtensible Stylesheet Language Transformations (XSLT). The input to the XSLT transformation program is an XML file that contains a UML design in XML format, produced by an existing UML/XMI tool according to the standard XML Metadata Interchange (XMI). The first transformation step converts the input UML model into an Intermediate Model (IM), which is also in XML format. The Intermediate Model contains only the necessary information for constructing a performance model, and was defined in this work based on the domain model of the UML Profile for Schedulability, Performance and Time. In a second transformation step, a performance model, such as LQN or CSIM, is generated from IM. As part of this thesis work, the proposed transformations from UML to performance models is applied to a case study and verification issues are discussed.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Professor Dorina C. Petriu, for providing an interesting topic, constant direction, valuable advice, and most of all the opportunity to work with great minds. I would also like to thank Professor C. M. Woodside and the PUMA group from Carleton University for inspiring discussions.

I am grateful for the support given by my parents and my wife during the course of this work.

Table of Contents

List of Figures

List of Tables

List of Acronyms and Symbols

ÆMPA	Architectural Description Language
AQNs	Augmented Queuing Networks
BSS	Building Security System
C++SIM	C++ Simulation Package
CASE	Computer Aided Software Engineering
CHAM	Chemical Abstract Machine
CLISSPE	CLIent-Server Software Performance Evaluation
COM+	Component Object Model
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CSIM	C Simulation Package
CSM	Core Scenario Model
DTD	Document Type Declarations
EA	Enterprise Architect
EJB	Enterprise JavaBeans Technology
EQN	Extended Queuing Networks
eXMLgebra	Extended XMLgebra
FCFS	First-Come, First-Served
GN	Gap Name
GSPN	Generalized Stochastic Petri Nets
IBM	International Business Machine
ICAD	Interactive Computer Aided Design System
ID	Identification
IM	Intermediate Model
IMM	Intermediate MetaModel
JAVASIM	JAVA Simulation Package
LHS	Left-Hand Side
LQN	Layered Queuing Networks
MC	Markov Chains
MDA	Model Driven Architecture

MOF	Meta Object Facility
OMG	Object Management Group
OOSE	Object-Oriented Software Engineering
PML	Performance Modeling Language
PN	Petri Nets
PROGRES	PROgrammed Graph Rewriting Systems
PUMA	Performance from Unified Model Analysis
QN	Queuing Networks
QoS	Quality of Service
QVT	Query/View/Transformation
RFP	Request for Proposal
RHS	Right-Hand Side
RSA	Rational Software Architect
SAXON	The XSLT and Xquery Processor
SimML	Simulation Modeling Language
SPA	Stochastic Process Algebras
SPE	Software Performance Engineering
SPT	Profile for Schedulability, Performance and Time
STPN	Stochastic Timed Petri Nets
TS	Transition System
TTM	Time-to-Market
UCMs	Use Case Maps
UML	Unified Modeling Language
VP-UML	Visual Paradigm for the Unified Modeling Language
W3C	World Wide Web Consortium
XACT	Microsoft Cross-Platform Audio Creation Tool
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XMLgebra	XML Algebra
XMLSpy	XML Editor
XPath	XML Path Language
XSL	XML Stylesheet Language
XSLT	eXtensible Stylesheet Language Transformations

CHAPTER 1: INTRODUCTION

1.1 MOTIVATION AND RESEARCH OBJECTIVES

Meeting performance requirements, such as response time and throughput, is a major concern for all kinds of software products with performance constraints, and especially for real-time systems. Software Performance Engineering (SPE) addresses performance issues throughout the whole software lifecycle and aims to insure that software products under development meet their performance requirements [118,120]. SPE uses predictive performance models to assess different design alternatives at an early stage, before major obstacles to performance are frozen in design and code. This can improve the quality of the final product by helping designers to make informed choices and tradeoffs early in the life cycle, when changes are not expensive and open alternatives still exist. The performance model can capture more system features and produce more accurate results as the product is evolving.

Despite the promises and successes of SPE, a large gap still exists between software development and the performance analysis domain. In order to help bridge this gap, the Object Management Group (OMG) adopted the UML Profile for Schedulability, Performance and Time (SPT) [78], which enables quantitative software performance analysis during the development process. Chapter 8 of the SPT Profile describes UML performance extensions via UML stereotypes, tagged values and constraints.

The SPE idea fits very well with the new vision proposed by OMG for software development known as Model Driven Architecture (MDA) [79]. MDA promotes the use of models throughout the entire lifecycle: from business modeling to system design, component construction, assembly, integration, deployment, management, and evolution. UML and other OMG standards play an important role in MDA. This change of focus from code to models in software development raises another problem for software designers, namely the need to be able to verify different functional and non-functional characteristics of UML models.

To conduct quantitative performance analysis of an annotated UML model, the architectural and performance information embedded in the UML model should be transformed first into a performance model, which can be then analyzed with the help of a performance analysis tool. In current practice, performance models are usually built by hand after a careful understanding and abstraction of the software specification, and then solved under different workload conditions in order to diagnose performance problems and to recommend design alternatives for performance improvement. The performance analysis cycle, when done properly from the early stages of design throughout all software development stages, may be time consuming and thus expensive. Automated techniques are therefore needed to ease and accelerate the process of building and solving performance models. The need for automation is inevitable as automation is cost-effective and rapid. If software performance models would be generated automatically, software designers could employ more easily the performance evaluation process into their software development cycle, and performance limitations could be identified and corrected from early development stages.

This thesis focuses on automatic transformation of annotated UML design specifications into performance models. The structure of the performance model is obtained from the UML model of the high-level software architecture that shows the concurrent components and their relationships, as well as from deployment information describing the allocation of software components to hardware devices. The behavior of the performance model is derived from key scenarios modeled as UML activity or interaction diagrams, showing the activities executed by different components and the flow of control/data in the system. Quantitative performance annotations added to the UML diagrams specify information such as processors, resources, processes, multiplicity, and CPU demands for every scenario step.

In order to assess the flexibility of the transformation approach proposed in the thesis, two types of performance models are considered as target models in this work: Layered Queuing Networks (LQN) [38,39,134] and CSIM [32,33] simulation models. The model transformation approach proposed in the thesis is based on two theoretical concepts: graph transformation principles [106,107] and XML transformations techniques, such as

XMLgebra [28,29,61,62]. The actual implementation of the proposed model transformation is based on eXtensible Stylesheet Language Transformations (XSLT) [128]. XMLgebra is a kind of algebra defined over the structure of XML data that provides a mechanism for the construction and deconstruction of XML data. It was proposed in [61,62] as a basis for introducing abstractions for XML processing that are necessary for web service applications into programming languages (e.g., Java). On the other hand, XSLT is a flexible language recommended by the World Wide Web Consortium (W3C) [128] for transforming XML documents into various formats.

The input to the XSLT transformation program is an XML file that contains a UML design in XML format, produced by an existing UML/XMI tool according to the standard XML Metadata Interchange (XMI) [83]. The first transformation step converts the input UML model into a so-called Intermediate Model (IM), which is also in XML format. The Intermediate Model contains only the necessary information for constructing a performance model, and was defined in this work based on the domain model of the UML Profile for Schedulability, Performance and Time (SPT) [78]. IM is similar to the Core Scenario Model (CSM) defined for the PUMA project [84,122,134], and was developed in parallel with CSM. In the second step, a performance model (e.g., either a CSIM compile-ready model (C/C++) or a LQN text descriptive file) is constructed from the Intermediate Model obtained in the previous step.

The proposed transformation used for the performance analysis of UML specifications is illustrated in Figure 1.1. The starting point is a software design represented as a UML model with performance annotations; a performance model will be obtained by applying the proposed transformation; performance analysis can be conducted by using an existing performance solver (e.g., LQNS analytical solver for LQN or CSIM18 simulation engine for CSIM). The performance evaluation results can be used to improve the UML design model and to correct performance limitations before the actual code implementation.

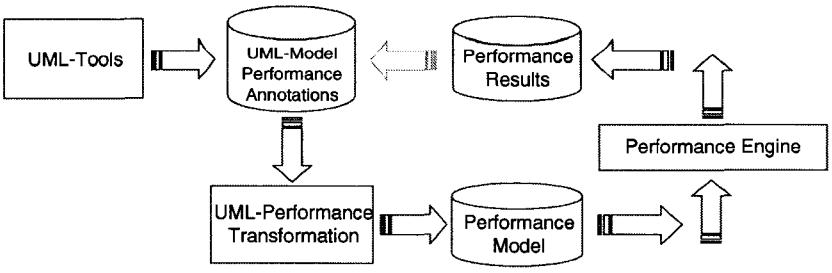


Figure 1.1: Performance analysis of UML models.

1.2 THESIS CONTRIBUTIONS

The main contribution of the thesis is a model transformation method that combines concepts from graph grammars (or graph transformations) and XML transformation techniques, such as XMLgebra. More specifically, the proposed method contains a low abstraction level for XML trees manipulations, similar to existing XML transformation techniques, and a high level of abstraction for defining the mapping between the input model and the output model, based on graph-grammar concepts.

XMLgebra [61,62] was introduced to resolve the problem of static validation of dynamically constructed XML documents used in web services. It is a theoretical foundation on top of which XML may become a first-class data type in any modern programming language (e.g., Java). It was implemented in the XACT framework [62]. The XMLgebra operations are based on the XPath and DTD standards, which in conjunction establish a powerful method for XML transformations. While XMLgebra and XACT focus on type checking of XML template manipulation, they do not support the definition of high-level transformations between the input model (i.e., UML) and the output model (i.e., performance model). This thesis selected and extended some desirable functions from XMLgebra, and built a tree-transformation technique at the metamodel level.

Looking in more detail at the higher layer of the proposed method, two parts can be distinguished: (i) a set of transformation rules mapping templates from the input DTD/schema (e.g., the UML metamodel) to the output DTD/schema (e.g., the IM metamodel), and (ii) a transformation algorithm deciding in what order to invoke the transformation rules over a given input tree for generating output subtrees, and how to

"glue" these subtrees together to construct the complete output tree. Conceptually, the "gluing" of subtrees is a label-based process, where the labels are node attributes of the output subtrees. The transformation rules (i) and algorithm (ii) are defined formally using an extension of XMLgebra from [61].

The proposed method is applied to define two transformations, one from UML to LQN and the other from UML to CSIM. This exercise is to assess how easy it is to specialize the proposed method to different output models. Each such transformation will accept as input a given UML software design with quantitative performance annotations, and generate the corresponding performance models (e.g., LQN or CSIM).

A more detailed list of thesis contributions is given below:

1. Extended XMLgebra to accommodate large XML templates. The extension includes a change of the notation for node numbering by using array expressions instead of strings, and the redefinition of the basic functions for template manipulations according to the new numbering expressions. The extended algebra is named eXMLgebra in the rest of the thesis.
2. Proposed a new XML-based model transformation method, which combines concepts from graph grammars and XML transformation techniques (more specifically, eXMLgebra). This method includes the following parts:
    a. Defined model transformation rules by using the eXMLgebra notation. Each rule maps a template compliant with the input DTD/schema (the left-hand-side) to a template compliant with the output DTD/schema (the right-hand-side). The rules define not only the structure of the subtrees to be generated, but also how to compute the node attributes. Some of these attributes are used as "gluing" labels.
    b. Proposed an eXMLgebra-based algorithm that traverses a given input XML template (tree) representing the input model, and applies the appropriate transformation rules. This step generates a set of XML subtrees compliant with the output DTD.

c. Proposed an eXMLgebra-based algorithm to construct the complete output template (tree) by gluing the subtrees obtained in the previous step. The final result represents the generated output model.

3. Implemented in XSLT the proposed transformation method, including the rules (a) and the algorithms (b) and (c).
4. Applied the proposed XML-based transformation method to generate different performance models from UML designs annotated with performance information. Such a transformation is performed in two steps: (a) from annotated UML models into an Intermediate Model (IM) that is defined based on the SPT profile, and (b) from IM to one of the target performance models. Two target performance models were chosen, LQN and CSIM, in order to assess the modularity and flexibility of the proposed transformation method.
5. Applied the proposed transformations from UML to performance models in various case studies and discussed verification issues.

1.3 THESIS ORGANIZATION

This thesis is organized as follows: Chapter 2 provides a review of the state of the art in transforming UML software specifications into various kinds of performance models. Chapter 3 describes the necessary background information for this research work and the tools used. Chapter 4 describes the XML-based model transformation method, and applies it to the transformation from UML to LQN, and from UML to CSIM. Chapter 5 introduces the implementation of the transformation method defined in eXMLgebra using XSLT. Chapter 6 discusses eight different test cases to verify the correctness of the transformation. At the end, Chapter 7 concludes the work done so far and discusses some aspects for future development.

CHAPTER 2: LITERATURE REVIEW

2.1 INTRODUCTION

For many years, the software industry adopted software development practices whose main concern was to get the products to market quickly. Such an approach over-emphasizes the time-to-market (TTM) strategy but may compromise the performance quality of the software products. This is because the functional design and implementation of the system were done without checking if the adopted solutions were able to meet the performance requirements. In many situations, the prototype failed to meet the performance requirements, resulting in an expensive fix or even a project failure.

Smith introduced the technique of Software Performance Engineering (SPE) [118] and proposed the use of quantitative methods and performance models to evaluate the performance of different design and implementation alternatives. SPE emphasizes the importance of doing performance analysis from the early stages of software development throughout the whole lifecycle. Performance modeling, if done in the early design stages, can reduce the risk of performance related failures by giving an early warning of problems. It also provides performance predictions under various environmental conditions or design alternatives. The performance model solution is compared with the requirements to reveal potential performance limitations. If the results do not meet the requirements, design parameters are changed, and the performance model is regenerated and resolved. This cycle continues until requirements are met. It is shown that early performance modeling has definite advantages, despite its inaccurate results, especially when the model and its parameters are continuously refined throughout the software lifecycle [118,120].

The Object Management Group (OMG) has defined the UML Profile for Schedulability, Performance and Time (SPT) [78]. The SPT Profile is aimed to enable the construction of models that can be used for quantitative performance predictions and schedulability analysis. SPT contains a Performance Profile, which provides mechanisms for capturing

performance requirements and for associating performance related QoS characteristics with the UML model. The domain model of the SPT Performance Profile contains basic abstractions used in performance analysis including scenarios, resource and workload. To conduct quantitative performance analysis on an annotated UML model, the architectural, behavioral and performance information embedded in the UML model must be translated into a performance model, before an existing performance analysis tool can be applied.

OMG's Model Driven Architecture (MDA) [79] is characterized by portability, interoperability and reusability. It facilitates integration and interoperability, and supports system evolution, which enables different applications to be integrated by explicitly relating their models. MDA also provides a set of guidelines for structuring specifications expressed as models and the mappings between those models. Meanwhile, the OMG has adopted the MOF 2.0 Query/View/Transformation (QVT) [80], whose goal is to specify a mechanism for transforming the elements of a model conforming to a particular metamodel into elements of another model that conforms to the same or to a different metamodel. Conceptually, QVT is based on the relational transformation approach, where the source and target models are described each by its own metamodel; a transformation defines relations (mappings) between different element types of the source and target metamodels. Such a mapping or transformation may be expressed as associations, constraints, rules and templates with parameters that must be assigned during the mapping.

Another known model transformation approach is based on graph transformation (i.e., graph rewriting of graph grammars) concepts. The graph transformation approach specifies how elements of one model are transformed into elements of another model in an operational manner, by using a set of transformation rules. In an article by Mens at al. [70,71], the graph-transformation and relational approaches are compared. While the former is based on matching and replacement, the latter is based on matching and reconciliation. Brief descriptions of QVT and graph grammar-based transformations are given section 2.5.1 and 2.5.2 respectively.

2.2 PERFORMANCE MODELS

Performance of hard real-time systems (where strict deadlines have to be met by various tasks for the software to be correct) is normally verified with schedulability analysis techniques, which are outside the scope of this thesis. The focus of this work is on the performance analysis of systems with stochastic properties, for which the following performance models are commonly used: Queuing Networks (QN) [68] and their extensions, Extended Queuing Networks (EQN) [30,41] and Layered Queuing Networks (LQN) [2,45,90]; Stochastic Timed Petri Nets (STPN) [60]; Stochastic Process Algebras (SPA) [20] and simulation models [12]. Figure 2.1 categorizes classes of performance models used for software performance evaluation, including simulation, analytical and schedulability analysis. The analytical models can be further divided into three main groups: queuing based, Petri Net based and stochastic process algebra.

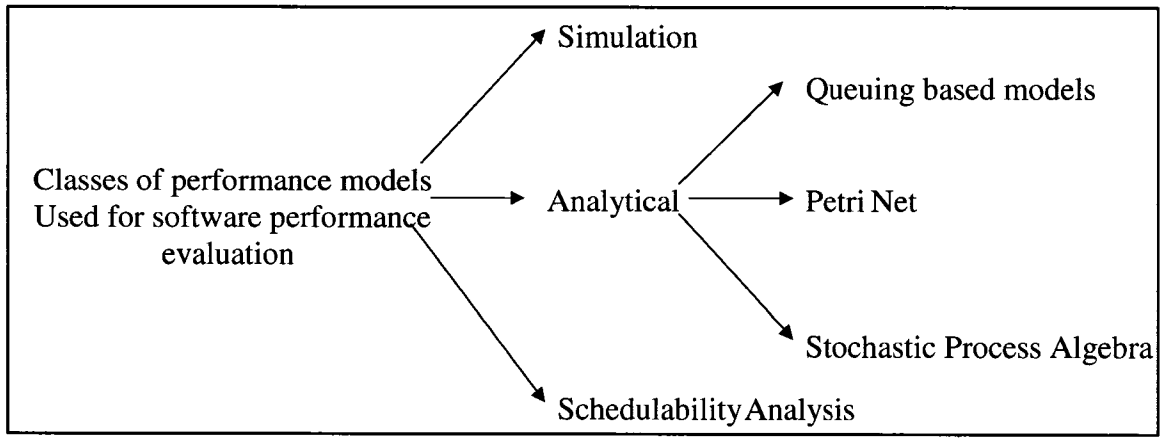


Figure 2.1: Classes of performance models used for software performance evaluation.

The performance models are used to evaluate a set of performance measures such as resource utilization, throughput and customer response time. Simulation is a widely used general technique, where the models "mimic" the structure and behavior of the actual system. Simulation models are less constrained in their modeling power and can capture more details than the analytical models. The main drawback of simulation is that it may require a high development and computational cost to obtain accurate results. In contrast, analytical models capture the essence of the modeled system as a set of mathematical equations. The models must satisfy a set of assumptions and constraints, which may limit

the ability of representing faithfully the actual system. The analytical solution of these performance models relies on stochastic processes, usually discrete-space continuous-time homogeneous Markov chains (MC).

Markov Processes [22]

Markov processes play an essential role in the analytical solution of the various classes of performance models. A Markov process is a stochastic process with a random variable X = $\{X(t) : t \in T\}$ where $X(t): T \times P \rightarrow E$ is defined on a probability space P and a time index set T with values in a state space E. Stochastic processes can be classified according to the state space, the time parameter, and the statistical dependencies among the variables X(t). The state space and the time parameter can be discrete or continuous, and dependencies among variables are described by the joint distribution function. A stochastic process is a Markov process if the probability that the process goes from state $s(t_n)$ to a state $s(t_{n+1})$ conditioned on the previous process history equals the probability conditioned on the last state $s(t_n)$ only. This implies that a process is fully characterized by the one-step probabilities, property known also as "memoryless property". Also, a Markov process is homogeneous when such transition probabilities are time independent. Due to the memoryless property, the time that the process spends in each state is exponentially distributed for the continuous-time process and geometrically distributed for the discrete-time Markov process, respectively. Markov processes can be analyzed under certain constraints to derive the stationary and the transient state probabilities.

Queuing Networks

A Queuing Network (QN) model is a collection of interacting service centers or servers representing system resources and a set of clients or jobs representing the users sharing the resources. It can be represented by a directed graph, whose nodes are service centers; after being served by a center, a client moves along an existing edge to another service center to wait for an another service. QN models are widely applied as system performance models to analyze resource-sharing systems [58,63,65,124]. The popularity of QN models for system performance evaluation is due to the relative high accuracy in performance results and the efficiency in model analysis and evaluation. A special class of QN named "product-form" network plays a particularly important role as efficient

algorithms can be applied to evaluate average performance measures. Algorithms, such as convolution and Mean Value Analysis that have a polynomial computational complexity, are used in most exact or approximate analytical methods and are widely applied for performance modeling and analysis.

An extension of the classical QN model, namely Extended Queuing Network (EQN) model, has been introduced in order to represent several interesting features of real systems, such as synchronization and concurrency constraints, finite capacity queues, memory constraints and simultaneous resource possession. EQN can be solved by approximate solution techniques [58,65]. Another extension of QN model is the Layered Queuing Network (LQN), which allows the modeling of client-server communication patterns in concurrent and/or distributed software systems [38,39]. The main difference between LQN and QN models is that a server in LQN may become a client (customer) of other servers while serving its own client requests. LQN models can be solved by analytic approximation methods based on standard methods for EQN with simultaneous resource possession and Mean Value Analysis. More information on LQN performance evaluation examples can be found elsewhere [39, 101].

Stochastic Process Algebra

Process Algebras [48] are widely known modeling techniques for the functional analysis of concurrent systems. They are described as collections of entities, or processes, executing atomic actions, which are used to represent concurrent behaviors that synchronize in order to communicate. Processes can be composed by means of a set of operators, which include different forms of parallel composition. Process Algebras provide useful techniques for the verification of system properties such as equivalences and pre-order, and can be used to describe systems at different levels of abstraction. Many notions of equivalence or pre-order are defined to study the relationship between different descriptions of the same system. Behavioral equivalences allow one to prove that two different system specifications are equivalent when "uninteresting" details are ignored, while pre-orders are suitable for proving that a low level specification is a satisfactory implementation of a more abstract one.

Stochastic Process Algebras (SPA) are extensions of Process Algebras, aiming at the integration of qualitative-functional and quantitative-temporal aspects into a single specification technique [48]. Additional information is added to actions by means of continuous random variables to allow evaluation of functional properties such as deadlock, throughput, waiting times, probability of timeout, and duration of action sequences of the modeled systems. More details on SPA can be found in various references [47,49,115].

Stochastic Timed Petri Net

Stochastic Timed Petri Net (STPN) is an extension of Petri Nets (PN). The underlying assumption in PN is that each activity takes zero time. Finite time durations with transitions and/or places are used in PN to answer performance related questions [1, 13,58]. The firing time of a transition is the time taken by the activity represented by the transition. In the stochastic timed extension, firing times are expressed by random variables.

The quantitative analysis of a STPN is based on the identification and solution of its associated Markov chain built on the basis of the net reachability graph. Non-polynomial algorithms exist for product-form STPN and many approximation techniques have been defined under further structural constraints [13]. Another extension of Petri Nets is called Generalized Stochastic Petri Nets (GSPN), which is continuous time stochastic Petri Nets that allow both exponentially timed and immediate transitions [1]. Immediate transitions fires immediately after enabling and have strict priority over timed transitions. Immediate transitions are associated with a normalized weight. In case of conflict, the choice of firing an immediate transition is done according to a probability.

Simulation Models

Simulation is actually the most flexible and general modeling technique, since any specified behavior can be simulated. A critical issue in simulation is the identification of the system model at the appropriate level of abstraction. Existing simulation tools, including CSIM, C^{++}SIM and JAVASIM [26,32,54], provide suitable specification

languages for the definition of simulation models, and a simulation environment to conduct system performance evaluation.

For instance, CSIM is a process-oriented discrete-event simulation package for use with C or C^{++} programs. It is implemented as a library of classes and procedures that provide all of the necessary structures and operations. The end result is a convenient tool that programmers can use to create simulation programs. A CSIM program models a system as a collection of CSIM processes that interact with each other by using the CSIM structures. The purpose of modeling a system is to produce estimates of time and performance. The model maintains simulated time to yield insight of the dynamic behavior in a modeled system.

It is unfeasible to review in detail all the performance models, as each of them is a huge research topic in itself. In this research work, LQN and CSIM performance models are selected due to their availability. Moreover, both models have gained considerable popularity because of their modeling power and ease of use.

2.3. SOFTWARE PERFORMANCE ENGINEERING

Performance is an important quality attribute of software systems. Combining software architecture design with a software performance model enables software designers to compare design alternatives, to test whether or not their software meets its intended performance requirements, and to avoid potential problems. Moreover, performance failures may result in damaged customer relations, lost productivity for users, loss of revenue, cost overruns due to tuning or redesign, and missed market windows. Stimulated by these reasons, the interest in relating transformation of software architecture designs to software performance analysis has been growing rapidly in the past years. Various approaches have been proposed to derive a performance model from software designs [3,4,14,15,18,19].

Software Performance Engineering (SPE) [119,130]:

Software performance engineering (SPE) presents a systematic, quantitative approach to construct software systems that meet performance requirements. It provides not only

principles for creating responsible software, but also incorporates models for representing and predicting performance. SPE uses the simplest possible model that identifies problems with the system architecture, design and implementation plans. The model is easily constructed and solved to provide feedback on whether the proposed software is likely to meet performance goals. As the software process progresses, the model is refined to more closely represent the performance of the emerging software. The precision of modeled results depends on the quality of the estimates of resource requirements. The estimations are usually difficult to obtain; however, SPE uses adaptive strategies, such as upper- and lower-bounds estimates and best- and worst-case analysis to manage the uncertainty.

SPE suggested two types of models that provide information for architecture assessment: the software execution model and the system execution model. The software execution model represents key aspects of the software execution behavior. It is constructed using execution graphs to represent workload scenarios. The graphs are hierarchical, with the lowest level containing complete information on estimated resource requirements. Solving the software model provides a static analysis of the mean, best- and worst-case response times. Software execution models are generally sufficient to identify performance problems due to poor architectural decisions. If the software execution model meets the requirements, the system execution model would then be constructed and solved.

The system execution model represents the computer resources as a network of queues. Queues represent components of the environment that provide some processing service, such as processors or network elements. Environment specifications provide device parameters. Workload parameters and service requests for the proposed software come from the resource requirements computed by solving the software execution model. The results of solving the system execution model identify potential bottleneck devices and correlate system execution model results with software components.

2.4 TRANSFORMING SOFTWARE DESIGN FOR PERFORMANCE ANALYSIS

Balsamo et al. [17] surveyed and compared various approaches for software design to performance model transformations. They introduced three comparative indices to evaluate different approaches.

- The integration level of the software model with the performance model: This integration level refers to the mapping provided by a given methodology between the software design artifacts and the performance model elements. A high level of integration indicates that the performance model has a strong semantic correspondence with the software model.
- The level of integration of performance analysis in the software lifecycle: The integration level indicates how precise and correct the performance analysis can be carried out to improve the software design.
- The methodology automation degree: This aspect refers to the degree of automation that the various approaches can provide.

There are a number of publications proposing different approaches to transform a software design to a corresponding performance model for evaluation [31,36,43,44,45]. These approaches can be grouped in accordance with the generated performance model. However, as this research work emphasizes the transformation process rather than the performance analysis, attention is given to the process of the software design transformation. Therefore, numbers of approaches are reviewed in accordance with different methodologies for deriving performance models from software architecture design.

2.4.1 Approaches Using Software Performance Engineering

Poor performance is often the result of problems in the architecture rather than in the implementation and the most performance failures are due to a lack of consideration of performance issues early in the development process in the architectural design phase. Smith and Williams proposed the use of software performance engineering (SPE) techniques to perform early assessment of software architectures and to determine if

performance objectives are met [118,120]. The use of SPE at the architectural design phase helps developers select a suitable architecture and its emphasis is in the construction and analysis of the software execution model, which is considered the target model of the specified software architecture and is obtained from the sequence diagrams. The class and deployment diagrams contribute to complete the description of the software architecture, but they are not involved in the transformation process. The SPE process requires additional information that includes software resource requirements for processing steps and computer configuration data.

Smith and Williams [119]:

The authors present a case study of an interactive computer aided design system (ICAD) to illustrate the derivation of a performance model from an object-oriented design aspect. The actual performance analysis is done by either approximate analytical methods or by simulation with a tool called SPE·ED that supports the SPE methodology. Starting from a set of scenarios described by message sequence charts, the authors derive the execution graphs that represent the software execution model. The QN model that represents the system execution model corresponds to various design architectures. The transformation from design architecture to performance model is done by hand. This transformation process requires a designer to have substantial knowledge of both software design and performance analysis in order to ensure that the performance model constructed is close enough to the original design.

Menascè and Gomaa [68]:

According to the SPE principles, the authors proposed a methodology based on CLISSPE (CLIent-Server Software Performance Evaluation), a language for the software performance engineering of client/server applications, for deriving queuing network performance models from software architecture specifications. Figure 2.2 illustrates the transformation step of the client/server software design or architecture specification to a performance model. Although the methodology does not explicitly use UML, the functional requirements of the system are specified in terms of use cases, and the system structure specification is analogous to a class diagram. The use cases, client/server software architecture specifications and mapping associating software components to

hardware devices, are used to develop a CLISSPE program specification. The CLISSPE system provides a compiler that generates a corresponding QN model. With the input performance parameters and appropriate solution methods, performance of the design architecture can then be evaluated.

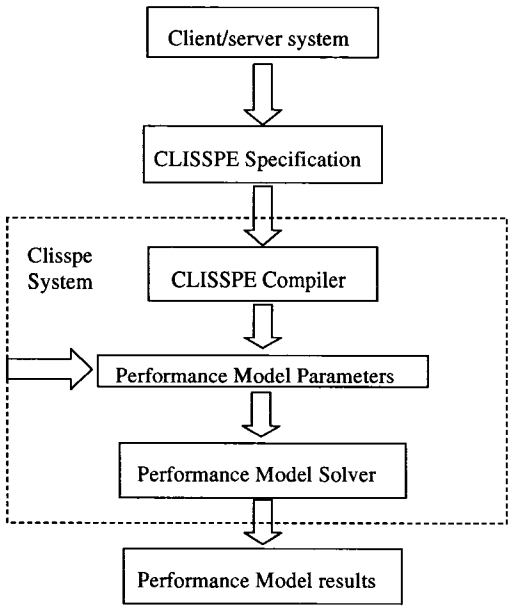


Figure 2.2: The transformation steps in CLISSPE system [68].

Balsamo et al.[16]:

The authors provided a method for the automatic derivation of a queuing network model from a software architecture specification using the "Chemical Abstract Machine" (CHAM) formalism. A software architecture specification is described in CHAM by a set of molecules that represent the static components of the architecture, a set of reaction rules that describe the dynamic evolution of the system through reaction steps, and an initial solution that describes the initial static configuration of the system. The paper presents an algorithm to derive a QN model from the CHAM specification. The algorithm is based on the analysis of the transition system (TS) associated to the CHAM specification. The TS represents the dynamic behavior of CHAM architecture and it can be represented by a directed labeled graph whose nodes are the system states, arcs are the transitions between states and labels are the transition rules that allow for state

transitions. The generation of the QN model depends on the interactions among the customers (users) and the components of the system. This kind of information is dynamic and it can only be derived from the analysis of the dynamic behavior of the system. The algorithm is organized into two sequential steps. The first step examines all the TS paths and looks at the states along a path in order to single out all the pairs of elements that are involved in an interaction. The second step uses these interaction pairs to devise the topology of the evaluation model and the customer classes. The algorithm does not completely define the QN model and its parameters, such as the service time distributions and the customer's arrival processes. The designer specifies these parameters. The solution of the QN model is derived by analytical methods or other means. The performance results provide insights into how to carry out the development process in order to satisfy given performance criteria.

2.4.2 Deriving Performance Models from UML Diagrams

The Unified Modeling Language (UML) has quickly become a standard notation for specification and design of software systems and it offers several diagrams describing different views of the design system. The advantage of transforming a UML design to a performance model resides in the fact that the latter can be derived from UML diagrams that are usually available early in the software lifecycle. This is important as it allows early evaluation of the software design.

The construction of a performance model requires structural and behavioral information about the designed system, as well as the allocation of software to hardware resources. The UML diagrams are used as the input information. Class and use case diagrams provide static aspects of a system while sequence and activity diagrams describe dynamic system behavior. The interactions among system components are represented using interaction diagrams, e.g., sequence and collaboration diagrams. The details about the physical implementation and the dependencies among software components are given via deployment and component diagrams.

There are different approaches and methodologies concerning the derivation of performance models from software architecture specification. These approaches have

different degrees of generality and architectural constraints or special assumptions on the software system, or they are based on the type of performance model and specification language. In different occasions, the transformation from UML model to performance model is carried out "by-hand", with graphical tools or transformation engines, etc. The following review provides a brief summary of various approaches in transforming a UML design to a performance model.

Cortellessa and Mirandola [30]:

The authors used the information from different UML diagrams of a system design to generate a performance model. The paper refers to the SPE methodology and describes the software architecture by using deployment, sequence, and use case diagrams. Their transformation includes the following steps:

1. Assigning a probability to every edge that links a type of user in the use case diagram. The probability applies to the execution of the corresponding use case, whose realization is given by sequence diagrams. This leads to the definition of the workload of the performance model.
2. Processing the corresponding sequence diagrams to obtain the meta-execution graph. The algorithm incrementally builds the meta-execution graph by processing the sequence diagrams.
3. Using the deployment diagram that shows the topology of the design platform to obtain the hardware platform for the EQN model and tailoring the meta-execution graph that defines the workload of the EQN.
4. Assigning numerical parameters to the execution graph instance.
5. Solving the obtained EQN performance model by using the SPE approach.

An important contribution of this methodology is adding information concerning performance evaluation to the UML diagrams to obtain the EQN model. However, there was no detail provided for the transformation and the methodology was not implemented in a tool.

Petriu and Wang [91,94,95]:

Their papers proposed a systematic approach to build LQN performance models from

UML description of high-level software architecture patterns of a system by using graph transformations. The high-level architecture patterns describe the main system components and their interactions at a level of abstraction that captures certain characteristics relevant to performance, such as concurrency, parallelism, contention for software resources, synchronization and serialization etc. The authors considered a significant set of architectural patterns (pipe and filters, client/server, broker, layers, critical section and master-slave). The transformation from the UML architectural description of a given system to its LQN model is based on PROGRES, a well-known visual language and environment for programming with graph rewriting systems. The approach follows the SPE methodology and generates the software and system execution models by applying graph transformation techniques. Software architectures are specified using UML use case, collaboration, and sequence and deployment diagrams.

To transform a UML design to a LQN model, the authors defined graph transformation rules for each architectural pattern. A PROGRES transaction was then executed for every architectural pattern found in the input architectural description graph. The translation process is completed when all the patterns have been processed. The final result is an LQN model written to a text file according to the predefined LQN model format.

Gomaa and Menascè [41]:

This transformation approach uses UML design models of component interconnection patterns and performance annotations of the UML design models using an XML-based notation. The performance-annotated UML design model is mapped to a performance model for performance evaluation of the software architecture with various system configurations.

In their paper, interconnection patterns for client/server systems were investigated. The patterns define and encapsulate the way in which client and server components of software architecture communicate with each other via connectors. UML class and collaboration diagrams are used to model the software system's static aspects and dynamic interactions between components and connectors objects. The model is then provided with additional performance annotations and translated into an XML notation

that contains both the architecture and performance parameters. The constructed performance models are EQN, and solutions are obtained by Markov chain analysis or approximate analytical methods.

Pooley and King [97]:

The authors of the paper described some preliminary ideas on how to derive a queuing network model from UML software design specifications. Use case diagrams are used to specify the workloads and classes of requests of the system. Component and deployment diagrams are used to define the available system resources. The transformation process is based on the correspondences between UML deployment and component diagrams, and queuing network models by mapping components and links to service centers. The mapping appears to be by hand as no specific transformation tool was mentioned. The authors also showed how to generate Stochastic Timed Petri Net models from the UML design specifications by considering the use case diagrams, collaboration and state diagrams [97]. The idea is to translate each state diagram that represents an object of the collaboration diagram into a Petri Net for performance evaluation. Another relevant paper by Pooley describes how to derive Stochastic Process Algebra models from UML design specifications consisting of a collaboration and state diagrams [98]. The concept is to produce a Stochastic Process Algebra description of objects in the collaboration diagram and to combine them into a unique model.

Hoeben [50]:

In his paper, the author recognized the lack of performance information in UML models and proposed rules to express or add useful information based on the use of stereotypes, tagged values and rules to propagate user requests, to derive performance evaluation from the UML diagrams. These rules allow performance evaluation of UML models at various levels of abstraction. The author provides a prototype tool to automatically create performance estimations based on QN models. Figure 2.3 shows the transformation concept proposed by the author. However, the author provided neither hints on how to obtain a QN model from the UML design specifications, nor a complete tool description.

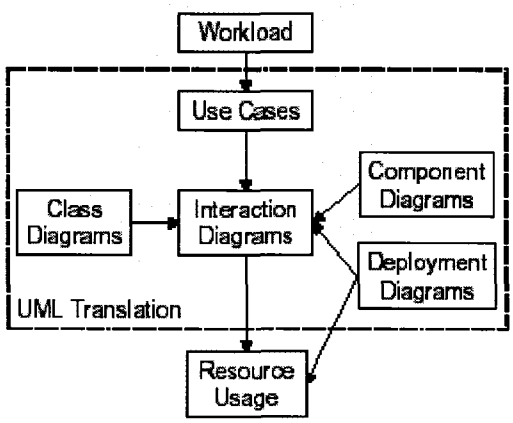


Figure 2.3: Overview of the usage of UML diagrams to obtain resource usage [50].

Arief and Speirs [12]:

The approach proposed in their paper presented a simulation framework named Simulation Modeling Language (SimML) to generate a simulation program from a system design specified with UML. The proposed UML tool allows the user to draw class and sequence diagrams and to specify the information needed for the automatic generation of the process oriented simulation model. The simulation program is generated in the Java programming language. Figure 2.4 shows the possible paths that can be taken to generate a simulation program from a design notation. The transformation involved storing UML model information in an XML formatted file for later use by the SimML framework. A SimML parser was required to read the information stored in an XML document before a simulation model can be generated.

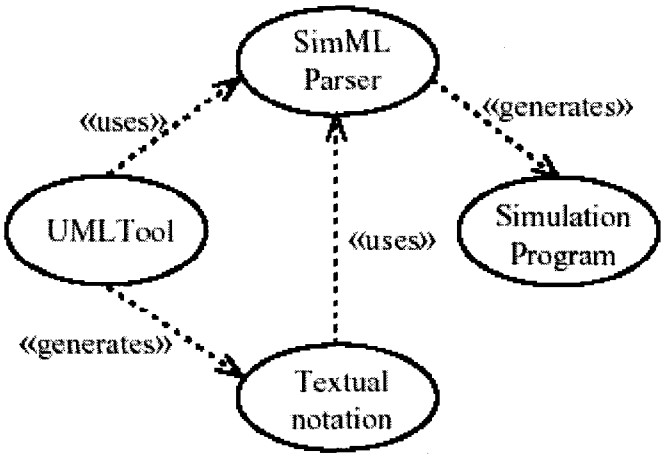


Figure 2.4 Possible paths to generate a simulation program [12].

Kähkipuro [55,56,57]:

The author constructed a performance-modeling framework that could be used in the development and maintenance of component-based distributed systems, such as those based on the CORBA, EJB, and COM+ platforms. The performance-modeling framework consists of three elements:

- A UML-based performance modeling notation,
- A set of modeling techniques,
- Tools for solving, presenting, and analyzing the models.

The framework defines mappings between the representations, as shown in Figure 2.5. The idea is to start from the UML representation and to proceed downward using the mappings. Once the bottom has been reached, an approximate solution can be found for the model. The mappings also indicate how the obtained metrics can be propagated upwards. The transformation steps include (1) normalizing UML diagrams into a textual format and removing elements that are not relevant for performance modeling by using the Performance Modeling Language (PML) textual notation; (2) expanding the PML based performance models into augmented queuing networks (AQNs); (3) using approximate techniques to solve the AQNs for performance evaluation.

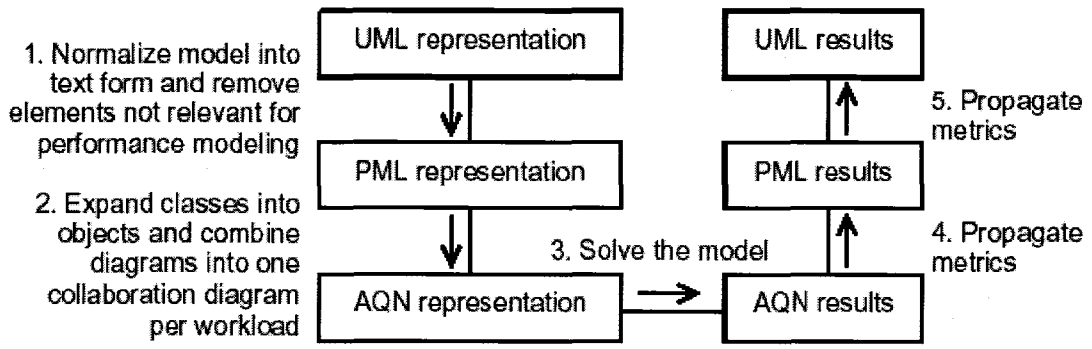


Figure 2.5: A schematic mapping of UML to PML and AQN representations [55].

The purpose of the performance-modeling notation is to provide a common language for representing performance related information in software systems. The author also proposed to use a subset of the UML notation with a few extensions for indicating

performance related information. The proposed extensions comply with the standard UML extension mechanisms. The proposed set of performance modeling techniques provides means for creating precise performance models for complex distributed systems using component-based distributed platforms.

Amer [2]:

This is an extension of Petriu and Wang's earlier work. This work proposes a graph grammar–based transformation from UML design models into Layered Queuing Network (LQN) performance models. The transformation is implemented with PROGRES, a graph-rewriting tool. The LQN model structure is generated from the high-level software architecture showing the architectural patterns used in the system, 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 either as UML interaction or activity diagrams annotated with performance information. If the scenarios are given as UML sequence diagrams, equivalent activity diagrams are produced and then converted into LQN. The proposed technique was applied to the performance analysis of three CORBA-based client server systems, and the performance model results are reasonably close to measurements obtained from the actual implementations.

Miguel et al. [74]:

The authors proposed extensions to UML diagrams to express requirements and resource usage of a real-time system. The extension is based on the use of stereotypes, tagged values and constraints. The software architecture is specified using the extended UML diagrams, which are used as input for the automatic generation of corresponding simulation models. The approach also provides a feedback mechanism to include simulated results in the tagged values. This constitutes a relevant feature, which facilitates getting feedback from the performance evaluation results to the software designers.

2.4.3 Deriving Performance Model from Use Case Maps

Scenario specifications provide a good starting point for system design and for analysis of various kinds of requirements. Use Case Maps (UCMs) are a graphical language specifically used for expressing scenarios and for experimenting with scenario interactions and architecture [5-8,25,73,75]. The UCM notation was invented by Buhr et al. [25] to capture designer intentions while reasoning about concurrency and partitioning of a system in the earliest design stages. A UCM is a collection of elements that describe one or more scenarios unfolding throughout a system and it can be used to develop the software architecture within a structural notation. It is designed to be intuitive and high level, although design details can be included.

The basic building block of the UCM notation is the path. The execution of a scenario in UCM is similar to a token traversing the path from start to end. UCM is a concurrent notation; there is no restriction as to the number of tokens that may traverse a given path or the position of any token on a path relative to any other token. UCM paths can also be overlaid on components, which represent functional or logical entities that are encountered during the execution of a scenario. Paths are refined to show more scenario detail through the addition of responsibilities. Responsibilities represent functions that need to be accomplished at given points in the execution of the scenario. UCM also incorporates forks and joins that show parallel and alternate scenario variants. More details about UCM can be found in [25]. Woodside and others have reported major work on the transformation of UCMs directly to LQN performance models [86-89,116,117].

Petriu and Woodside [85-89]:

The authors have a number of papers describing the development of a transformation from UCMs to LQN implemented in a tool called UCM2LQN. A flow chart of UCM to LQN transformation is given in Figure 2.6. The algorithm used to generate LQNs uses a point-to-point traversal of the UCM paths. A calling structure between the components based on the order in which they are traversed in the path is inferred. The proposed algorithm transforms UCM scenario models into LQN performance models and can be applied in principle to other source scenario models, including message sequence charts, UML activity, and sequence diagrams.

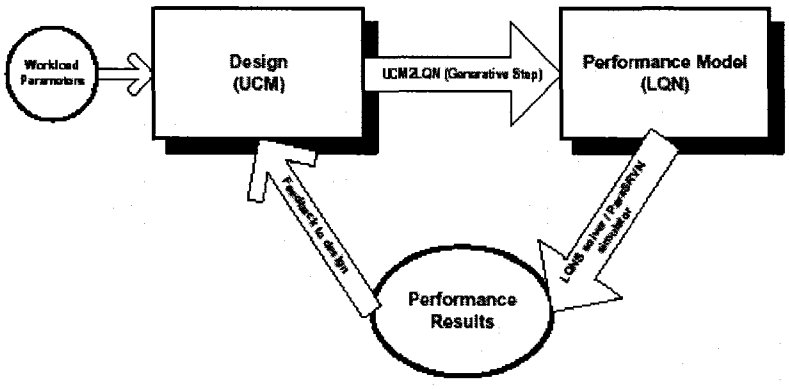


Figure 2.6: Flow chart of UCM to LQN transformation [86].

2.4.4 Deriving Performance Model from the UML Performance Profile

The UML Profile for Schedulability, Performance and Time [78] profile extends the UML metamodel with stereotypes and tagged values. It provides a notation for capturing performance requirements within the design context for specifying execution parameters, which can be used by modeling tools to compute predictions, and for displaying performance results computed by modeling tools or found in testing. Because SPT is an OMG standard, it stimulates recent research in the area of transformation from UML designs with performance annotations to performance models.

Petriu and Shen [90]:

The paper proposes a graph-grammar based method for transforming automatically a UML model annotated with SPT performance information into a LQN performance model. The input for the transformation algorithm is an XML file that contains the UML model in XML format according to the standard XMI interface. The transformation algorithm was completely implemented in Java on top of a rather large open source Java library (Argo UML) that implements and manipulates the UML 1.4 metamodel. The output is the corresponding LQN model description file, which can be read directly by 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 interaction or activity

diagrams. This paper has the following characteristics: a) it accepts an input XML files produced by UML tools, b) it generates LQN performance models by applying graph transformation techniques to graphs of meta-objects that represent different UML diagrams of the input model, and c) it uses the SPT performance profile for adding performance annotations to the input model.

Xu et al. [135]:

This paper demonstrated the use of the SPT profile to describe performance aspects of a UML design, and the use of the LQN performance model for giving feedback to the designers. Using a Building Security System (BSS) design as an example, the authors illustrated the performance annotations according to the SPT profile. The design was then transformed to a LQN performance model by a manual mapping. The focus of the paper is on addressing different kinds of performance concerns, and interpreting the performance results in order to suggest modifications to the design and to the planned run-time configuration. A more comprehensive introduction to the use of SPT for annotating UML software designs for performance analysis is given in [96].

2.4.5 Other Approaches

There are other approaches for transforming software designs to performance models that are not in the categories described above. For example, Andolfi et al. [9] presented an approach to automatically generate queuing network models from software architecture specifications described by means of Message Sequence Charts (MSC), that correspond to UML sequence diagrams. The idea is to analyze MSCs in terms of the trace languages (sequences of events) they generate, in order to single out the real degree of parallelism among components and their dynamic dependencies. This information is then used to build a QN model corresponding to the software architecture description.

Aquilani et at. [10] proposed the derivation of QN models from Labeled Transition Systems (LTS) describing the dynamic behavior of software architectures. The approach takes LTS model in the transformation process and no information concerning the system implementation or deployment is needed.

Bernardo et al. [20] introduced an architectural description language based on Stochastic Timed Process Algebras. This approach provides an integration of a formal specification language and performance models. The aim is to describe and analyze both functional and performance properties of software architectures in a formal framework. The approach proposes the adoption of an architectural description language called ÆMPA, and gives its syntax with a graphical and textual notation and its semantics in terms of Stochastic Timed Process Algebras. The authors illustrated various functional and non-functional properties including performance evaluation based on Markov chains.

It should be noted that the above review only covers a subset of the research papers published in recent years, which are considered more relevant to this proposed thesis research work. It is difficult if not impossible to do an exhaustive review of the literature here; however, more information can be found in other references [66,77,84,92,129, 125,126,131].

2.5 MODEL TRANSFORMATIONS

Any general transformation semantic needs to define the source model, target model and a set of textual or graphical rules to govern the transformations between the source and target models. As indicated in the paper by Blaha and Premerlani [21] that gives a concise overview of the model transformation theory in the context of object-oriented modeling, there are three types of transformations:

1. Equivalent Transformation: There is a unique one-to-one relationship between source instances and target objects in the models. Incidental information such as association or role names may be lost, but all of the modeled information should be retained.
2. Information-losing Transformation: The source model is more constrained than the target model. All instances of the source model can be mapped to the target model but not all target model objects can be mapped to valid instances in the source model. Information is lost as the result of the transformation.
3. Information-gaining Transformation: The source model is less constrained than the target model. A source model instance may not have a valid target model

object, but all target model objects can be generated from source model instances. Information is gained in the transformation, which must be supplied from another source.

Kleppe et al. gave another useful model transformation definition as follows [64]:
A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.

According to Mens et al. [70,71,72], a model transformation can be endogenous or exogenous, based on the language in which the source and target models of a transformation are expressed; and horizontal or vertical, upon whether the source and target models reside at different abstraction levels. Endogenous transformations take place between models expressed in the same language, while exogenous transformations between models expressed in different languages. In a horizontal transformation the source and target models reside at the same abstraction level, while in a vertical transformation the source and target models reside at different abstraction levels. According to the taxonomy of model transformations proposed in reference [70], the transformation proposed in this thesis is both exogenous (i.e., the source and target language of the models are different) and vertical (abstraction-raising, to be more precise). Another aspect discussed is that one can choose to execute the actual transformation in either the XML or the MDA technological space (which encompasses MOF and UML).

The transformation of a UML design annotated with performance information into a performance model is an information losing transformation, as there are many aspects of software design that will be abstracted away; it can also be considered as a vertical transformation. In terms of model transformation approaches, Czarnecki [34,35] pointed out that the domain analysis is concerned with analyzing and modeling the variabilities

and commonalities of systems. Common notation in domain analysis of model transformation includes transformation rules, rule application scope, source-target relationship, rule scheduling and organization etc. In his paper, Czarnecki [35] also summarized various model-to-model transformation approaches including direct-manipulation, relational, graph-transformation-based, structure-driven, and hybrid approaches. There are still many other approaches for specific applications [27,72,136]; however, relational, graph-transformation-based and abstraction raising approaches are of interest for this work.

2.5.1 GRAPH TRANSFORMATION CONCEPTS

The term of "graph transformations" is used to represent two kinds of graph transformation techniques: graph grammars and graph rewriting systems. The essential idea of all implemented graph grammars or graph rewriting systems is that they are a generalization of string grammars used in compilers [106]. The terms "graph grammars" and "graph rewriting systems" are often considered synonymous. However, strictly speaking, a graph grammar is a set of production rules that generates a language of terminal graphs and produces non-terminal graphs as intermediate results. On the other hand, a graph rewriting system is a set of rules that transforms one instance of a given class of graphs into another instance of the same class of graphs without distinguishing terminals and non-terminals.

A graph rewriting system is a tool that implements a graph grammar as a collection of rules to rewrite an input graph, including an algorithm for selecting the rules to be applied. The graph grammar is a specification of the family of graphs that can be constructed from a given graph instance. The algorithm guides the XSLT coding effort. An example of such a system is the algorithmic attributed graph grammar system named PROgrammed Graph Rewriting Systems (PROGRES) that comprises a language and its corresponding toolset [106,107].

The main elements of PROGRES are the following: typed nodes, labeled directed edges, a graph schema, graph rewriting rules, and algorithms that are used to select and apply the graph rewriting rules [106,107].

Node and Edge: The node and edge are the primitive elements of a graph. A node is considered to be a data structure that stores application information as attributes. A directed and labeled edge represents a binary relationship between the source and target nodes, with the edge label identifying the types of the relationship. Edges do not have attributes.

Graph Schema: A graph schema captures the application domain knowledge and describes how a host graph can be constructed because it identifies what node types and edge types can be connected, and the direction of the edge relationships between the nodes. It is very similar to a database schema defined in a binary entity relationship notation or object-oriented analysis class diagram.

Graph Rewriting Rules: A graph-rewriting rule is understood as an operation of host graph traversal and update. There are two main graph-rewriting mechanisms in PROGRES: production and transaction. A *production* is the most primitive graph rewriting operation. It modifies a fragment of the host graph, as illustrated in Figure 2.7. A *transaction* selects and applies one or more productions to rewrite a host graph, and it is atomic. A transaction is made up of imperative control statements and declarative rules, to specify, select, and apply a set of graph rewriting rules.

Figure 2.7 illustrates conceptually the application of the production rule from Figure 2.7(a) to the host graph in Figure 2.7(b). The production rule has a left-hand side (LHS) that shows a subgraph pattern to be replaced by its right-hand side (RHS). The production rule also specifies in detail how to compute the attributes of the RHS nodes, and how the RHS will be embedded in the host graph. In this example, node 1 from LHS will be kept in the RHS, whereas node 2 and its edges will be deleted and replaced by two new nodes, 3 and 4. A subgraph that matches the left-hand side is found in the host graph (if more such subgraphs exist, one will be chosen in a non deterministic fashion). The respective subgraph will be cut off from the host graph and replaced by the right hand side. Actually, the replacement procedure takes place as follows: the nodes to be kept are not eliminated together with their links with the rest of the graph. The nodes to be deleted are

cut off, and their links are eliminated as well. The new nodes are added and are linked by edges, as shown in the right hand side. The graph transformation concepts presented here are applied to XML tree transformation in the method proposed in the thesis, as described in Chapter 4.

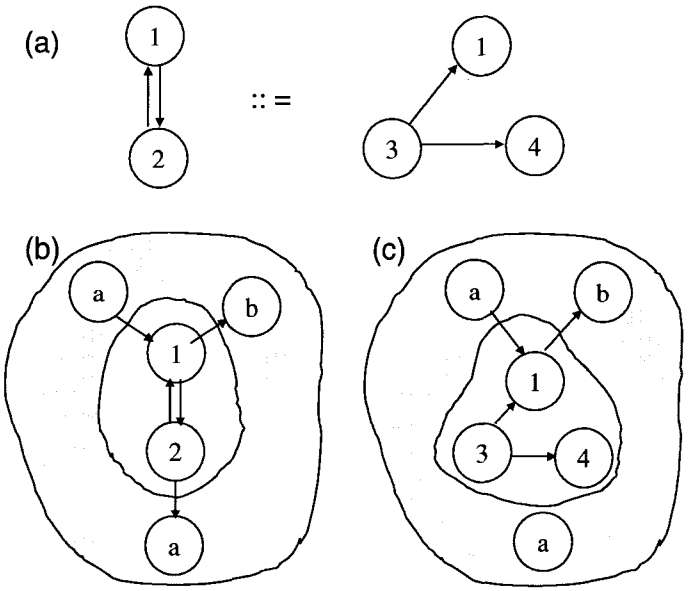


Figure 2.7: (a) Production rule, (b) host graph and (c) host graph after applied the rule.

2.5.2 OVERVIEW OF MOF QVT

The main requirement of OMG's QVT initiative is to provide a standard for expressing model transformations [80]. QVT requires that model transformations be defined precisely in terms of the relationships between a source metamodel and a target metamodel. A *transformation* is a generalization of both *relation* and *mapping*. Relations are bi-directional transformation specifications of the relationships between MOF models. Relations are not executable, in the sense that they are unable to create or alter a model; they can, however, be used for consistency checking between two models. Mappings are unidirectional transformation implementations used to generate a target model from the source model.

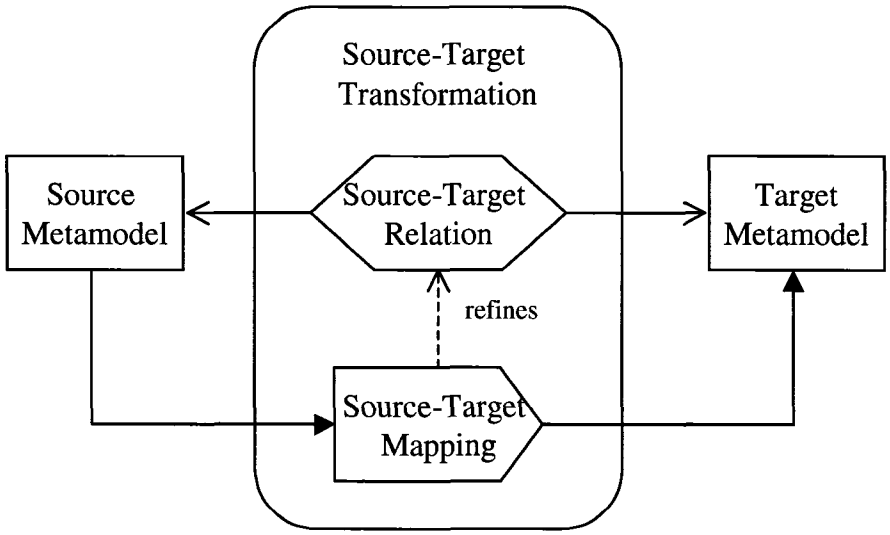


Figure 2.8: Transformation, relation and mapping in QVT.

QVT/Relations is a declarative pattern-matching language that allows the expression of complex constraints on the source metamodel, so that model fragments can be matched against metamodel patterns and used in a transformation [80]. A relation specifies a pattern expression on the source domain, a pattern expression on the target domain, and optionally a pair of when and where clauses. A pattern expression can be viewed as a graph of object nodes, their properties and association links originating from an instance of the domain's type. The *"when clause"* specifies the conditions under which the relationship needs to hold. The *"where clause"* specifies the conditions that must be satisfied by all model elements participating in the relation, and it may constrain any of the variables in the relation. A mapping can be a refinement of one or more relations. Therefore, the mapping must be consistent with the refined relations, which are used for checking the validity of the mapping. Figure 2.8 illustrates the refinement relationship between a relation and a mapping. Mapping operations may be used to implement one or more relations when it is difficult to provide a purely declarative specification. Mapping operations invoking other mapping operations always involve a relation for the purposes of creating a trace between model elements, and an entire transformation can be written in an imperative style. A transformation entirely written using mapping operations is called an operational transformation.

2.5.3 ABSTRACTION-RAISING TRANSFORMATION

A particular kind of transformation relevant to this work is the so-called "abstraction-raising" transformation, whereby a more abstract target model is generated from a more detailed source model [104]. This is the opposite of the transformations used in MDA for generating code from models, where a "refining" transformation produces a more detailed target model. Abstraction-raising transformation is particularly useful for generating analysis models that can be used for the verification of functional and non-functional characteristics of UML software models. Many modeling formalisms (such as queuing networks, Petri nets, fault trees, formal logic, process algebras, etc.) and corresponding tools have been developed over the years for the analysis of different non-functional characteristics (such as performance, reliability, scalability, security, etc.). The challenge is not to reinvent new analysis methods targeted to UML models, but to bridge the gap between UML-based software development tools and different existing analysis tools. In general, an analysis of model abstracts away many details of the original software, emphasizing only the aspects of interest.

In the abstraction-raising transformation approach proposed in [104], the source and target models are described by separate metamodels, between which transformations must be defined. The target metamodel represents analysis domain concepts, which are usually at a higher level of abstraction than the source model concepts. In order to define mappings between the source and target models, sometimes it is necessary to group a large number of source model elements according to certain rules, and to map the whole group to a single target element.

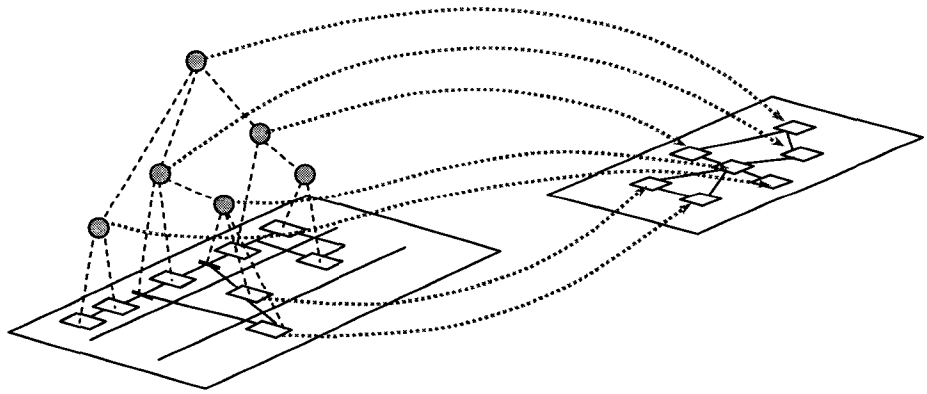


Figure 2.9: Illustration of the concept of abstraction raising transformation [104].

Some target model elements can be obtained by a direct mapping from source model elements, like in a relational transformation, whereas other target elements, representing more abstract concepts, correspond to graph-grammar non-terminals obtained by parsing the source model as indicated in Fig. 2.9. In such a case, an aggregation corresponds to the raising in the abstraction level necessary for bridging the semantic gap between the source and the target model. Rules are also dependent on the semantic differences between the source and target model, and are not represented in the source metamodel.

It is interesting to mention that the QVT/Relations language discussed in the previous section cannot easily define complex mappings as those required in an abstraction-raising transformation, where a group of source model elements must be aggregated together according to some given aggregation rules applied repeatedly, then mapped to a single target model element. The transformation approach proposed in this work contains abstraction-raising rules, as discussed in the following chapters.

CHAPTER 3: BACKGROUND TECHNIQUES AND TOOLS

3.1 INTRODUCTION

As mentioned, the significance of transforming UML software designs to performance models consists in the fact that it enables early performance evaluation and modifications in order to avoid performance inefficiencies. In Chapter 2, several approaches to transform UML designs to performance models have been reviewed. Although all these approaches have their uniqueness and advantages, they still suffer from all or some of the following insufficiencies.

- Non-standard performance annotations of the UML specifications complicate the processing of standard UML models. (This is especially true for early approaches that did not use the standard SPT profile).
- Transformation is specific to a certain tool or performance model. Some of the transformations rely on specific tools that require specific input languages and are heavily dependent on the performance modeling formalism.
- It is not interconnected with a UML tool. Many of the existing transformations of software designs do not accept direct input from commercially available UML design tools.
- It is a single performance model generation. Normally, previous transformations are performance model specific, i.e., the entire transformation is hard coded for a given performance model and is difficult to adapt to an alternative performance model.

The transformation method proposed in this work aims to address the above insufficiencies. More exactly, the goal is to accept a standard input model (including the performance annotations) from any UML tool. The transformation approach should depend as little as possible on the target performance modeling formalism, so that it can

be easily adapted to another performance model. As indicated in Figure 3.1, the proposed transformation consists of the following steps:

1. The starting point is a UML design model with performance annotations: This step involves using a UML tool to build a UML model and the SPT Performance Profile to annotate the UML diagrams with the necessary performance information. This step also involves an XMI specification compliant tool to convert the UML design to XML format.
2. Transformation of the UML model into a performance model: In this thesis, the transformation is conducted in two steps: first a transformation from UML (in XMI format) to an intermediary model, and then from the intermediary model to a performance model (either LQN, or simulation-based models).
3. Performance evaluation: The generated performance model can then be evaluated using an existing performance analyzer (LQN solver or CSIM18 simulation engine).
4. Result feedback and design optimization: The performance analysis may reveal the limitations and deficiencies of the original design, if any, which can be related back to the original design for further modifications.

It should be noted that this thesis work is focused on step 2 of the transformation process.

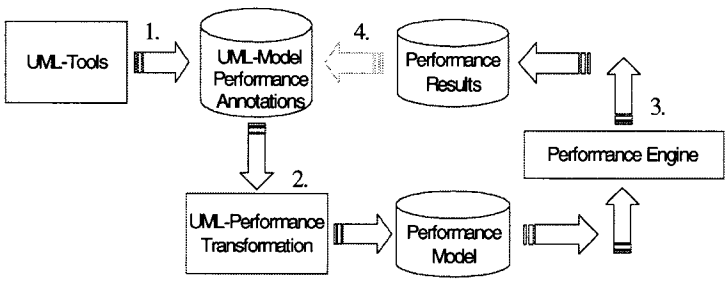


Figure 3.1: Steps of the performance evaluation process of UML models.

In this thesis work, XMLgebra is applied and extended to define a tree transformation technique based on graph transformation concepts. The implementation of the proposed method is done in the XML technological space; the tool used is SAXON XSLT processor [59].

3.2 BACKGROUND TECHNOLOGIES

In order to make this thesis self-contained, this chapter presents an overview of the background information related to the thesis research, such as the Unified Modeling Language (UML), SPT Profile, the eXtensible Markup Language (XML), the XML Metadata Interchange (XMI), Extensible Stylesheet Language for Transformations (XSLT), UML design tools, and performance models including Layered Queuing Networks (LQN) and simulation (CSIM18).

3.2.1 Unified Modeling Language

The Unified Modeling Language (UML) is a well-known graphical language for modeling software systems. It started as a combination of the Booch and Object Modeling Technique (OMT) notations, which was later combined with Jacobson's Object-Oriented Software Engineering (OOSE) [82]. Although UML is not necessarily tied to any particular application area or modeling process, its greatest applicability is in the area of object-oriented software design. Version 1.1 of the UML was submitted to the Object Management Group in September 1997 in response to an OMG RFP requesting a standard approach to object-oriented modeling. The proposal was accepted by OMG in November 1997. Version 1.3 of the UML was finalized in June 1999. The current version UML 2.0 is used throughout this thesis [81], which has various significant changes compared with its previous versions.

The UML consists of a notation and semantics. The notation is the collection of diagrams and the graphical and textual features used within those diagrams, and the semantics defines the meaning of the diagrams and features. The definitive descriptions of the UML latest version can be found in OMG website [81], however there are a number of books which describe the language more informally [23,52,53,102,103].

3.2.2 UML Profile for Schedulability, Performance and Time (SPT)

A UML profile is a domain-specific UML extension that uses the standard extension mechanisms (stereotypes, tagged values and constraints). The "UML Profile for Schedulability, Performance and Time" adopted by OMG [78] defines a general resource

model, time modeling, general concurrency, schedulability and performance modeling. Particularly, Chapter 8 defines the so-called Performance Profile dedicated to performance modeling, which provides mechanisms for capturing performance requirements and for associating performance related QoS characteristics with the UML model. The Performance Profile facilitates the following [78]:

- 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.

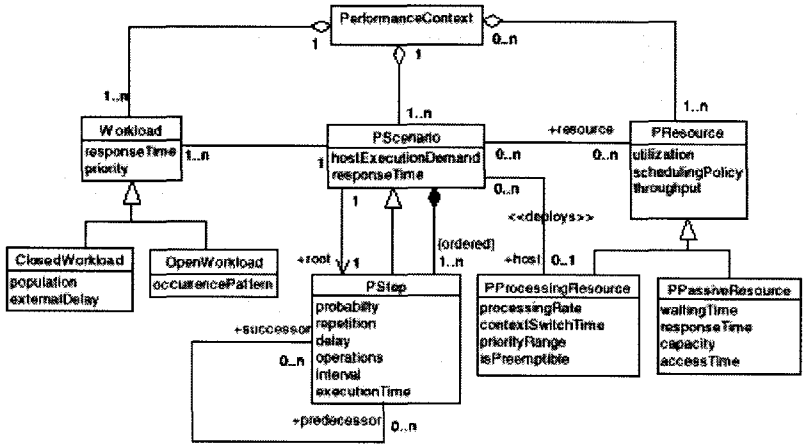


Figure 3.2: The performance analysis domain model [78].

The Performance Profile describes a domain model that contains the basic abstractions used in performance analysis, which includes scenarios, resources and workload (see Figure 3.2). Scenarios define response paths through the system, and can have QoS requirements such as response times or throughputs. Scenarios are executed by either closed or open workloads. Each scenario is composed of scenario steps that can be joined in sequence, loops, branches, forks and joins. 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 demands 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 profile maps each class to a stereotype that can be applied to a number of UML model elements, and each class attribute to a tagged value. For example, the basic abstraction PStep is mapped to the stereotype «PAstep». The main stereotypes used for performance modeling include «PAclosedLoad», «PAopenLoad», «PAhost», «PAstep» and «PAresource», which are defined as follows:

- «PAclosedLoad» stereotype represents a closed workload; it has four tags: PArespTime, PApriority, PApopulation, and PAextDelay.
- «PAopenLoad» models an open workload with the following tags: PArespTime, PApriority, and PAoccurrence.
- «PAhost» stereotype models a processing resource; its tags include: PAutilization, PAschdPolicy, PApreemptable, PAthroughput etc.
- «PAresource» stereotype models a resource with tags: PAutilization, PArespTime, PAthroughput etc.
- «PAstep» models a step in a scenario with tags: PAdemand, PArespTime, PAprob, PAdelay etc.

Through these stereotypes and tags, performance annotations can be attached to a UML design model. The importance of the Performance Profile rests in the fact that it provides a standard way of attaching quantitative performance attributes to the UML design model. These attributes may represent values that are required, assumed, measured, or computed from a model.

3.2.3 XML and XMI

In the proposed transformation, the XML (eXtensible Markup Language) is used to describe the source and target models. The representation of the source UML model is according to XMI (XML Metadata Interchange), an OMG standard specification that defines the MOF to XML model conversion.

eXtensible Markup Language (XML):

XML is a standard adopted by the World Wide Web Consortium (W3C) to complement HTML for data exchange on the Web. It provides an easy way to work with information in a tree-structured form. XML programming is a large subject and more information on XML can be found in the W3C web site [24] and books [46]. In this work we use XML as a data storage structure.

XML consists of two parts: documents and DTDs (Document Type Declarations). DTD serves as grammar for the underlying XML document [46]. An XML document is valid if it conforms to its DTD. In other words, elements in a valid document may be nested only in the way described by the DTD and may have only the attributes allowed by the DTD. XML document allows users to define new tags to indicate the structure of their documents. Hierarchical structures make XML documents faster to access and easier to manipulate. It is easy to apply the XML Stylesheet Language (XSL) to display the data, to produce output in postscript, TXT, PDF, or some other format defined by the user. For this reason, XML is fast becoming the data representation of choice for the Web, especially when used in combination with network-centric programs that send and retrieve information.

XML Metadata Interchange (XMI):

XMI is an interchange format for metadata that is defined in terms of the Meta Object Facility (MOF) standard [83]. As stated in the XMI specification, the main purpose of XMI is to enable easy interchange of metadata between UML modeling tools and metadata repositories in distributed heterogeneous environments. XMI integrates three key industry standards:

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

The UML standard defines a rich object-oriented modeling language that is supported by a range of graphical design tools. The MOF standard defines an extensible framework for

defining models for metadata, and providing tools with programmatic interfaces to store and access metadata in a repository. XMI allows metadata to be interchanged as streams or files with a standard format based on XML. The integration of these three standards into XMI combines their metadata and modeling technologies, allowing developers of distributed systems to share object models and other metadata over the Internet. The current official version of XMI is 2.1 [83] and the specification consists mainly of a set of XML Document Type Definition (DTD) production rules for transforming MOF based metamodels into XML DTDs, a set of XML Document production rules for encoding and decoding MOF based metadata, design principles for XMI based DTDs and XML Stream, and DTDs for UML and MOF.

The XMI schema and DTD architecture provide the necessary infrastructure for information transfer by defining an uniform treatment of object identity, internal and external references, document partitioning, tool-specific extensions and round-trip exchanges etc. [83]. 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. These fixed elements provide a default set of data types and document structure starting with the top-level XMI element. Each XMI document contains one or more elements called XMI that serves as a top-level container for the information to be transferred. Detailed descriptions of XMI DTD design and generation principles can be found in reference [24]. UML DTD is the most widely used XMI DTD. It is a physical mechanism for interchanging UML design models conforming to the UML metamodel, which is fed into an XMI DTD generator to produce the UML DTD used by tools to export and import UML models.

3.2.4 XMLgebra [28,29,61,62]

XMLgebra is a programming language abstraction for XML processing introduced in [61]. It can be considered as a kind of algebra over the structure of XML data and it provides a high-level mechanism for construction and deconstruction of XML data. It can serve as the basis for XML operations in a programming language [28,29,61,62]. In XMLgebra the processing of documents is done in terms of XML templates. An XML

template is a well-formed XML fragment, containing named gaps that may appear in place of elements and attributes. XML templates can be constructed using the special "plug" operation, which operates by inserting existing XML templates or string values into the gaps of other XML templates. XML templates can also be obtained by selecting an existing XML template using special "select" and by inserting gaps using "gapify" operations. The followings are descriptions of some basic definitions and operations of XMLgebra described by Kirkegaard [62]:

Template Domain Definition:

Let N be the set of natural numbers without zero, and let N be the set of strings over N. Let S be a finite alphabet of labels, let A be a finite set of attribute names, and let D be an infinite domain of values. Let G be a finite set of gap names, and assume that* $D \cap G = 0$. *A template domain* τ *over N is a subset of N*, such that:*

(1) if $v \cdot i \in \tau$, *where* $v \in N^*$ *and* $i \in N$, *then* $v \in \tau$;

(2) if $v \cdot i \in \tau$, *where* $v \in N^*$, $i \in N$ *and* $i > 1$, *then* $v \cdot (i-1) \in \tau$.

The empty string is denoted by ε *that represents the root of* τ *and elements in* τ *are called vertices. A vertex w is a child of a vertex v (and v the parent of w) if* $v \cdot i = w$, *for some i* $\in N$. *All template domains are assumed to be finite.*

Template Definition:

A template is defined as $t = (dom_t ; lab_t ; (\lambda_t^a)\ a \in A)$ *where* dom_t *is a finite template domain over N,* lab_t: $dom_t \rightarrow \Sigma$ *is a labeling function, and for every attribute name* $a \in A$, $\lambda_t^a : dom_t \rightarrow D \cup G$ *is a partial attribute value function. The set of all such templates is denoted* $T_{(\Sigma;A;G)}$.

According to Kirkegaard, any XML template can be expressed by: (1) Σ, a finite set of vertex labels; (2) A, a set of attribute names; and (3) G, a set of gap names. The structure of an XML template is represented as a template domain; it starts with the empty string ε that represents a logical root element. The template domain is a numbering scheme that traverses the XML elements in document order and assigns a string value from N* to

each element. For a given XML element, with vertex $v \in N^*$, the vertex for the i'th child element is numbered by assigning the string value $w = v \cdot i$.

The labeling function labels a vertex in a domain (dom_t) with its corresponding element label in Σ (the set of vertex labels). An attribute value of a vertex can be extracted by the functions λ_t^a, where "a" is an attribute name of a vertex in an XML template t. A *GAP* is a symbol of an empty element and its content can be filled in later. There are two different types of *GAPs*. One is a node *GAP* and it is an empty vertex. The other is an attribute GAP where the attribute value is unknown. The empty elements (gaps) of an XML template are represented using the symbol $GAP \in \Sigma$. The gap name (*GN*) is expressed by an attribute name $GN \in A$, together with an attribute function λ_t^{GN} mapping *GN* to the corresponding gap name at each *GAP*. Since there are finite template domains, it is possible to express any template $t \in T_{(\Sigma;A;G)}$ by using the mapping just described.

Figure 3.3 shows an example of a template fragment where `[Send]` and `[entryname]` are node and attribute GAPs, respectively. The label function (lab_t) maps the dom_t to Σ and the attribute function returns the value of the attribute.

```
<IM:Task name="T1" hardwareref="procA">
  <IM:Service name=[entryname]>
    <[Send]>
    <IM:Step name="f1">
      <IM:From name="b1"/>
      <IM:To name="b3"/>
    </IM:Step>
  </IM:Service>
</IM:Task>
```

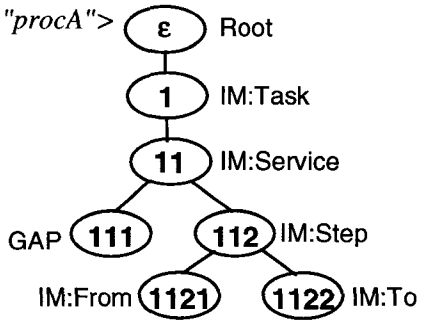


Figure 3.3: An example of a template.

The details of $T_{(\Sigma;A;G)}$ are as follows:

$dom_t = \{\varepsilon;1;11;111;112;1121;1122\}$,
$\Sigma = \{$ *Root; IM:Task; IM:Service; GAP; IM:Step; IM:From; IM:To* $\}$
$A = \{$*name; hardwareref; GN*$\}$
$G = \{$*entryname; Send*$\}$
$lab_t = [\ \varepsilon \rightarrow ROOT;\ 1 \rightarrow IM:Task;\ 11 \rightarrow IM:Service;\ 111 \rightarrow GAP;\ 112 \rightarrow IM:Step;$

1121 → IM:From; 1122 → IM:To]

$\lambda_t^{name}(1) = T1$; $\lambda_t^{hardwareref}(1) = procA$;

$\lambda_t^{name}(11) = entryname$; $\lambda_t^{GN}(111) = Send$; $\lambda_t^{name}(112) = f1$;

$\lambda_t^{name}(1121) = b1$; $\lambda_t^{name}(1122) = b3$.

Moreover, XMLgebra contains a number of functions. Some of the functions define operations that are useful in this work, such as "template plug", "string plug", "select" and "gapify" are briefly introduced below:

Template Plug Function Definition:

Let t_1; $t_2 \in T_{(\Sigma,A,G)}$ *be templates. The template plug function:* $\mathbf{tplug} : T_{(\Sigma,A,G)} \times G \times T_{(\Sigma,A,G)} \rightarrow T_{(\Sigma,A,G)}$ *is defined as* $\mathbf{tplug}(t_1, g, t_2) = (\mathbf{dom}_t ; \mathbf{lab}_t ; (\lambda_t^a)_{a \in A})$ *where:*

$$dom_t = \{v | v \in dom_{t1} \wedge v \notin gaps_{t1}(g)\} \cup \{v | v = u.w \wedge u \in gaps_{t1}(g) \wedge w \in dom_{t2}\}$$

$$lab_t(v) = \begin{cases} lab_{t1}(v) \ \textbf{if} \ (v \in dom_{t1} \wedge v \notin gaps_{t1}(g)) \\ lab_{t2}(w) \ \textbf{if} \ (v = u.w \wedge u \in gaps_{t1}(g) \wedge w \in dom_{t2}) \end{cases}$$

$$\forall a \in A: \lambda_t^a(v) = \begin{cases} \lambda_{t1}^a(v) \ \textbf{if} \ (v \in dom_{t1} \wedge v \notin gaps_{t1}(g)) \\ \lambda_{t2}^a(w) \ \textbf{if} \ (v = u.w \wedge u \in gaps_{t1}(g) \wedge w \in dom_{t2}) \\ null \quad \textbf{otherwise} \end{cases}$$

where $gaps_t(g)$ is an addressing function on templates in $T_{(\Sigma,A,G)}$. Given a gap name g, the addressing function finds matched GAP label vertices within the template and returns them. If the gap name is unique, the addressing function returns only one GAP labeled vertex. Let $t \in T_{(\Sigma,A,G)}$ be a template, the function *gaps* of t is defined as follows:

$$gaps_t(g) = \{v | v \in dom_t \wedge lab_t(v) = \text{GAP} \wedge \lambda_t^{GN}(v) = g\}$$

The $tplug(t_1,g,t_2)$ function returns a new template where t_2 is plugged into all GAP labeled vertices that matches the gap name g in t_1. We can plug a template to a given location by providing a unique g name. By using *tplug* function, we can construct a large template from number of small templates. The attribute functions (λ_t^a) $a \in A$ are defined to map attribute names of the originating value in t_1 or t_2 to the newly constructed template.

String Plug Function Definition:

splug : $T_{(\Sigma A,G)} \times G \times D \rightarrow T_{(\Sigma A,G)}$ *is defined as* $\mathbf{splug}(t_1,g,d) = (\mathbf{dom}_t ; \mathbf{lab}_t ; (\lambda_t^a)a \in A)$

where:

$$dom_t = dom_{t1}$$

$$lab_t(v) = \begin{cases} lab_{t1}(v) & \textbf{if } (v \in dom_{t1} \wedge v \notin gaps_{t1}(g)) \\ PCDATA & \textbf{if } v \in gaps_{t1}(g) \end{cases}$$

$$\forall a \in A,$$

$$\lambda_t^a(v) = \begin{cases} \lambda_{t1}^a(v) & \textbf{if } \lambda_{t1}^a(v) \notin G \\ d & \textbf{if } \lambda_{t1}^a(v) \in G \wedge a \neq GN \text{ // the case of attribute gap} \\ d & \textbf{if } v \in gaps_{t1}(g) \wedge a = GN \text{ // the case of node gap} \\ null & \textbf{otherwise} \end{cases}$$

The *splug*(t_1,*g*,*d*) function returns a new template where the string *d* is plugged into all matched gaps (both node and attribute gaps) with name *g* in t_1. The attribute function $(\lambda_t^a)a \in A$ maintains all original attribute value in t_1 except for node or attribute gaps. In either case, the gaps in t_1 are replaced by the value *d* in *t*.

Selection Function Definition:

Let $t_1 \in T_{(\Sigma A,G)}$ *be a template, and let* $p \in P$ *be a location path. Let the vertex set:* $E = evalpath_{t1}(p)(\varepsilon) = \{v_1, \ldots, v_n\}$ *contain the results of location evaluation, and assume that the elements* v_i *are numbered in template order according to* dom_{t1}. *The select function:* *select* : $T_{(\Sigma A,G)} \times P \rightarrow T_{(\Sigma A,G)}$ *is defined as* $\mathbf{select}(t_1, p) = (\mathbf{dom}_t ; \mathbf{lab}_t ; (\lambda_t^a)a \in A)$ *where:*

$$dom_t = \{\varepsilon\} \cup \cup_{vi \in E} \{i \cdot w \in N^* | u = v_i \cdot w \wedge u \in dom_{t1}\}$$

$$lab_t(v) = \begin{cases} Root & \textbf{if } v = \varepsilon \\ lab_{t1}(v_i \cdot w) & \textbf{if } v = i \cdot w;\ 1 \leq i \leq n \end{cases}$$

$$\forall a \in A,$$

$$\lambda_t^a(v) = \begin{cases} \lambda_{t1}^a(v_i \cdot w) & \textbf{if } v = i \cdot w;\ 1 \leq i \leq n \\ null & \textbf{otherwise} \end{cases}$$

Using the input argument XPath $p \in P$, the *select* function copies a sub section from the input template $t_1 \in T_{(\Sigma;A;G)}$. The new template has a root vertex ε and the labeling

function lab_t labels all vertices in the new template according to their corresponding labels in t_1. The attribute function $(\lambda_t^a)a \in A$) assigns attributes and values to all vertices in the new template according to their corresponding attributes and values in t_1.

Gapify Function Definition:

Let $t_1 \in T_{(\Sigma A,G)}$ be a template, and let $p \in P$ be a location path. Assume that the vertex set of the location path evaluation is: $E = evalpath_{t1}(p)(\varepsilon) = \{v_1,\ldots, v_n\}$. The function gapify: $T_{(\Sigma A,G)} \times P \times G \rightarrow T_{(\Sigma A,G)}$ is defined as **$gapify(t_1, p, g) = (dom_t ; lab_t ; (\lambda_t^a)a \in A)$** *where:*

$$dom_t = \{w \mid w \in dom_{t1} \wedge \forall v_i \in E,\ w \notin descendants_{t1}(v_i)\}$$

$$lab_t(v) = \begin{cases} lab_{t1}(v) & \textbf{if } v \notin E \\ GAP & \textbf{if } v \in E \end{cases}$$

$\forall a \in A,$

$$\lambda_t^a(v) = \begin{cases} \lambda_{t1}^a(v) & \textbf{if } v \notin E \\ g & \textbf{if } (v \in E \wedge a=GN) \\ null & \textbf{otherwise} \end{cases}$$

where the *descendant* function returns the set of the parameter's descendants. So, let $t \in T_{(\Sigma;A;G)}$ be a template, and define the function $descendants_t$ as:

$descendants_t(v) = \{w \mid w = v.s \wedge s \in N^*\}$

for each vertex in $v \in dom_t$, *it* gives the set of descendant vertices.

The *gapify* function replaces a sub-template indicated by the input argument p (XPath) with a GAP named g. The labeling function lab_t maps vertices to its corresponding label in t_1, except for vertices that have been replaced by a *GAP*. The attribute function $(\lambda_t^a)a \in A$ preserves attributes $a \in A$ of vertices $v \in dom_t$ to their values in t_1 except for the vertices that have been replaced.

Example:

This example illustrates the plug function which constructs a new template (T3) from templates T1 and T2 by plugging T2 in the GAP name g = [IM:Activity] in T1, according to the operation $tplug(t_1, g, t_2)$. The starting and resulting templates are shown in Figure 3.4.

Template T1:

```
<IM:Task name="application" hardwareref="ApplProc">
<IM:Service name="e2">
<[IM:Activity]/>
<IM:Join name="s2(application)">
<IM:From name="send_reqest"/>
<IM:From name="App_idle"/>
<IM:To name="process"/>
</IM:Join>
</IM:Service >
</IM:Task>
```

$T1_{(\Sigma;A;G)}$:

Σ={ Root; IM:Task; IM:Service; GAP; IM:Join; IM:From; IM:To }
A={name; hardwareref;GN}
G = { IM:Activity }
dom_{t1} = {ε; 1; 11; 111; 112; 1121; 1122; 1123},
lab_{t1} = [$\varepsilon \rightarrow$ ROOT; 1 $\rightarrow$ IM:Task; 11 $\rightarrow$ IM:Service;111 $\rightarrow$ GAP; 112 $\rightarrow$ IM:Join;1121 $\rightarrow$ IM:From; 1122 $\rightarrow$ IM:From;1123 $\rightarrow$ IM:To]
$\lambda_{t1}^{name}(1)$ = application; $\lambda_{t1}^{hardwareref}(1)$ =ApplProc; $\lambda_{t1}^{name}(11)$ = e2; $\lambda_{t1}^{GN}(111)$ = IM:Activity; $\lambda_{t1}^{name}(112)$ = s2(application); $\lambda_{t1}^{name}(1121)$ = send_reqest; $\lambda_{t1}^{name}(1122)$ = App_idle; $\lambda_{t1}^{name}(1123)$ =process.

Template T2:

```
<IM:Activity name="process" servicetime="2.0">
<IM:From name="s2(application)"/>
<IM:To name="s3(application)"/>
</IM:Activity>
```

$T2_{(\Sigma;A;G)}$:

Σ = { Root; IM:Activity ;IM:From; IM:To }
A = {name; servicetime}
G = {null}
dom_{t2} = {ε;1;11;12},
lab_t = [$\varepsilon \rightarrow$ ROOT; 1 $\rightarrow$ IM:Activity; 11 $\rightarrow$ IM:From;12 $\rightarrow$ IM:To]
$\lambda_{t2}^{name}(1)$ = process; $\lambda_{t2}^{servicetime}(1)$ =2.0; $\lambda_{t2}^{name}(11)$ = s2(application); $\lambda_{t2}^{name}(12)$ = s3(application).

Template T3:

```
<IM:Task name="application" hardwareref="ApplProc">
<IM:Service name="e2">
<IM:Activity name="process" servicetime="2.0">
<IM:From name="s2(application)"/>
<IM:To name="s3(application)"/>
</IM:Activity>
<IM:Join name="s2(application)">
<IM:From name="send_reqest"/>
<IM:From name="App_idle"/>
<IM:To name="process"/>
```

```
</IM:Join>
</IM:Service>
</IM:Task>
```

$T3_{(\Sigma;A;G)}$:

Σ={ Root; IM:Task; IM:Service; IM:Activity; IM:Join; IM:From; IM:To }
A={name; hardwareref; servicetime}
G = {null}
dom_{t3} = {ε; 1; 11; 111; 112; 1111; 1112; 1121; 1122, 1123},
lab_{t3} = [ε → ROOT; 1 → IM:Task; 11 → IM:Service;111 → IM:Activity; 112 → IM:Join; 1111 → IM:From;1112 → IM:To; 1121 → IM:From; 1122 → IM:From;1123 → IM:To]
$\lambda_{t3}^{name}(1)$ = application; $\lambda_{t3}^{hardwareref}(1)$ =ApplProc; $\lambda_{t3}^{name}(11)$ = e2; $\lambda_{t3}^{name}(111)$ = process; $\lambda_{t3}^{servicetime}(111)$ =2.0; $\lambda_{t3}^{name}(112)$ = s2(application); $\lambda_{t3}^{name}(1111)$ = s2(application); $\lambda_{t3}^{name}(1112)$ = s3(application); $\lambda_{t3}^{name}(1121)$ = send_reqest; $\lambda_{t3}^{name}(1122)$ = App_idle; $\lambda_{t3}^{name}(1123)$ =process.

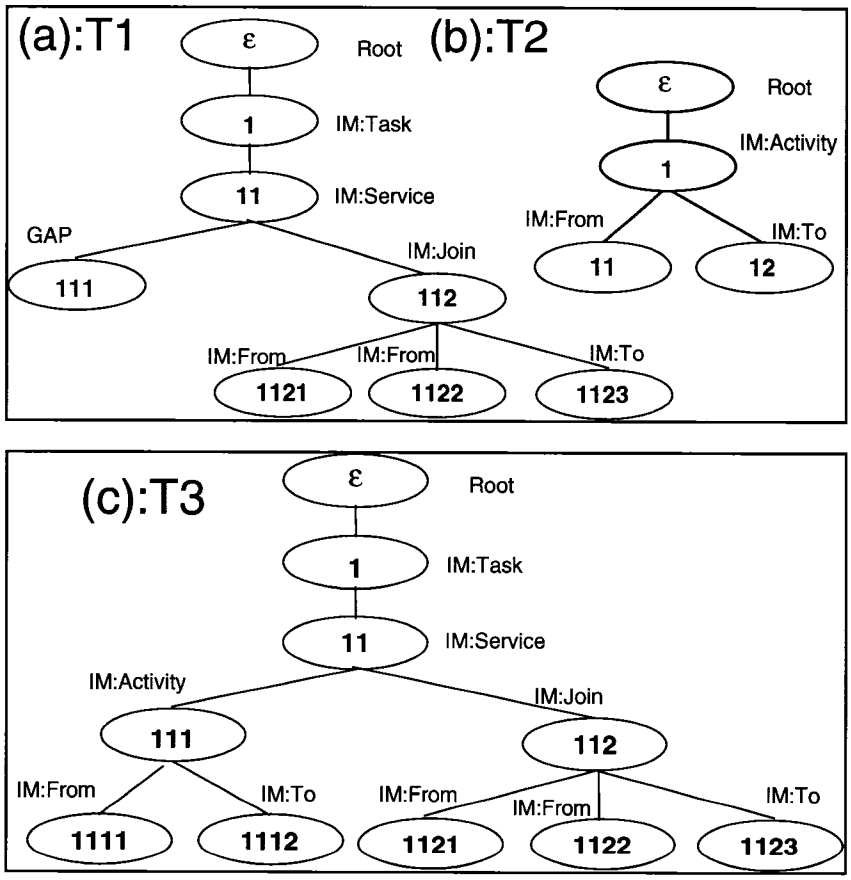


Figure 3.4: (a) Template T1; (b) Template T2 and (c) resulting template T3.

3.2.5 Extensible Stylesheet Language for Transformations (XSLT)

The World Wide Web Consortium (W3C) recommends the use of XSLT for transforming XML documents into various formats including HTML, XML, text, PDF and JAVA or C/C++ code [128]. An XSLT *stylesheet* contains a set of template rules. A template rule contains a *pattern* that is matched against nodes in the source tree and a *template* that can be instantiated to form part of the result tree. This rule allows a *stylesheet* to be applicable to a wide class of documents that have similar source tree structures.

In the transformation process, a template can contain elements that specify instructions for creating result tree fragments. When a template is instantiated, each instruction is executed and replaced by the result tree fragment that it creates. Instructions can select and process descendant source elements. Processing a descendant element creates a result tree fragment by finding the applicable template rule and instantiating its template. Elements are only processed when they have been selected by the execution of an instruction. The result tree is constructed by finding the template rule for the root node and instantiating its template.

In the process of finding the applicable template rule, more than one template rule may have a pattern that matches a given element. However, only one template rule will be applied. A single template by itself has considerable power: it can create structures of arbitrary complexity; it can pull string values out of arbitrary locations in the source tree; it can generate structures that are repeated according to the occurrence of elements in the source tree. For simple transformations where the structure of the result tree is independent of the structure of the source tree, a *stylesheet* can often consist of only a single template, which functions as a template for the complete result tree.

In this work, the free software SAXON is used as the XSLT engine to perform transformation [59]. The SAXON package is a collection of tools for processing XML documents and its main components include an XSLT processor, which implements the Versions 1.0 and 1.1 XSLT, and a Java library, which supports a similar processing model to XSL allowing full programming capability needed to perform complex processing of the data or to access external services. Also, SAXON provides a set of

services that are particularly useful when converting XML data into other formats. The output format may be XML, or HTML, or some other formats.

3.2.6 UML Design Tools

UML-based CASE (Computer Aided Software Engineering) tools support the use of design diagrams for object-oriented software development. XML is normally used as a universal data exchange format since many software vendors are building tools for importing and exporting XML data. Some of the most commonly used UML design tools are Poseidon [99], IBM Rational Software Architect [51] and Together Solo [123]. These tools support the UML DTD and XMI.

Poseidon and ArgoUML:

Poseidon for UML is a professional UML CASE tool. It is based on the open source project ArgoUML, a pure Java open source tool that provides support for object-oriented design [11,99]. The Community Edition of Poseidon (free version) provides editing and code generation features that enhance usability and support the needs of designers. The major features related to UML and XMI are described briefly below [99]:

- Implemented in Java, platform independent;
- UML diagrams are supported;
- Compliant to the UML 1.4 and 2.0 standards;
- XMI 2.1 is supported as a standard saving format.
- Graphic formats jpeg and png supported.

A recent Poseidon for UML is version 5.0, which supports UML 2.0 Component diagrams and has stability improvements.

Rational Software Architect (RSA) [51]:

IBM Rational Software Architect (RSA) is a construction tool that automates much of the effort needed to support a model-driven approach to the design and development of service-oriented architectures. It supports object-oriented modeling and design using the open standard Unified Modeling Language (UML 2.0) to document applications. RSA

has a function that allows a direct conversion of a UML 2.0 graphical design to XML format in accordance with the XMI specification.

Together Solo[123]:

The Borland Together Solo integrates UML design and development capabilities into an environment for modeling small-scale application projects with speed and efficiency. Together Solo includes a UML diagram editor, programming editor, pattern support, and simultaneous round-trip engineering, all designed to accelerate the software development.

Other tools, including Telelogic Tau [121], Enterprise Architect (EA) [37] and Visual Paradigm for the Unified Modeling Language (VP-UML) [127], are claimed supporting UML and XMI. However, RSA is the design tool used in this research work and other tools are not used.

3.2.7 Architectural Patterns

The high-level architectural patterns used in a system design provide key information for constructing the performance model. A pattern definition describes both the structure and the behavior; the former represents the components and the latter shows how they interact in a design system. There are a number of architectural patterns that have been identified in literature such as client-server, pipeline and filters, blackboard, broker, etc. A pattern introduces a high-level of abstraction design artifact by describing a specific type of collaboration between a set of prototypical components playing well-defined roles, and it helps our understanding of complex systems. 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. A pattern is usually represented as a UML collaboration drawn as an ellipse with dashed lines that may have an "embedded" square showing the roles played by different pattern participants. For instance, Figure 3.5 shows the structure and behavior of the client-server pattern. There are two variants: with blocking and non-blocking calls, respectively. In the former, the client sends a request and remains blocked until the server replies (Figure 3.5b), and in the latter the client continues its work after sending the request, and accepts the server's reply later (see Figure 3.5c).

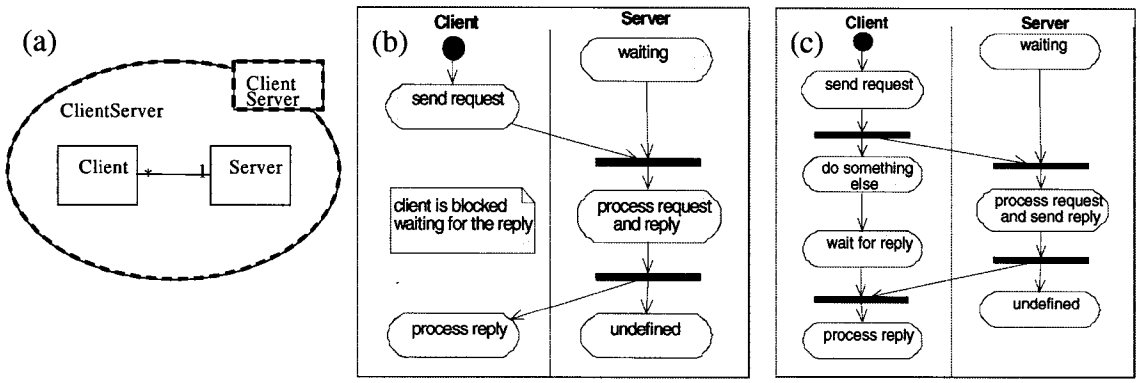


Figure 3.5: (a) Client-Server Pattern diagram; (b) Client-Server Pattern with blocking call; (c) Client-Server Pattern with non-blocking call.

Architectural patterns are useful in software architecture to performance model mapping. Therefore, any pattern design information in the UML model should be retained in the transformation so that the information could be used at a later stage.

3.2.8 Performance Modeling Techniques

There are numbers of performance models, including QN, LQN, Petri Nets and simulation, used to evaluate the performance and to improve the system by identifying bottlenecks as discussed in Chapter 2. Two performance models are chosen here as target models: one is analytic model– namely LQN, and the other is a simulation model – namely CSIM. The following sections provide brief descriptions of the LQN and CSIM models.

3.2.8.1 Layered Queuing Network (LQN)

Queuing network (QN) modeling is a widely used technique for analyzing the performance of computing systems and it has been successfully used in the context of traditional time-sharing computers [63]. Regardless of its success, QN often fails to capture complex interactions among various software and hardware components in client server distributed processing systems. The Layered Queuing Network (LQN) [38,132,133] is a modeling technique that was developed for handling such complex interactions. LQN is an extension of queuing models for systems with software and

hardware servers and resources. It has been applied to a number of industrial systems and was proven useful for providing insights into performance limitations at both software and hardware levels. A model in LQN is closely linked to the software specifications, which makes it easy to develop and understand. It is suitable for systems with parallel processes running on a multiprocessor system or on a network, such as client-server systems.

A LQN model is represented as an acyclic graph whose nodes are software entities and hardware devices, and whose arcs denote service requests. Tasks represent hardware or software objects that may execute concurrently. A task can be a client, a server or an active server, which sends, receives requests, or does both. An active server, to which requests are arriving and queuing for service, may become a client to other servers. This gives rise to nested services and it marks the main difference between LQN and QN. It is important to note that the word "layered" in the name of LQN does not imply a strict layering of the tasks. A task may call other tasks in the same layer, or skip over layers [39]. The LQN toolset includes both simulation and analytical solvers and it can be used in solving the generated LQN performance models [93].

The main LQN components include processors, tasks, activities, entries and arcs. Brief descriptions are given below. However, for more details of LQN, readers may refer to publications by Franks et al. [38], Woodside et al. [132,133] and others [76,101].

Tasks and Processors:

Tasks and Processors can be considered as server nodes. They are classified into three categories; namely clients, servers, and active servers. Clients that only send requests are used to model actual users and other sources of input. Servers that only receive requests are used to model hardware devices such as processors or disk devices. Active servers that can receive or send requests are used to model typical operating system processes. Although not explicitly illustrated in LQN notation, each server has an implicit message queue, called the request queue, where the incoming requests are waiting in the queue to be served. The default scheduling policy of the request queue is FCFS, but other policies are also supported.

A software or hardware server node can be either a single-server or a multi-server. A multi-server is composed of more than one identical clone that work in parallel and share the same request queue. A multi-server can also be an infinite-server if there is no limit to the number of its clones.

Entries:

An Entry acts as a port or an address for a particular service offered by a task. A service has its own execution time and demands for other services given as model parameters. Servers with more than one entry still have a single input queue, where requests for different entries wait together. An entry can be further decomposed into phases or activities, if more details are required to describe its execution scenario. This is typically required when entries have fork and join interactions.

Activities:

Activities are components that represent the lowest level of detail in the LQN performance model. They can be connected together not only sequentially, but with fork and join interactions as well. Activities are connected together to form a directed graph, which represents one or more execution scenarios. Execution may branch into parallel concurrent threads of control, or choose randomly between different paths. An Activity may have service time demand on the processor on which it runs. Similar to entries, activities can make requests to other tasks by way of synchronous or asynchronous messages.

An "AND-fork" allows all successor activities to execute in parallel; whereas an "OR-fork" allows only one of its successor activities to be executed with a given probability. A sequential execution is a special case of an "OR-fork" with only one branch. An "AND-join" introduces synchronization delay [93]. Forking happens when a thread of control splits into two or more concurrent sub-threads, while joining happens when two or more tasks synchronize with one another. There are two forms of fork-join behavior that are based on whether the fork and join take place within the same task (known as intra-task fork-join), or in two separate tasks (known as inter-task fork-join). This pattern is

particularly useful for improving performance if parallelism is involved in an application [93].

Arcs (Requests):

Arcs in an LQN model denote requests of services. The labels on the arcs denote the average number of requests made each time the corresponding phase (activity) is executed. Requests for service from one server to another can be made via three different kinds of messages in LQN models: synchronous, asynchronous and forwarding [39]. 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. After accepting a request for one of its entries, the server starts to process it by executing a sequence of one or more phases of that service. At the end of phase 1, the server replies to the client, the latter 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. During any phase, the server may act as a client to other servers. In the case of asynchronous message, the client does not block after sending the message and the server does not reply back, executing its phases instead (Figure 3.6(a) and (b)).

Figure 3.6(c) demonstrates a forwarding message (represented by a dotted request arc) with a synchronous request that is served by a chain of servers. The client sends a synchronous request to Server1, which begins to process the request, then at the end of phase1 forwards it to Server2. Sever1 proceeds normally with the remaining phases in parallel with Server2 and starts another cycle at the end of its last phase. The client, however, remains blocked until Server2, which replies to the client at the end of its phase 1, serves the forwarded request. A forwarding chain can contain any number of servers, in which case the client waits until it receives a reply from the last server in the chain. The total CPU demand of a phase is divided up into exponentially distributed slices; each of which is delimited by a request to lower level servers. The mean execution time is the same for all the slices. Requests to lower level servers are geometrically distributed with

a specified mean in stochastic phases, and occur for a fixed number of times in a deterministic phase.

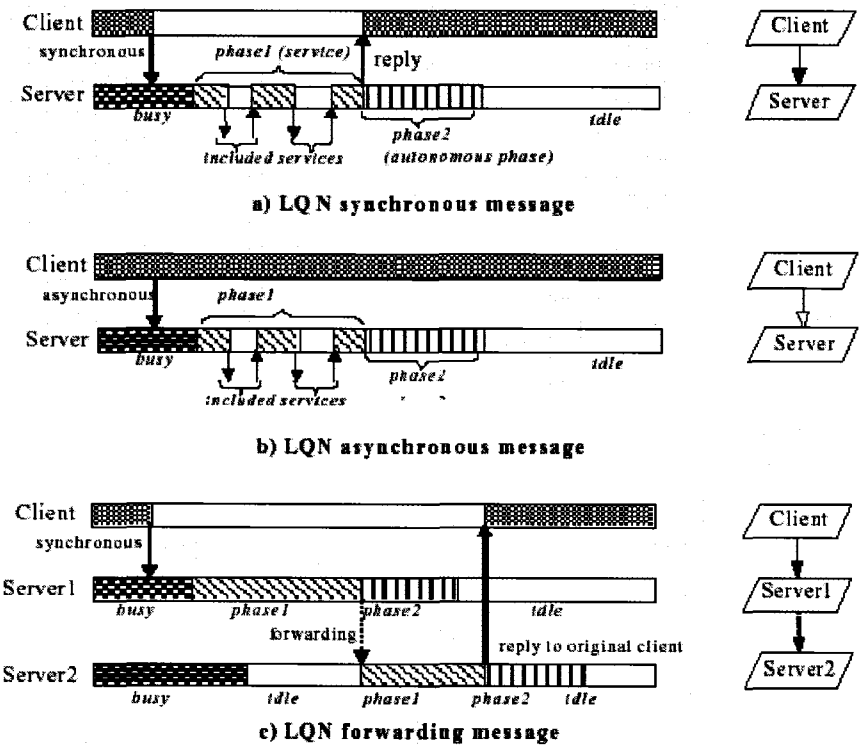


Figure 3.6: Execution of synchronous, asynchronous, and forwarding LQN requests [38].

LQN models can be solved using the analytical solver provided in the toolset [38]. Typical results of a LQN model are response times, throughput, utilization of servers on behalf of different types of requests, and queuing 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 [76].

3.2.8.2 CSIM18

The CSIM18 is a process-oriented discrete event simulation package for use with C or C++ programs [33]. It is implemented as a library of classes and procedures that implement all necessary structures and operations. It is a convenient tool that programmers can use to create simulation programs. A CSIM program models a system as a collection of CSIM processes that interact with each other by using the CSIM structures. The purpose of modeling a system is to produce estimates of time and

performance. The model maintains simulated time to yield insight of the dynamic behavior of the modeled system. The CSIM provides simulation objects including processes, facilities, storages, buffers, events and mailboxes etc. It also provides means for managing queues, statistics, collecting data and other services. In here, the common elements used in our simulation are introduced. More information of CSIM is provided elsewhere [108-114].

Processes:

Processes represent the active entities in a CSIM model. There can be several simultaneously "active" instances of the same process. Each of these instances appears to be executing in parallel in simulated time even though they are in fact executing sequentially on a single processor. The CSIM runtime package allows that each instance of every process has its own runtime environment. All processes have access to the global variables of a program. When a procedure executes a create statement, the process executing the create statement (the called process) is established and is made "ready to execute" at the statement following the create statement, and the calling process continues its execution at the statement after the procedure call to the called process. The calling process continues as the active process until it suspends itself.

Processes appear to operate simultaneously with other active processes at the same points in simulated time. The CSIM process manager creates this illusion by starting and suspending processes as time advances and as events occur. Processes execute until they "suspend" themselves by doing one of the following actions: execute a hold statement, execute a statement that causes the processes to be placed in a queue, or terminate. Processes are restarted when the time specified in a hold statement elapses or when a delay in a queue ends. The simulated time passes only by the execution of hold statements. While a process is actively computing, no simulated time passes.

Facilities:

A facility is normally used to model a resource in a simulated system. A CPU and a disk might both be modeled by CSIM facilities. A simple facility consists of a single server and a single queue for processes waiting to gain access to the server. Only one process at

a time can be using a server. A multi-server facility contains a single queue and multiple servers. All of the waiting processes are placed in the queue until one of the servers becomes available. A facility set is an array of simple facilities; in essence, a facility set consists of multiple single server facilities, each with its own queue. Normally, processes are ordered in a facility queue by their priorities. In cases of ties in priorities, the order is FCFS. An FCFS facility can be designated as a synchronous facility.

Storages:

CSIM storage is a resource that can be partially allocated to a requesting process. A storage consists of a counter to indicate the amount of available storage and a queue for processes waiting to receive their requested allocation. A storage set is an array of these basic storages. A storage can be designated to be synchronous. In a synchronous storage, each allocate is delayed until the onset of the next clock cycle.

Buffers:

A CSIM buffer is a resource that stores (holds) a number of tokens. The primary operations for a buffer are put, which places a number of tokens into the buffer, and get, which removes a number of tokens from the buffer. A buffer has a maximum capacity for holding tokens. A get operation stalls if there are too few tokens in the buffer, and a put operation stalls if there is not enough space in the buffer. A buffer consists of a counter indicating the number of tokens in the buffer, and two queues: a put-queue, for processes waiting to complete a put operation and a get-queue, for processes waiting to complete a get operation.

Events:

Events are used to synchronize the operations of CSIM processes. An event exists in one of two states: occurred or not occurred. A process can change the state of an event or it can suspend its execution until an event has occurred. When a process is suspended it can join a set of processes, all of which will be resumed when the event occurs. Or, it can join an ordered queue from which only one process is resumed for each occurrence of the event. An event is automatically reset to the not occurred state when all of the suspended processes that can proceed have done so. Advanced features of events include the ability

to create sets of events for which processes can wait and the ability for a process to bind its waiting time by specifying a time-out event. Events can also be used to construct other synchronization mechanisms such as semaphores.

Mailboxes

A mailbox allows synchronous exchange of data between CSIM processes. Any process may send a message to any mailbox, and any process may attempt to receive a message from any mailbox. A mailbox is comprised of two FCFS queues: a queue of un-received messages and a queue of waiting processes. At least one of the queues will be empty at any time. When a process sends a message, the message is given to a waiting process or it is placed in the message queue. When a process attempts to receive a message, it is either given a message from the message queue or it is added to the queue of waiting processes.

Statistics Gathering:

CSIM automatically gathers and reports performance statistics for certain types of model components, including facilities, storages and buffers. CSIM also provides general-purpose statistics gathering tools that can be used to obtain statistics for facilities, storages, mailboxes, events and other model components. Most of the reports produced by the statistics gathering tools accommodate floating-point numbers to given reasonable precision.

Tables and Qtables:

A table is used to gather statistics on a sequence of discrete values such as inter-arrival times, service times, or response times. Data values are recorded in a table to include them in the statistics. A table does not actually store the recorded values; it simply updates the statistics each time a value is included. The statistics maintained by a table include the minimum, maximum, range, mean, variance, standard deviation, and coefficient of variation.

A qtable is used to gather statistics on an integer-valued function of time, such as the length of a queue, the population of a subsystem, or the number of available resources. Similar to a table, a qtable does not actually save the functional values; it simply updates

the statistics each time the value changes. A qtable is also used to gather statistics on a real-valued (double) function of time. Every change in the value of the function is noted by calling a CSIM function. The statistics maintained by a qtable also include the minimum, maximum, range, mean, variance, standard deviation, and coefficient of variation. The number of changes in the functional value is maintained, as well as the initial and final values. Both table and qtable have the optional features that allow the creation of a histogram, the calculation of confidence intervals, and the computation of statistics for values in a moving window.

3.3. SUMMARY

This chapter describes basic concepts, techniques and tools for the proposed transformation of UML software design to a performance model for early performance evaluation. It provides brief descriptions for the background information related to this thesis work, such as the Unified Modeling Language (UML), UML Performance Profile, XMLgebra, the eXtensible Markup Language (XML), the XML Metadata Interchange (XMI), Extensible Stylesheet Language for Transformations (XSLT), UML design tools, architecture patterns and performance models of LQN and simulation (CSIM18). In the next chapter, the proposed XML-based transformation from a UML model to a LQN and CSIM-based simulation model will be discussed.

CHAPTER 4: FROM UML DESIGNS TO PERFORMANCE MODELS

4.1 INTRODUCTION

An overview of the model and metamodel transformation relationships is shown in Figure 4.1. The top level is OMG's Meta-Object Facility MOF (Meta-meta-model), which is used to define the metamodels on the next level associated with different domains. In the left of second level, it is the standard UML metamodel plus the SPT profile (the latter is shown in Figure 3.2). In the middle, it is the intermediate model (IM) metamodel, which is closely defined in accordance with the SPT domain model. The IM metamodel (and the corresponding DTD or XML schema) captures the essential entities in the SPT domain model that are required for building performance models, as described in section 4.5. The LQN and CSIM metamodels describe each the elements of the respective performance models. The third level corresponds to the concrete model; and it is at this level that the actual transformation (represented as a thick arrow) takes place. In order to do the transformation of a model, we need to know the input model and the two metamodels involved (input and output), but what we actually transform is a model instance (input or source) into another model instance (output or target). The transformation rules are defined in terms of metamodel elements.

Figure 4.2 further details the proposed transformation. Conceptually, at the metamodel level, two relations could be used to specify the transformation: one between the UML metamodel extended with SPT and the Intermediate Metamodel (IMM), and the other between IMM and the Performance Metamodel. These relations are instantiated at the model level by two corresponding relations, UML_IM and IM_PerfM, respectively. All three models, UML, IM and Performance Model can be converted to XML according to the rules for document productions specified by the XMI standard [83]. It is at the XML level that the proposed transformation is implemented in the form of two mappings, UML2IM and IM2PerfM, as shown in Figure 4.2. Each such mapping implements the corresponding relation from the metamodel level, and realizes the relation from the model level. The mappings are composed of transformation rules and algorithms, which

are expressed by using the XMLgebra formalisms and are implemented in XSLT. The transformation rules specify how certain patterns found in the input (source) model are mapped to elements of the output (target) model. The algorithms are used for rule scheduling and for performing abstraction-raising aggregation by grouping elements of the source model before mapping them to the target model, if necessary.

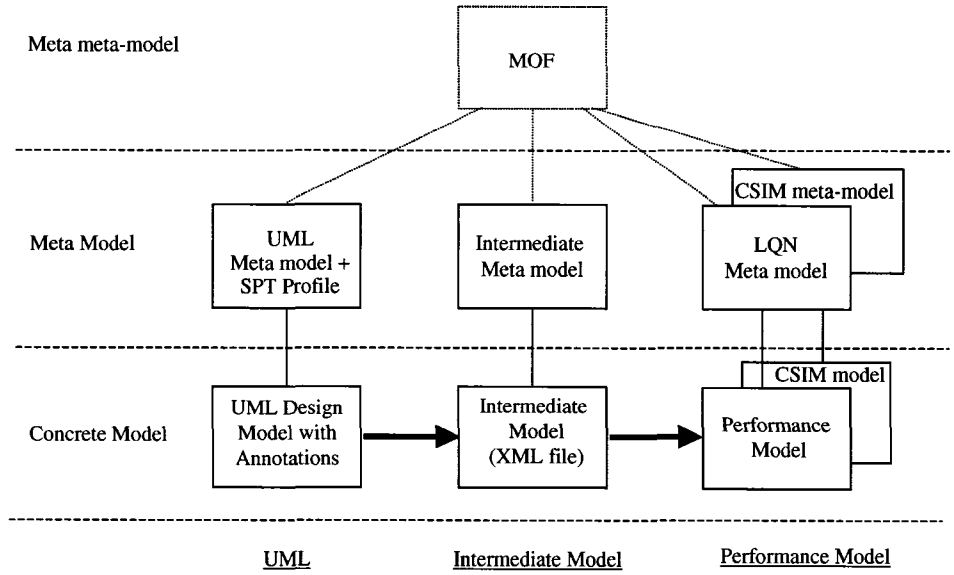


Figure 4.1: An overview of the metamodel levels in the context of the proposed transformation.

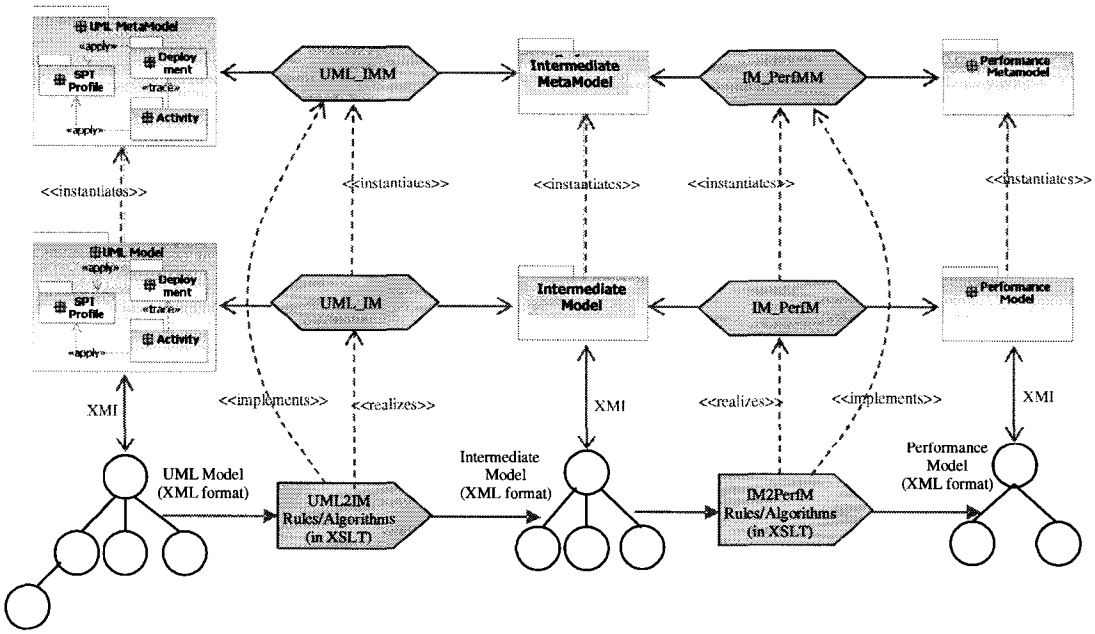


Figure 4.2: Relations and mappings in the proposed transformation.

It should be mentioned that the relations shown in Figure 4.2 at the metamodel and model level are not expressed formally in this work, only the mappings are. The way in which the transformation rules are related to the respective source and target metamodels is explained informally.

As indicated in the previous chapters, necessary performance information must be provided in the UML model before it can be transformed for performance evaluation. UML performance annotations describing relevant performance parameters, such as workload, service demands and so on, are embedded in the UML model in accordance with SPT profile definitions before the transformation steps are applied. The first step in the proposed transformation is the generation of the intermediate model (IM) file. In this chapter, we discuss the steps for transforming UML designs to performance models (LQN and CSIM). The following issues will be discussed:

- UML design with performance annotations;
- The concept of UML design to performance model transformation
- Performance information extraction;
- Intermediate model (IM) schema and construction;
- Generic transformation from intermediate model to performance models;
- IM to LQN transformation;
- IM to CSIM18 transformation.

In this chapter, an example of 3-tier client-server model is provided to illustrate the transformation between UML design and performance models. Patterns transformation, including simple client/server, pipeline and filter, and storage buffer resource patterns, are also discussed.

In the following text, words that are exchangeable unless otherwise specified include:

- "input" model and "source" model
- "output" model and "target" model
- "node" and "vertex"– a node in an XML tree structure;

4.2 UML DESIGN WITH PERFORMANCE ANNOTATIONS

The performance annotations defined in the SPT Performance Profile provide a mechanism for capturing performance requirements and performance characteristics within the UML model. Before being transformed into a performance model, the UML model must contain important system features such as the high-level software architecture patterns, the allocation of software components to hardware resources and the key performance scenarios in the system. These features are represented in UML diagrams as follows:

1. The high-level software architecture is very important in defining the system performance since it shows the software components and their cooperations. It can be represented as a component or class diagram that indicates the architectural patterns used between high-level components.
2. The deployment of the software to physical resources has a strong impact on the overall performance, so it should be clearly represented in the UML model. It can be expressed in UML as a deployment diagram.
3. The UML model should also represent the key performance scenarios of the system. Each step of the scenario uses software and hardware resources. The resource demands must be given as performance annotations. The scenarios can be represented in UML either as interaction or as activity diagrams.

4.2.1 Incorporating Performance Features into UML Model

The SPT Performance Profile extends UML by providing stereotypes and tagged values to represent performance requirements; the resources used by the system and the response paths (scenarios) are important for performance. Each scenario is composed of "scenario steps" that are making demands to resources. In this section, we use a simple 3-tier client server model (Figures 4.3a-c) to illustrate what should be represented in the UML model in order to derive a corresponding performance model.

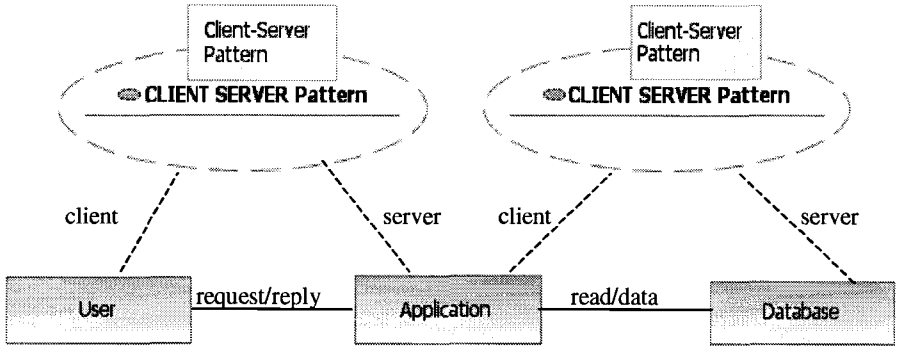


Figure 4.3(a): High-level architecture of the 3-tier client server model.

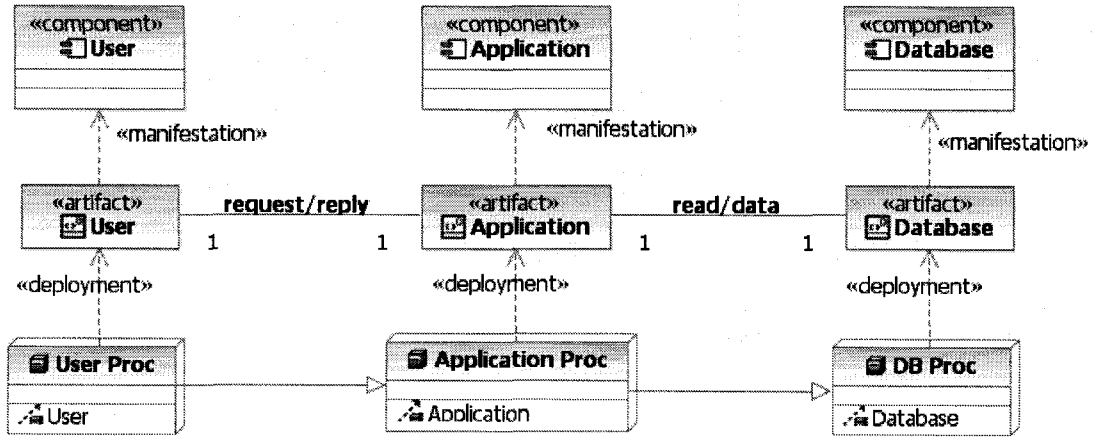


Figure 4.3(b). Deployment diagram of the 3-tier client server model.

Figure 4.3(a) represents the software architecture; it shows the high-level software components (User, Application and Database) and their relationships in the form of a Client-Server Pattern. It should be noted that the definitions of patterns are coded in the transformation algorithm for their recognition. Figure 4.3(b) is a deployment diagram, where the hardware components are linked according to the design blueprint. The software components are allocated to hardware devices as shown.

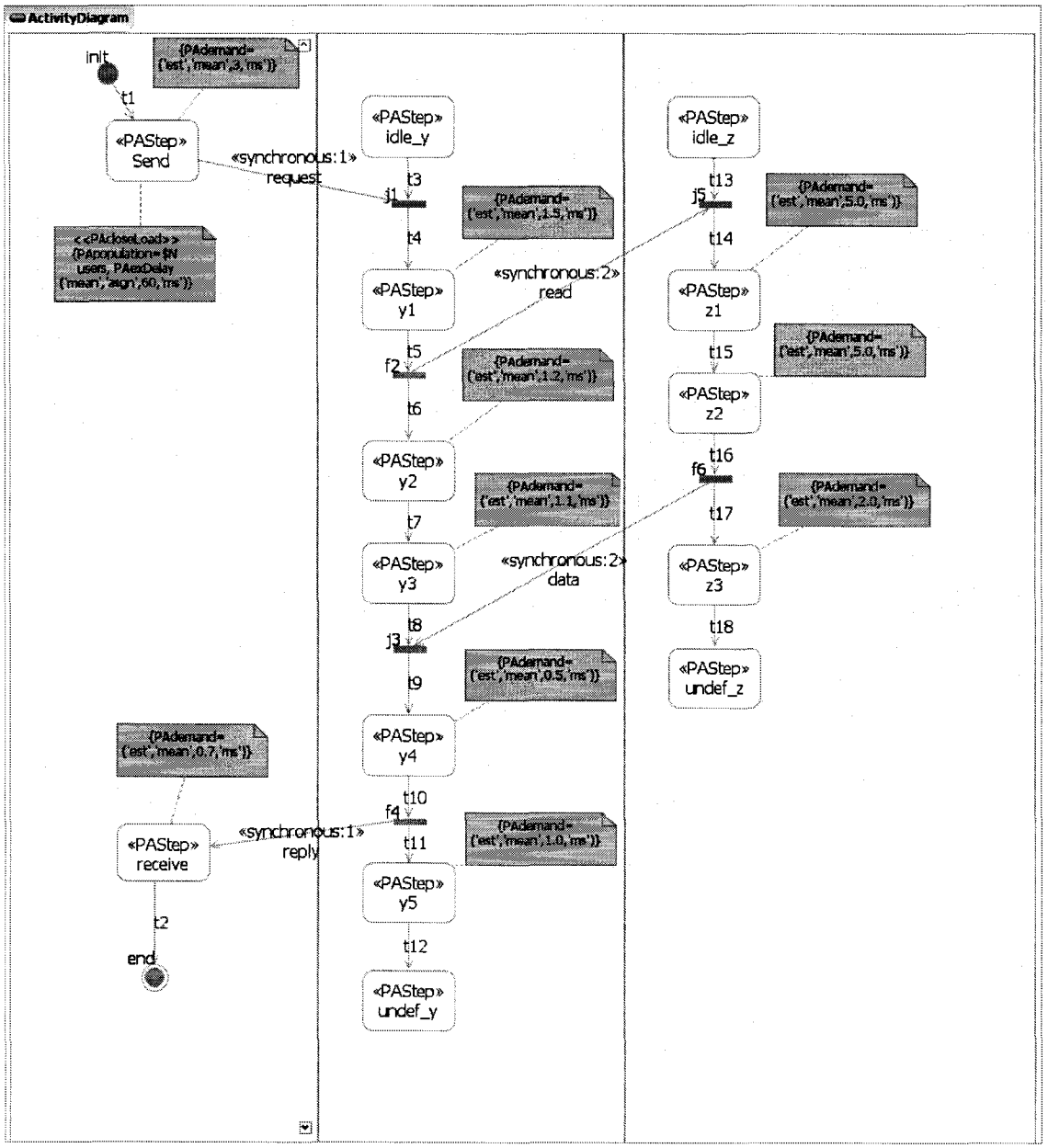


Figure 4.3 (c): A scenario represented by an activity diagram.

The activity diagram from Figure 4.3(c) represents a key scenario of the 3-tier client-server model. Each software process from Figure 4.2(a) is represented by a separated partition lane, which contains all the steps performed by that process. Inter-process messages are indicated by transitions that cross the partition border. In order to simplify the transformation, it is assumed that the names of these messages (indicated in activity diagram) are the same in the high-level architecture. The type of the message, either

synchronous or asynchronous, is also denoted in the diagram. In the activity diagram we applied the following convention to represent the sending and receiving of messages: use a "fork" to represent the sending of a message by a concurrent component to another one, and a "join" to represent the receiving of a message. The activity diagram (Figure 4.3c) depicts a scenario containing two synchronous messages (request/reply and read/data) as indicated in the architecture diagram. The performance annotations are given in stereotypes and tagged values, and notes attached in each step are for illustration.

According to the SPT Performance Profile, the starting step of a scenario has associated workload information. In this case, the note associated to the step "send", indicates a closed workload with population and an external delay of 60 ms (which represents the think time). The step "send" itself has an estimated mean service demand of 3 ms, given by the tagged value <<PAdemand>>. The mean service demands for other steps are shown in a similar way in the figure.

4.3 THE CONCEPT OF UML TO PERFORMANCE TRANSFORMATION

Software architecture, hardware deployment, execution scenario and performance attributes are the essential elements required in order to transform a UML design to a performance model. Fortunately, the required information can be directly obtained from UML architecture, deployment, activity diagrams and UML performance annotations. The architecture diagram provides task architectural structure and software pattern information, whereas the deployment diagram shows the hardware allocation. The activity diagram offers detailed steps of activities in an execution path or a scenario. A sequence diagram can be used instead of an activity diagram for the UML to performance model transformation. Sequence diagram is very good at showing the responsibilities of different objects and the linear execution of sequential steps, but does not represent well concurrent flows of control. The UML1.4 standard, which was used at the beginning of the thesis research, lacks convenient features for representing loops, branches and fork/join structures in sequence diagrams. For this reason, we have chosen to use activity diagram to model the key performance scenario. The new UML 2.0 contains enhanced sequence diagram features, which eliminate the drawbacks present in UML 1.4. Although

the work described in this thesis was carried out using UML 2.0, we still use activity diagram instead of sequence diagram to retain the continuity of the research work.

4.3.1 Transformation Steps

The transformation algorithm proposed here has two stages (see Figure 4.4), which are implemented by using the XSLT transformation tool SAXON [59].

1. Extract the UML model information including hardware and software architecture from the UML XMI file to generate an Intermediate Model (XML file) that contains the information required to build a corresponding performance model.
2. Construct a LQN descriptive (text) or a CSIM (C/C++ compile-ready) model from the intermediate model XML file. The generated performance model can be evaluated with existing tools (LQN Solver or CSIM-based engine, respectively).

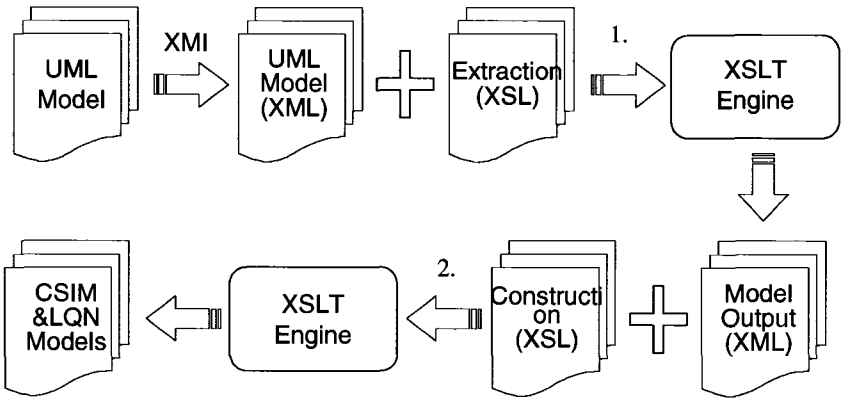


Figure 4.4: Automatic transformation approach.

Stage 1 includes data extraction, transformation and construction of the intermediate model file and stage 2 consists of the construction of the performance model from the intermediate model file. All these are XML file operations except for the last operation (XML→ Text), which is a conversion from XML file to LQN text or C/C++ description file. Since each of these operations basically converts an XML file (template) to another, it is convenient to use XMLgebra to define the details of tree-transformation rules and transformation algorithms. The next section discusses an extension of XMLgebra proposed in this work in order to cope with large templates.

4.3.2 Extended XMLgebra (eXMLgebra)

A drawback of the XMLgebra [61,62] string expression for denoting vertices is that it fails to handle unambiguously large XML templates that contain at least a vertex v with more than 10 child vertices. For instance, the string expression "1111" may be interpreted as "1 11 1", "1 1 11" or "1 1 1 1", which represent different tree vertices. In order to avoid such a confusion, we propose to replace the string expression for denoting v by an array expression $(c_1, c_2, c_3,\ldots,c_n)$, where the values of element c_i represents the position of the ancestor situated on the layer i among its own siblings (Figure 4.5). By definition, the result of the *concatenation operation* of two arrays $v.w$, where $v = (c_1, c_2, c_3,\ldots,c_n)$ and $w = (d_1, d_2, d_3,\ldots,d_m)$ is an array:

$$v.w = (c_1, c_2, c_3,\ldots,c_n, d_1, d_2, d_3,\ldots,d_m)$$

Similarly, the result of the concatenation of an array v with a natural number i is an array:

$$v{\cdot}i = (c_1, c_2, c_3,\ldots,c_n, c_{n+1}), \text{ where } c_{n+1}=i\,.$$

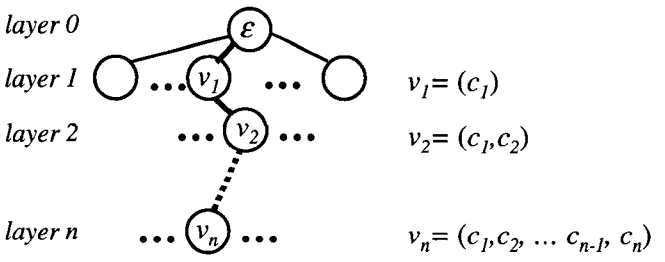


Figure 4.5: Array expression for denoting a node v.

Revised Template Domain Definition:

Let $c_1, c_2, c_3,\ldots,c_n$ belong to the set of natural numbers N, and let N be the set of arrays over N. A vertex v is denoted by an array expression $(c_1, c_2, c_3,\ldots,c_n) \in N^*$. Let S be a finite alphabet of labels, let A be a finite set of attribute names, and let D be an infinite (recursively enumerable) domain of values. Let G be a finite set of gap names, and assume that $D \cap G = 0$. A template domain τ is a subset of N*, such that:*

(1) if $v{\cdot}i \in \tau$, where $v \in N^*$ and $i \in N$, then $v \in \tau$;

(2) if $v{\cdot}i \in \tau$, where $v \in N^*$, $i \in N$ and $i > 1$, then $v{\cdot}(i-1) \in \tau$.

The empty vertex is denoted by ε that represents the root of τ, and elements in τ are called vertices. A vertex w is a child of a vertex v (and v the parent of w) if v·i = w, for some i ∈ N. All template domains are assumed to be finite.

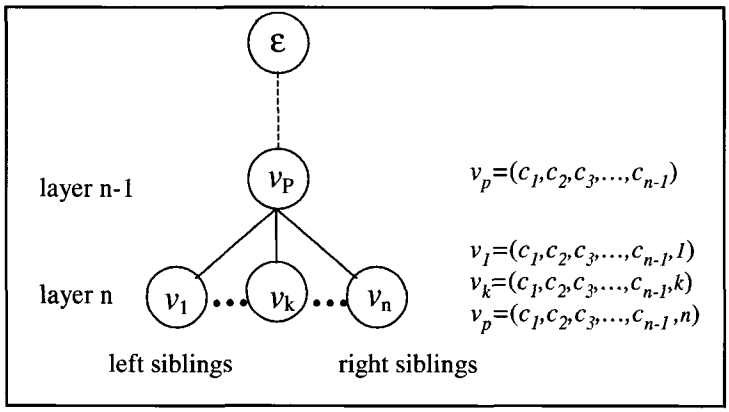


Figure 4.6: An example of a more general array expression for vertices.

The conditions (1) and (2) are similar to the template domain definition of XMLgebra, except for that the string expressions have been replaced with arrays expressions, and the string concatenation operation by array concatenation. A more general example of array expression of vertices is shown in Figure 4.6.

The XMLgebra functions discussed in Chapter 3 are still valid under the revised template domain definition. However, according to the needs of this work, some of the functions including *select* and *descendants* are slightly modified to provide a clearer definition. There are also new functions defined for expressing the transformation algorithms, which will be discussed in the following sections.

Select Function Definition:

Let $t_1 \in T_{(\Sigma A,G)}$ be a template, and let $p \in P$ be a location path. Let the vertex set: E = $evalpath_{t1}(p)= \{v_1, \ldots, v_n\}$ contain the results of location evaluation, and assume that the elements v_i are numbered in template order according to dom_{t1}. E is defined as $E = \{v \mid v=(v_p.w) \wedge w \in N^* \wedge v \in dom_{t1}\}$, where v_p is determined by path p. The select function: $select : T_{(\Sigma A,G)} \times V \rightarrow T_{(\Sigma A,G)}$ is defined as $\mathbf{select}(t_1, v_p) = (\mathbf{dom}_t ; \mathbf{lab}_t ; (\lambda_t^a)_{a \in A})$ *where:*

$$dom_t = \{\varepsilon\} \cup \cup_{vi \in E} \{w \in N^* | v_i = v_p.w \wedge v_i \in dom_{tI} \wedge w \in N^*\}$$

$$lab_t(v) = \begin{cases} Root & \textit{if } v = \varepsilon \\ lab_{tI}(v_p.w) & \textit{if } v = w; \end{cases}$$

$\forall a \in A,$

$$\lambda_t^a(v) = \begin{cases} \lambda_{tI}^a(v_p.w) \textit{ if } v = w; \\ null \quad \textit{otherwise} \end{cases}$$

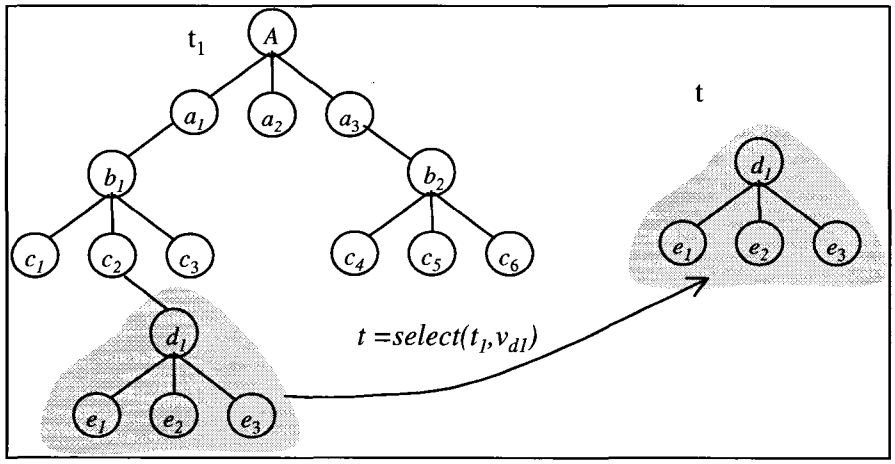


Figure 4.7: A graphical illustration of *select* function (null node ε is not shown).

The *select* operation uses the input template $t_I \in T_{(\Sigma,A,G)}$ and a node v to generate and return a new template. The function can also take an XPath $p \in P$ as input since the path p can also be expressed as $c_1, c_2, c_3, \ldots, c_n$ which is an expression of a node. The computed template has a template domain with one logical root vertex ε, and under it a sequence of sub-templates represented by prefixed subsets of dom_{tI}. The labeling function lab_t maps the root vertex to the root label, and maps all other vertices of t according to their corresponding label in t_1. The attribute function (λ_t^a, $a \in A$) only preserves vertex $v \in dom_t$ and maps an attribute ($a \in A$) according to their corresponding value in t_1. All other attributes values are *null*. Figure 4.7 shows a graphical illustration of *select* function, which returns a subtree.

The descendant function:

The descendant function returns the set of the descendants of a vertex given as parameter.

So, let $t \in T_{(\Sigma,A,G)}$ be a template, and define the *descendant vertex map* of t as the result returned by the function $descendants_t$ given as:

$descendants_t(v_i) = \{v | v=(v_i.w\ )^{\wedge}\ w \in N^* \wedge v \in dom_t\}$, which for each vertex in $v \in dom_t$ gives the set of descendant vertices.

The remove function:

The *remove* function removes a given vertex v and its descendants. Let t_1 be a template and v_o the vertex to be removed. The function $remove(t, v_o)$: $T_{(\Sigma,A,G)} \times V \rightarrow T_{(\Sigma,A,G)}$ is defined as follows:

$$dom_t = \{\varepsilon\} \cup \{ v \mid v \notin rightsiblings_{t1}(v_o) \wedge v \in dom_{t1}\} \cup \{ v=(c_1, c_2, \ldots, c_n - 1\ ) \text{ if } v \in rightsiblings_{t1}(v_o) \wedge v \in dom_{t1}\} \cup \{ v=(c_1, c_2, \ldots, c_n - 1\ )^{\wedge} w^{\wedge} v \in dom_{t1}\}$$

$$lab_t(v) = \begin{cases} Root & \text{if } v = \varepsilon \\ lab_{t1}(v) & \text{if } v \notin rightsiblings_{t1}(v_o) \wedge v \in dom_{t1} \\ lab_{t1}(c_1, c_2, \ldots, c_n + 1) & \text{if } v \in rightsiblings_{t1}(v_o) \\ lab_{t1}(c_1, c_2, \ldots, c_n + 1)^{\wedge} w & \text{if } v \in descendants_{t1}(c_1, c_2, \ldots, c_n + 1) \end{cases}$$

$\forall a \in A,$

$$\lambda_t^a(v) = \begin{cases} \lambda_{t1}^a(v) & \text{if } v \notin rightsiblings_t(v_o) \wedge v \in dom_{t1} \\ \lambda_{t1}^a(c_1, c_2, \ldots, c_n + 1)\ ) & \text{if } v \in rightsiblings_t(v_o) \\ \lambda_{t1}^a(((c_1, c_2, \ldots, c_n + 1)^{\wedge} w) & \text{if } v \in descendants_{t1}(c_1, c_2, \ldots, c_n + 1) \\ null & \textbf{otherwise} \end{cases}$$

where the $rightsiblings(v_o)$ function returns the set of vertices $v \geq v_o + 1$. So, let $t \in T_{(\Sigma;A;G)}$ be a template, and the *rightsiblings* function is defined as:

$$rightsiblings_t(v_o) = \{v | v = v_o + k, k = 1,2,3\ldots,^{\wedge} v \in N^* \wedge v \in dom_t\}$$

The function $remove(t, v_o)$ removes node v_o and its descendants.

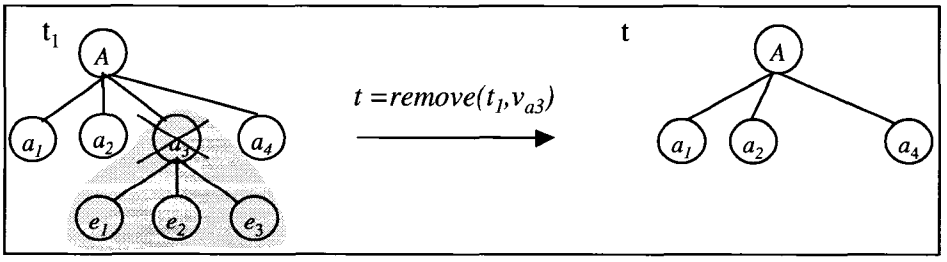


Figure 4.8: A graphical illustration of *remove* function (null node ε is not shown).

The gapifyAChild function:

The *gapifyAChild(t, v, g)* function creates a gap under the vertex $v = (c_1, c_2, \ldots, c_n)$ which has k child vertices . The function $T \times V \times G \rightarrow T$ is defined as:

$$dom_t = \{v \mid v \in dom_{t1}\} \cup \{(c_1, c_2, \ldots, c_n, k+1)\}$$

$$lab_t(v) = \begin{cases} lab_{t1}(v) & \text{if } v \neq (c_1, c_2, \ldots, c_n, k+1) \\ GAP & \text{if } v = (c_1, c_2, \ldots, c_n, k+1) \end{cases}$$

$\forall a \in A$:

$$\lambda_t^a(v) = \begin{cases} \lambda_{t1}^a(v) & \text{if } v \neq (c_1, c_2, \ldots, c_n, k+1) \\ g & \text{if } v = (c_1, c_2, \ldots, c_n, k+1) \\ null & \text{otherwise} \end{cases}$$

Different from the *gapify(t, p, g)* defined earlier, this function creates a gap under an existing vertex $v \mid v \in T$. If v has k existing child vertices, e.g., $v_k = (c_1, c_2, \ldots, c_k)$ where $c_k = k$, the gap will be the $v_{k+1} = (c_1, c_2, \ldots, c_k, c_k+1)$ *where* $c_k+1 = k+1$.

4.4 PERFORMANCE INFORMATION EXTRACTION

4.4.1 UML XML Model File

The XMI support tool from IBM RSA produces the XML files containing the input UML model according to the XMI interface. The UML model has a tree structure, like other XML files, and the node types correspond to UML metamodel classes. Figure 4.9 shows a section of the UML 2.0 design model in tree structure. The performance related design information resides in subtrees a few levels below the root. Within the "uml:Model" tree, there are subtrees such as "ownedMember" and "packageImport". The "ownedMember" has an attribute called "xmi:type" that further distinguishes "uml:Node", "uml:Activity", "uml:Class", "uml:Artifact" and so on for deployment, activity, class, and component diagrams, respectively. Moreover, an "edge" is a transition and a "group" is a partition and a "node" is an action step in the activity diagram etc. Each of these consists of a number of subtrees that contain data and values to generate a performance model. For example, the activity diagram nodes are grouped in a number of important subtrees, including "edge", "group", "node" as shown in Figure 4.10. Furthermore, many nodes contain references to other nodes through attributes named "xmi:id". For example, an "edge" references to its source and target node; and a "node" refers to its incoming and

outgoing transition. The traversal of the activity diagram makes use of these node references. Comparing with activity diagram, the subtree structure for component and deployment diagram are relatively simple and the extraction of the corresponding information is relatively straightforward.

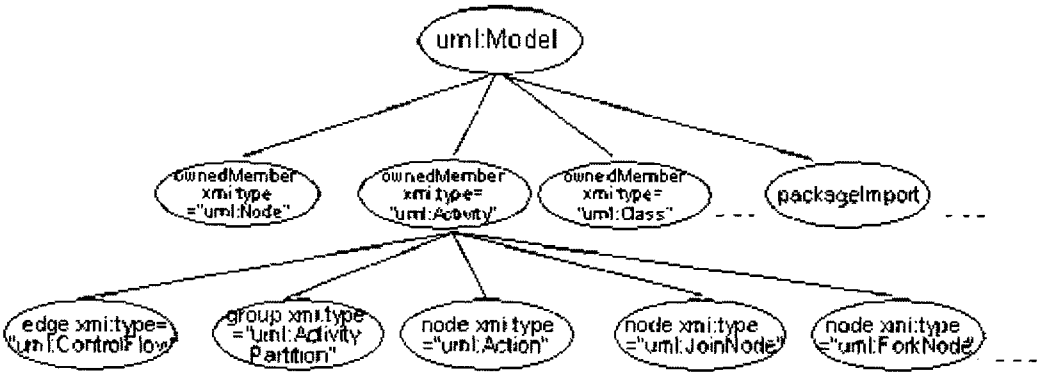


Figure 4.9: XML tree structure for a UML Model.

The UML XML design model is lot more complex than what is shown in Figure 4.9. Subtrees of use case and sequence diagrams are not discussed here, since they are not used in this transformation. Only the subtrees related to the proposed transformation are briefly mentioned here.

4.4.2 Template Extraction

The first operation of the transformation process is to extract required information from the UML XML file. Let a UML XML model be expressed as a template, $T(uml)_{(\Sigma,A,G)}$, from which we can easily find a set T_p of smaller templates that contain the information necessary to build the performance model. Let this set be $T_p = \{T_{p1}, T_{p2}, T_{p3} \ldots\ldots T_{pi}\}$ where $T_p \in T(uml)$. It is possible to express T(uml) using $T_{(\Sigma,A,G)}$, however, this is not necessary since we can obtain absolute paths to all the templates $T_{p1}, T_{p2}, T_{p3} \ldots\ldots T_{pi}$, and use the *select* function *select(t, p)*: $T_{(\Sigma,A,G)} \times P \rightarrow T_{(\Sigma,A,G)}$ to obtain the templates that contain the required information. For example, to extract an "edge" template $T_{E(\Sigma,A,G)}$ from a UML XML file, we use the following function:

$$select(t_{UML}, v_{edge}):T(uml)_{(\Sigma,A,G)} \times V_{edge} \rightarrow T_{E(\Sigma,A,G)}$$

where v_{edge} is indicated by p_{edge}=*.../ownedMember/edge.* Besides the edge template other templates that correspond to model elements of interest can be also extracted from the

UML XML file, such as: group T_G, action node T_A, hardware node T_N, comment (attached notes) T_C, and stereotype T_S.

In some cases, the selected templates contain other nodes that are not required. To remove unwanted nodes along with their children at the current level, we can apply the following *removeall* algorithm.

```
removeall(t, s){
  if(v_o = findAunwantedNode(t,s))==null) return t;
  t' = remove(t, v_o);
  removeall(t' s);
}
findAunwantedNode(t,s){
  v = first(t);i=1;
  while(u = v·i∈ t ){
    if(lab(u) = s| λ_t^a (u) = s) return u;
    i=i+1 }
}
```

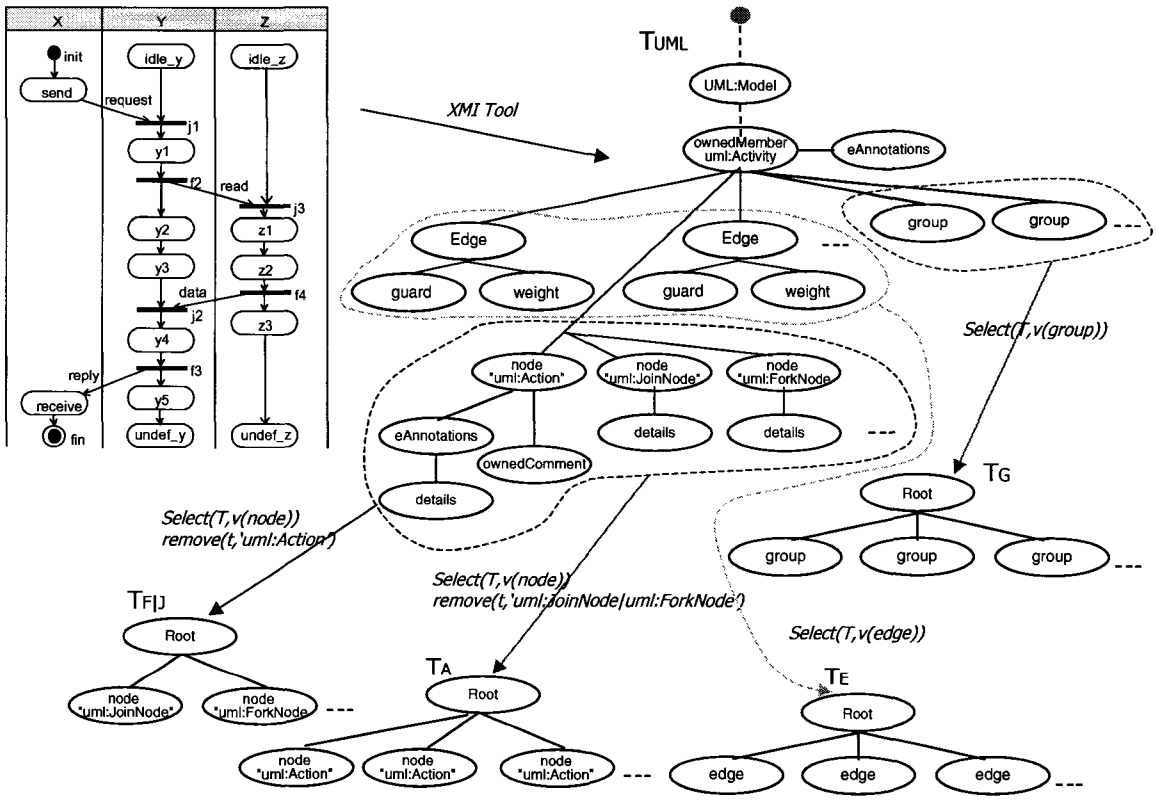


Figure 4.10: An illustration of the *select-remove* process.

where 's' is a string name and s ∈ SN, a string set of node names. Basically, the algorithm finds a node whose name or test attribute is 's' and uses the *remove* function to remove the unwanted node and its descendants. The algorithm stops when no unwanted node is found any more. Using the 3-tier client sever model as an example (top left), Figure 4.10 illustrates the *select-remove* process. The circled areas are selected subtrees, which contains "node", "group" and "edge". By applying the *removeall* operation with a string parameter 's' = 'uml:Action', we can obtain 'uml:ForkNode' and/or 'uml:JoinNode' subtrees. The template selected from the source model will be transformed into IM target model template, as described in section 4.5.2.

4.4.3 Call Sequence and Type in Activity Diagram

Another issue in data extraction is to obtain inter-task calls represented by transitions that cross partition boundary. Call information must be obtained in order to build the performance model. Figure 4.11 shows some sequence of calls that may possibly take place in software designs. A solid line arrow represents a synchronous call that has a reply and a dashed line arrow stands for an asynchronous call that does not have a reply. A synchronous call may be blocking (where the client blocks after sending, waiting for the reply) or non-blocking (where the client continues to do something else before accepting the reply). For example, case (1) involves two separated calls from A to B and cases (2) to (4) have both synchronous and asynchronous calls in various sequences. Cases (5) to (6) are calls involving three tasks, and particularly, case (7) contains interleaved calls and (8) nested calls. To sort out these different call scenarios, two pieces of information must be obtained, i.e., call sequence and type. The former can be obtained by traversing the execution path of the respective scenario; however, the solution for the latter is not so obvious. Simply by looking for a reply call to determine if an earlier call is synchronous could be ambiguous such as in case (2) or (4) shown in Figure 4.11. In our approach, stereotypes (synchronous / asynchronous) are used to denote the call type in the design stage. In the data extraction, a cross boundary transition is extracted along with its corresponding stereotype. If it is a synchronous call, the reply call will be searched with the following conditions:

1. Any call must either have a reply (synchronous) or no reply (asynchronous or notification).
2. The source and target tasks of the request arc must be the same as the target and source tasks of the reply arc, respectively.
3. The request arc takes the closest reply arc in call sequence, which satisfies (2) as its reply arc.

It should also be noted that an arc is only recorded once along with the requesting task in the intermediate model.

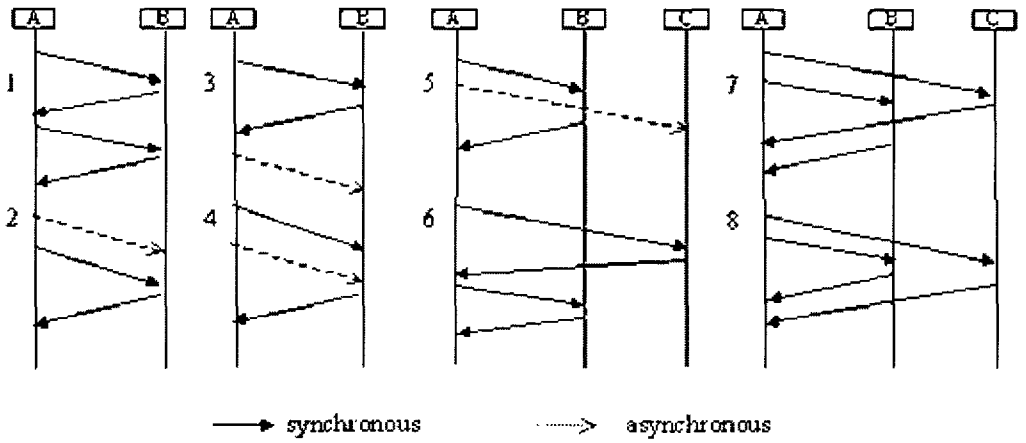


Figure 4.11: A schematic representation of different sequences of calls.

4.5 INTERMEDIATE MODEL (IM) SCHEMA AND CONSTRUCTION

4.5.1 Intermediate Model (IM) Template Definition

One of our goals is to develop a generic flexible transformation method, easy to specialize in order to accept several kinds of input models (such as UML 1.4 and UML 2.0) and to generate different kinds of output models. The PUMA project [122] makes a strong argument for solving this type of N-by-M problem by introducing a common intermediate format, which is named the Core Scenario Model (CSM). PUMA's CSM captures the essence of performance specifications from a UML design as expressed in the SPT Profile, and strips away the design detail that is irrelevant to performance analysis. In this work, the Intermediate Model (IM) was developed in parallel with CSM and has many of the same features, as it is also based on the SPT Profile. IM has been developed separately from CSM for practical reasons, to allow for independent work by

different researchers in the same group. The main different between IM and CSM is that the former is a task-based model while the latter is a scenario-oriented one. Both approaches have their advantages. It is possible for IM to merge with CSM in the future, but it is out of the scope for this work.

The importance of the intermediate model (IM) rests on the fact that it does not only contain essential performance information as defined in the SPT performance profile, but also captures the system core scenarios to provide a generic starting point for further transformation to different performance models. This is because scenarios are often used to describe system functionality and provide the basis for analyzing system performance. Therefore, the schema of the intermediate model should meet the following requirements:

- It must be based on the OMG UML Profile for Schedulability, Performance and Time.
- It should allow for the specification of different kinds of scenarios between multiple concurrent tasks.
- It should be generic and adaptable to the generation of a variety of performance models including QN, LQN and simulation models.

Intermediate Model Schema:

Figure 4.12 shows the schema of the intermediate model in a tree structure. The "IM:Model" contains "IM:HardwareEntity" (hardware devices and processors), "IM:TaskEntity" (software components), "IM:ServiceEntity" (scenario section set) and "IM:PatternInformation". The "IM:Task", "IM:Service" and other smaller elements such as "IM:Step", "IM:Fork|Join" and "IM:TransitionArc" are also included. The intermediate model expressed in the DTD is given in Appendix I. Unlike CSM whose steps are grouped in accordance with scenario regardless of the tasks and hardware devices executing the steps, the intermediate model is a task-based model and it puts action steps together by the task executing them.

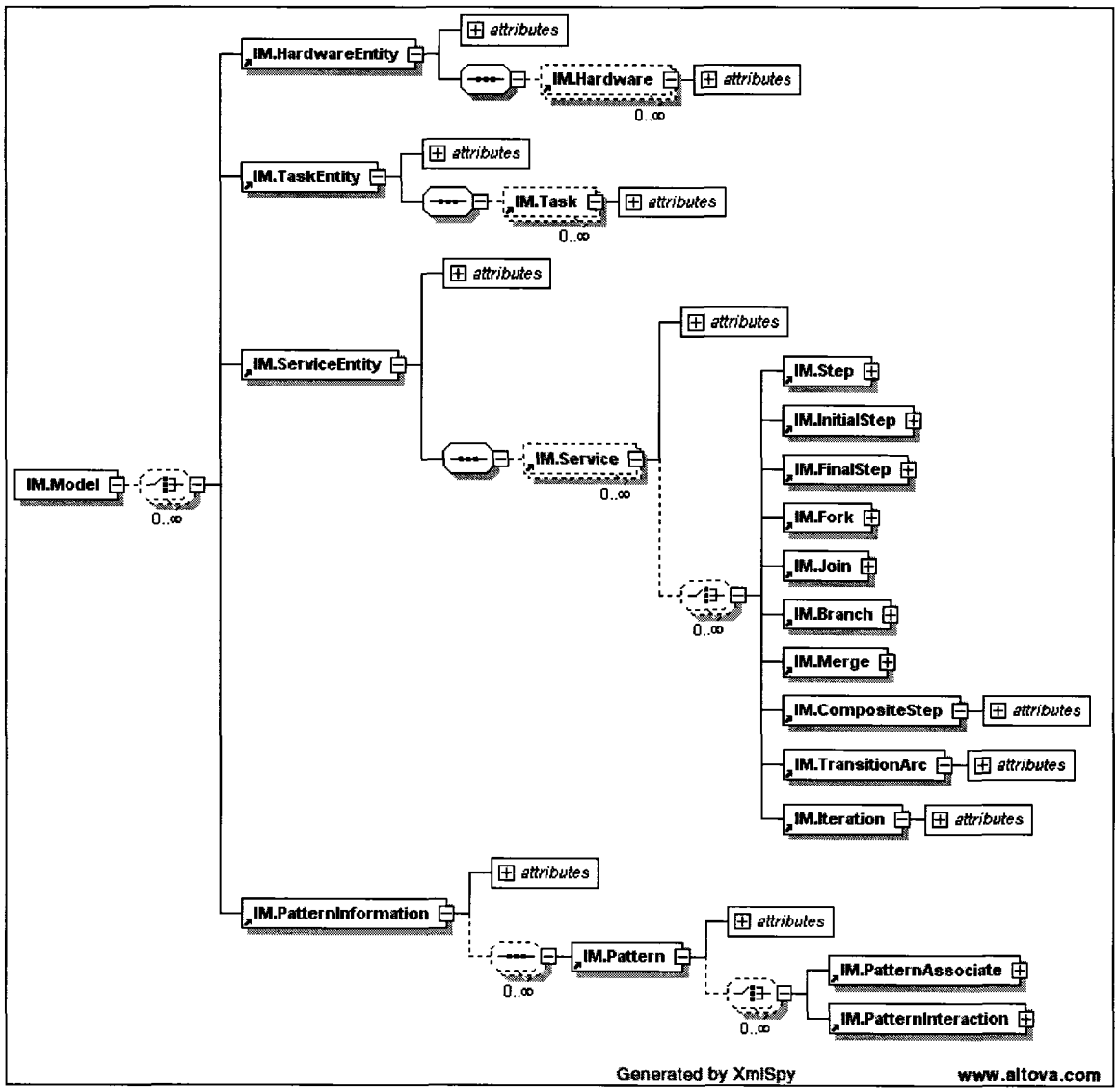


Figure 4.12: XML schema of the intermediate model.

Gluing Attributes:

Grouping the execution steps by task facilitates the data extraction, but it may introduce some difficulty in building the performance model, since the execution scenario must be reconstructed. To ease this problem, an attribute named "pname" is introduced as the gluing attribute to remember the parent node. Moreover, "IM:From" and "IM:To" are also used to record execution sequence in "IM:Step", "IM:Join" and "IM:Fork", as well as attributes such as 'source' and 'target' in "IM:TransitionArc". In the case of joining two or more scenarios (represented by different activity diagrams), "IM:CompositeStep" that has a reference to "IM:Service" is used to handle the aggregation of steps.

4.5.2 Template Transformation and Rules

The extracted templates cannot be directly used to construct the intermediate model template since their structure and namespace are different. Therefore, we need to define template conversion rules that dictate the structure and namespace transformation.

Transformation Rule Definition:

A transformation rule is defined such that a vertex or node (LHS of the rule), from a domain compliant with a DTD, can be transformed to a null, vertex or tree (RHS of the rule) of another domain (compliant with another DTD) if the transformation is 1-to-0, 1-to-1 or 1-to-n transformation, respectively. The transformation is done through a "select-apply" function, which is defined as:

$$w\ (V \rightarrow W) = \textit{select-apply}(v)\ \textit{where}\ w = \textit{null}, v, t;\ \textit{lab}(v) \rightarrow \textit{lab}(w);\ \lambda^a(v) \rightarrow \lambda^a(w).$$

Where V and W are collection sets of source and target nodes respectively. The symbol "→" is an expression of the transformation. A $v \rightarrow w$ transformation is done via the *select-apply* function, which takes a source node to map to a target node. The return of the function can be null, a vertex or a template (tree). A transformation rule defines both label and attribute conversions.

Transform Function:

The transform function realizes an "in place" tree rewriting, used to transform a source template into a target template by substituting one node at a time. Let the intermediate model (in XML format) be expressed as a template $T'_{(\Sigma A,G)}$ and define a set of templates $T'_p = \{T'_{p1}, T'_{p2}, T'_{p3}\ .....T'_{pi}\}$ where $T'_p \in T'_{(\Sigma A,G)}$ such that T'_{pi} corresponds to a template from the source model $T_{pi} \in T(UML)_{(\Sigma A,G)}$, obtained from the extraction operation. Each template extracted from a UML design template can be transformed to an IM target template. Let the transform rules be defined in $M\ (V \rightarrow W)$, and the function $\textit{select-apply}(v \mid v \in T_{(\Sigma,A,G)}) \rightarrow w$, $w \in T'_{(\Sigma,A,G)}$. (Note that *select-apply*(v) returns *null* if the target node does not exist). Also, let $v = (c_1, c_2, \ldots, c_n)$ where n is the number of layers in the template and c is the position of each child vertex in its layer. Then, the function *transform:* $T_{(\Sigma,A,G)} \times V \times M \rightarrow T'_{(\Sigma,A,G)}$ is defined as $\boldsymbol{transform(t_1, v, m) = (dom_t ; lab_t ; (\lambda_t^a)_{a \in A})}$ *where:*

$$dom_t = \begin{cases} dom_{t1} & \textbf{if } select\text{-}apply(v) = w \quad //one\ to\ one\ mapping \\ dom_{t1} \cup \{w = (c_1, c_2, \ldots, c_n, (k+i)\cdot u'),\ u' \in dom_{t2} & \\ \qquad \textbf{if } select\text{-}apply(v) = t_2, \quad // one\ to\ n\ mapping & \\ \{\varepsilon\} \cup \{w \mid w = v' \mid v' \notin descendants_t(v) \wedge v' \notin rightsiblings_t(v) \wedge & \\ \quad v' \in dom_{t1} \cup \{w = (c_1, c_2, \ldots, c_{n-1}, x)\} \cup \{(c_1, c_2, \ldots, c_{n-1}, x)\cdot u \wedge & \\ \quad u \in N^* \} \textbf{ if } select\text{-}apply(v) = null,\ u \in dom_{t1}\ //one\ to\ null\ mapping & \end{cases}$$

where k is the number of child vertices v has and i is the number of child vertices increased due to transformation; x is defined: number of left siblings < x ≤ number of upgraded child vertices + number of rightsibling vertices.

$\forall w \in dom_t$

case: $select\text{-}apply(v) = w$ *//one to one mapping*

$$lab_t(w) = \begin{cases} lab_{t1}(w) & \textbf{if } w \neq select\text{-}apply(v) \\ select\text{-}apply(v) & \textbf{otherwise} \end{cases}$$

case: $select\text{-}apply(v) = t_2$, *// one to n mapping*

$$lab_t(w) = \begin{cases} lab_{t1}(w) & \textbf{if } w \in dom_{t1} \\ lab_{t2}(first(t_2)) & \textbf{if } w = v \\ lab_{t2}(u \mid u = 1\cdot u' \in dom_{t2}) & \textbf{if } w = (c_1, c_2, \ldots, c_n, (k+i)\cdot u') \end{cases}$$

case: $select\text{-}apply(v) = null$ *//one to null mapping*

$$lab_t(w) = \begin{cases} lab_{t1}(w)\ if\ w \notin descendants_t(v) \wedge v' \notin rightsiblings_t(v) & \\ lab_{t1}(c_1, c_2, \ldots, c_n, k) & \textbf{if } w = (c_1, c_2, \ldots, c_n\text{-}1 + k)\ //upgraded\ children \\ lab_{t1}(c_1, c_2, \ldots, c_n + j) & \textbf{if } w = (c_1, c_2, \ldots, c_n + j)\quad //right\ siblings \\ lab_{t1}((c_1, c_2, \ldots, c_n, k)\cdot u) & \textbf{if } w = (c_1, c_2, \ldots, c_n\text{-}1 + k)\cdot u\quad //child\ of\ u.c. \\ lab_{t1}((c_1, c_2, \ldots, c_n + j)\cdot u) & \textbf{if } w = (c_1, c_2, \ldots, c_n + j)\cdot u\quad //child\ of\ r.s. \end{cases}$$

$\forall a \in A$

$$\lambda_t^a(w) = \begin{cases} \lambda_t^a(select\text{-}apply(v)) & \textbf{if } w = v \mid v \in dom_{t2} \\ \lambda_{t1}^a(v) & \textbf{otherwise} \end{cases}$$

The *transform* function converts the input node within the input template according to the transformation rule (m) via the *select-apply* function. There are three different situations depending on what is returned from the *select-apply* function. If a node is returned, then there is no structural change. However, if null or a tree is returned, the resulting template structure is altered. In the former case, the current node (to be transformed) is deleted and

its children are upgraded while, in latter, the current node is replaced by the returned tree as indicated in Figure 4.13.

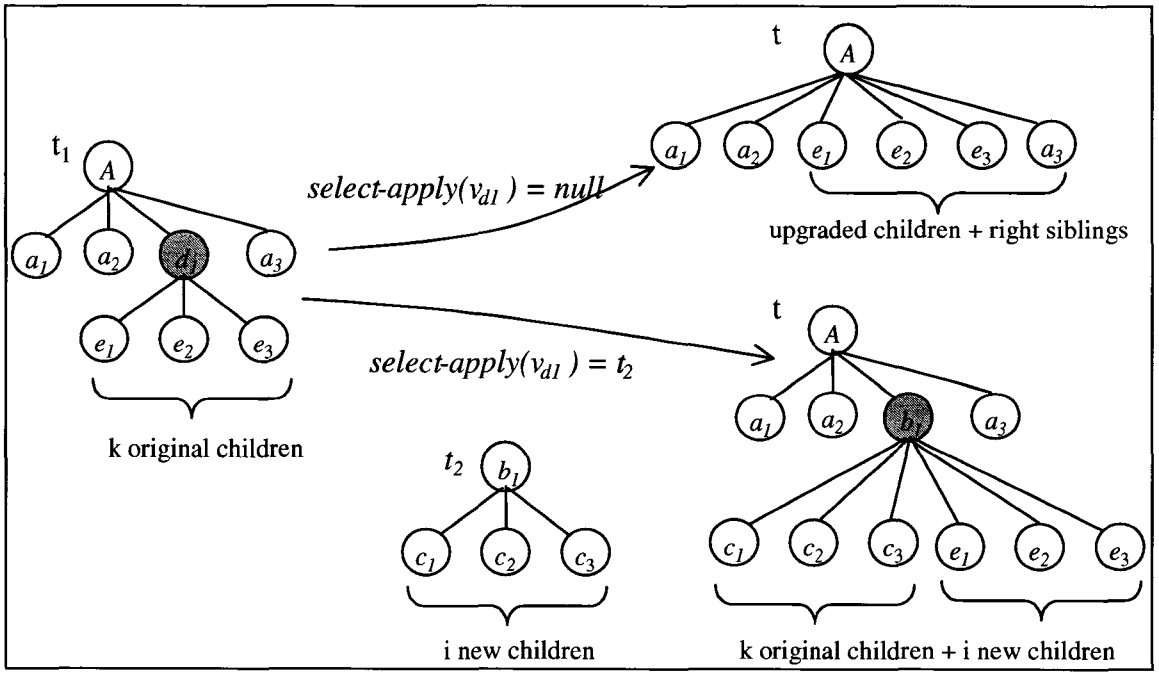


Figure 4.13: A graphical illustration of *transform* function (null node ε is not shown).

Transform Algorithm:

The transform function converts a node from a domain to another. The following simple algorithm can be used to transform an entire template from a domain to another. The algorithm takes the first vertex of the input template and converts it into a new vertex according to the transformation rule, then repeat the operation recursively with all the child vertices until all the vertices are transformed.

```
transformatemplate(t, t_o, v){
 if(v ∉ dom_to ) return t; // do not transform
 t' = transform(t,v,m);
 v' = select-apply(v);
if(descendants_t'(v') = {0}) return t';
 else{
   ∀v_i | v_i ∈ descendants_t(v'){
     transformatemplate (t', t_o, v_i);
   }
}
```

where t is the working template and t_o is the original template. At the beginning, the vertex v is assigned to be the first vertex of the template t. The algorithm returns a transformed template t'.

Individual Mapping Rules:

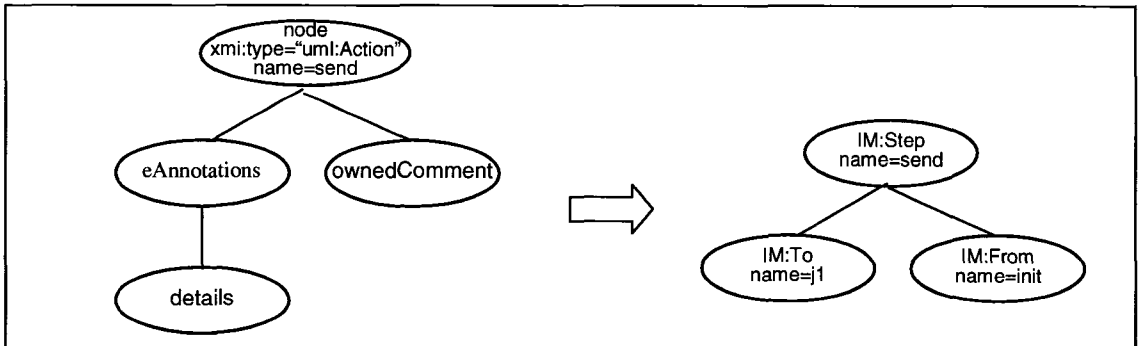
In the data extraction, we obtained the following kind of templates: "node(uml:Action)" (T_A), "node(uml:ForkNode| JoinNode)" ($T_{F|J}$), "group" (T_G), "edge" (T_E), "Node" (T_N) templates etc. To convert all these templates from the UML domain (structure and namespace) to the IM domain, specific transformation rules are defined. Let m $\in$ M, where M is a set of transformation rules. Each rule defines both vertex and attributes transformations between two different template domains. Transformation rules that are used for mapping UML to IM concepts include:

$T(A)_{(\Sigma,A,G)} \times M(A) \rightarrow T(Step)_{(\Sigma,A,G)}$,

$T(F|J)_{(\Sigma,A,G)} \times M(F|J) \rightarrow T(Fork|Join)_{(\Sigma,A,G)}$,

$T(N)_{(\Sigma,A,G)} \times M(N) \rightarrow T(Hardware)_{(\Sigma,A,G)}$,

$T(G)_{(\Sigma,A,G)} \times M(G) \rightarrow T(Service)_{(\Sigma,A,G)}$, etc.



node(xmi:type, xmi.id, name, outgoing, incoming, inPartition) →
IM:Step(name, taskname, servicetime, prestate, pname) if $\lambda^{xmi:type}(v\text{node})$ = 'uml:Action'<(IM:To(name, pname) if $\lambda^{outgoing}(v\text{node}) \neq$ null), (IM:From(name, pname) if $\lambda^{incoming}(v\text{node}) \neq$ null)>;
eAnnotations(xmi:id, source)→null();
details(xmi:id, key)→null();
ownedComment(xmi:id, body, annotatedElement)→null();

$\lambda^{name}(v\text{step}) = \lambda^{name}(v\text{node})$;
$\lambda^{taskname}(v\text{step}) = \lambda^{name}(v\text{group}| \lambda^{xmi.id}(v\text{group}) = \lambda^{inPartition}(v\text{node}))$;
$\lambda^{servicetime}(v\text{step}) = \lambda^{body}(v \mid v\text{ownedComment} \in \text{descendants}(v\text{node}))$;
$\lambda^{name}(v\text{from}) = \lambda^{name}(v \mid \lambda^{incoming}(v\text{node}) = \lambda^{outgoing}(v))$;
$\lambda^{name}(v\text{to}) = \lambda^{name}(v \mid \lambda^{outgoing}(v\text{node}) = \lambda^{incoming}(v))$;

Figure 4.14: The "node(uml:Action)"→ "IM:Step" rules.

Figure 4.14 shows the "node(uml:Action)"→ "IM:Step" rule set, which indicates that a "node" vertex (that has five attributes) is converted to an "IM:Step" vertex (which has also five attributes) if the xmi:type = "uml:Action". An "IM:To" vertex is created if the "outgoing" attribute of the "node" is not null. Similarly, an "IM:From" vertex is generated if the "incoming" attribute of the "node" is not null. An attribute is always defaulted to null unless a value is explicitly assigned. The "eAnnotations", "details" and "ownedComment" vertices are converted to null. (A null mapping denotes a vertex that does not map to any vertex in the intermediate model). The attribute rule set indicates the attribute value conversion as expressed in the λ functions. Some of the attribute value assignments are obvious but others may be indirect. For example, $\lambda^{name}(v_{step}) = \lambda^{name}(v_{node})$ means that the name of the "node" is assigned to the name of "IM:Step".

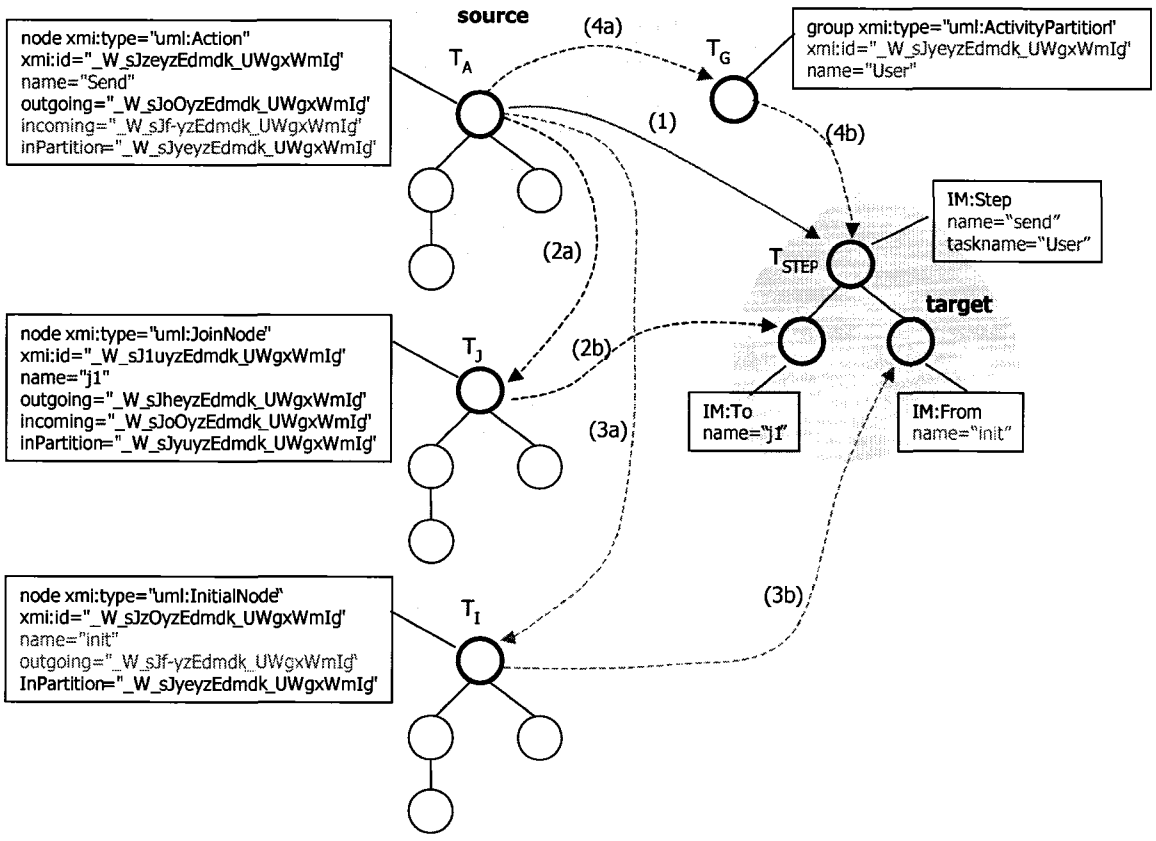


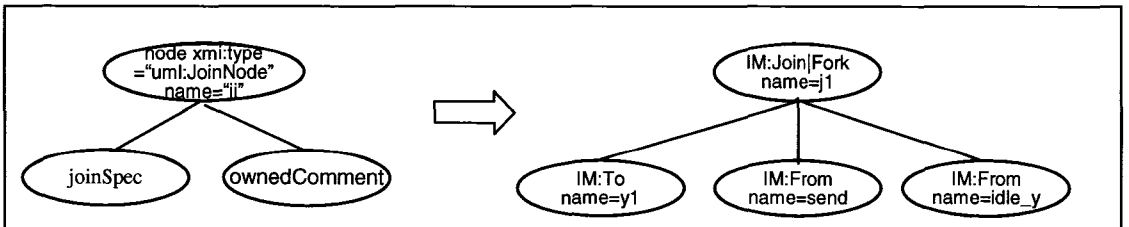
Figure 4.15: Example of how attribute values are assigned in transformation.

Figure 4.15 further illustrates how the rules shown in Figure 4.14 work in node type and attribute transformations. The example is to transform a "node(uml:Action)" to an "IM:Step". Four steps are shown in different colors to detail the attribute transformations.

Step 1: The "node (xmi:type = "uml:Action")" is transformed to "IM:Step" vertex and name "User" is assigned to it. Also, since both incoming and outgoing attributes are not null, "IM:From" and "IM:To" are generated.
Step 2: The outgoing attribute of the "node" is a reference id, from which the name of the outgoing vertex can be obtained (see 2a). The obtained name is assigned to attribute name of "IM:To" (see 2b).
Step 3: The incoming attribute of the "node" is a reference id, from which the name of the incoming vertex can be obtained (see 3a). The obtained name is assigned to attribute name of "IM:From" (see 3b).
Step 4: The inPartition attribute of the "node" is a reference id, from which the name of the group vertex can be obtained (see 4a). The obtained name is assigned to the attribute task name of "IM:Step" (see 4b).

The above example only shows one "node" → "IM:Step" transformation. Let $T_{step} = \{v_j \mid lab(v_j) = IM:Step, j = 1,2,3,\ldots\}$ be a collection set that contains all the "IM:Step" subtrees (subtemplates), which is used in next step in constructing the intermediate model.

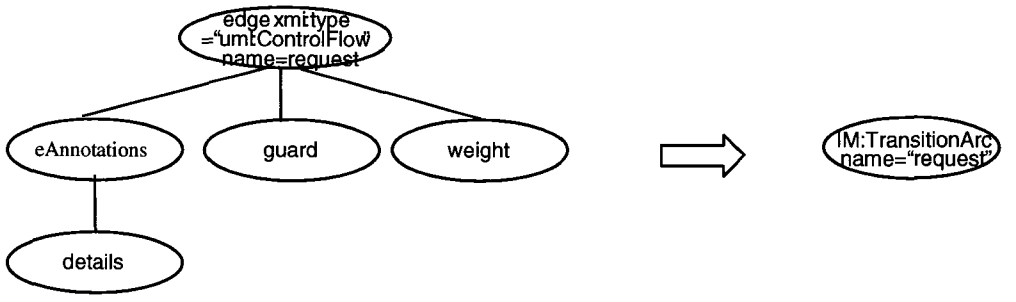
The "node(xmi:type = "uml:JoinNode")" or "node(xmi:type = "uml:ForkNode")" → "IM:Join" or "IM:Fork" rule set indicates that a "node" vertex (that has five attributes) is converted to an "IM:Join" or "IM:Fork" vertex depending on the value of the xmi:type attribute (Figure 4.16). Same as the "node(xmi:type = "uml:Action")" → "IM:Step" rule set, the attributes outgoing and incoming are used to generate "IM:From" and "IM:To" vertices depending on their contents. Also, let $T_{join|fork} = \{v_j \mid lab(v_j) = IM:Join|IM:Fork, j = 1,2,3,\ldots\}$ be a collection set that contains all the "IM:Join" and "IM:Fork" subtrees.



node(xmi:type, xmi.id, name, outgoing, incoming, inPartition) →
{ IM: Join(name, taskname, prestate, pname), if $\lambda^{xmi:type}(v\text{node})$ = 'uml:JoinNode' <(IM:To(name, pname) if $\lambda^{outgoing}(v\text{node}); \neq$ null), (IM:From(name, pname) if $\lambda^{incoming}(v\text{node}); \neq$ null)>
IM: Fork(name, taskname, prestate, pname), if $\lambda^{xmi:type}(v\text{node})$ = 'uml:ForkNode' <(IM:To(name, pname) if $\lambda^{outgoing}(v\text{node}); \neq$ null), (IM:From(name, pname) if $\lambda^{incoming}(v\text{node}); \neq$ null)>
joinSpec(xmi:type, xmi:id)→null();
ownedComment(xmi:id, body, annotatedElement)→null();

$\lambda^{name}(v\text{Join|Fork}) = \lambda^{name}(v\text{node})$;
$\lambda^{taskname}(v\text{Join|Fork}) = \lambda^{name}(v\text{group}|\ \lambda^{xmi.id}(v\text{group}) = \lambda^{inPartition}(v\text{node}))$;
$\lambda^{name}(v\text{from}) = \lambda^{name}(v \mid \lambda^{incoming}(v\text{node}) = \lambda^{outgoing}(v))$;
$\lambda^{name}(v\text{to}) = \lambda^{name}(v \mid \lambda^{outgoing}(v\text{node}) = \lambda^{incoming}(v))$;

Figure 4.16: The "node(xmi:type = "uml:JoinNode")"→ "IM:Join" rules.



Edge(xmi:type,xmi:id,name,source,target,inPartition)→
IM:TransitionArc(name,sourcetask,targettask,source,target,arctype,calltype)
eAnnotations (xmi:id, source) →null();
details(xmi:id, key) →null();
guard(xmi:type, xmi:id)→null();
weight(xmi:type, xmi:id, value) →null();

$\lambda^{name}(v\text{transitionarc}) = \lambda^{name}(v\text{edge})$;
$\lambda^{source}(v\text{transitionarc}) = \lambda^{source}(v\text{edge})$; $\lambda^{target}(v\text{transitionarc}) = \lambda^{target}(v\text{edge})$;
$\lambda^{sourcetask}(v\text{transitionarc}) = \lambda^{name}(v)|\ \lambda^{name}(v\text{group}) = \lambda^{inPartition}(v\text{edge})$;
$\lambda^{targettask}(v\text{transitionarc}) = \lambda^{name}(v)|\ \lambda^{name}(v\text{group}) = \lambda^{inPartition}(v\text{edge})$;

Figure 4.17: The "edge" → "IM:TransitionArc" rules.

When an edge is a call that crosses different partitions, the edge is transformed to an "IM:TransitionArc". Depending on whether it is a request or a reply call, then the "edge"

transforms to "IM:TransitionArc" with an attribute "calltype" = "request" or "reply", respectively. Figure 4.17 shows the rule set for "edge"→ "IM:TransitionArc".

There are still more transformation rules than that are expressed above, for instance, "ownedMember(xmi:type="uml:Node")" → "IM:Hardware", "group" → "IM:Service" and so on. With all the small templates ready, large templates can be constructed using both template and string plug functions.

4.5.3 Determination of Execution Sequence in Intermediate Model

In the above extraction and transformation operations, the execution sequence shown in the UML activity diagram is not explicitly expressed. Without the scenario execution step sequence, it is impossible to construct a target performance model. Therefore, the "IM:Service" template in the intermediate model must reflect the execution scenario step sequence that is contained in the UML activity diagram. In constructing the "IM:Service" template, the information contained in "group" (partition) template must be revisited. The algorithm shown below determines what kind of gap to generate according to the scenario steps in the partition template. Starting from the initial vertex of each partition, the algorithm traverses each element in the partition template and creates gaps and plugs in the step templates, the fork and join templates, and the transition templates that cross partition borders (i.e., correspond to inter-process messages). Each of the plugged templates contains information on its predecessor and successor nodes to allow that the intermediate model maintains the same scenario sequence described in the UML activity diagram.

```
construct(T_service, T_collection, T_partition){
  v = first(T_partition);
  t1 = T_service ; t2 = T_partition ; t3 = T_collection ;
  T_service = traverse(v, t1, t2, t3)
}
traverse(v, t1, t2, t3 ) {
  if ( descendants_t2(v) = {0}) return; //this vertex has no outgoing transition
  if(v ∉ t2) return; //this vertex is not in this partition
  if (lab_t2(v) = s| s ∈ S ) {
    g = lab_t2(v);
```

```
    gapifyAChild(t1, u|lab_t1(u) = 'IM:Service', g);
    t = select(w| w ∈ t3, λ_t3^name(w) = λ_t2^name(v));
    tplug(t1, g, t);
  }
  ∀v_i| v_i= descendants_t2(v) ^ lab_t2(v_i) = 'IM:To') { // outgoing transition
    if(v|λ_t3^name(v) = λ_t2^target(v_i) ^ v ∉ t2) // this transition crosses boundary
        g = lab_t2(v_i);
        gapifyAChild(t1, v|lab_t1(v) = 'IM:Service', g);
        t = select(w| w ∈ t3, λ_t3^name(w) = λ_t2^name(v));
        tplug(t1, g, t);
        }
    else{ //a local transition
        v_t =( v|λ_t3^name(v) = λ_t2^target(v_i) ^ v ∈ t2);
        traverse(v_t, t1, t2, t3);
        }
  }
  if(∃v| v∈ t2 ^ v∉ t1) traverse(v, t1, t2, t3 );
}
```

This is a recursive algorithm, which takes a service (or working) template ($T_{service}$), collection template ($T_{collection}$) and its corresponding partition template ($T_{partition}$) and returns a completed service template. S is a name set containing strings of "IM:Step", "IM:Fork", "IM:Join" and "IM:TransitionArc" etc. The collection template is a global template that unions step, join|fork and transition templates.

Figure 4.18 explains how the algorithm works in a high-level overview. Starting from the first node in a partition (group), the algorithm finds the next node through the "IM:To" node or an outgoing transition (that crosses partitions). This step is repeated until one of two things happen. (1) There is no outgoing transition (the end node) or (2) the transition is a call to another partition where target node is not in the same group. Figure 4.18 only shows the Database (Z) partition as an example. In step (4), node "f4" has an outgoing transition "IM:TransitionArc" named "Data" that goes to "j2". This reply arc crosses to another partition; therefore, the outgoing arc is output and the search returns to next "IM:To" node.

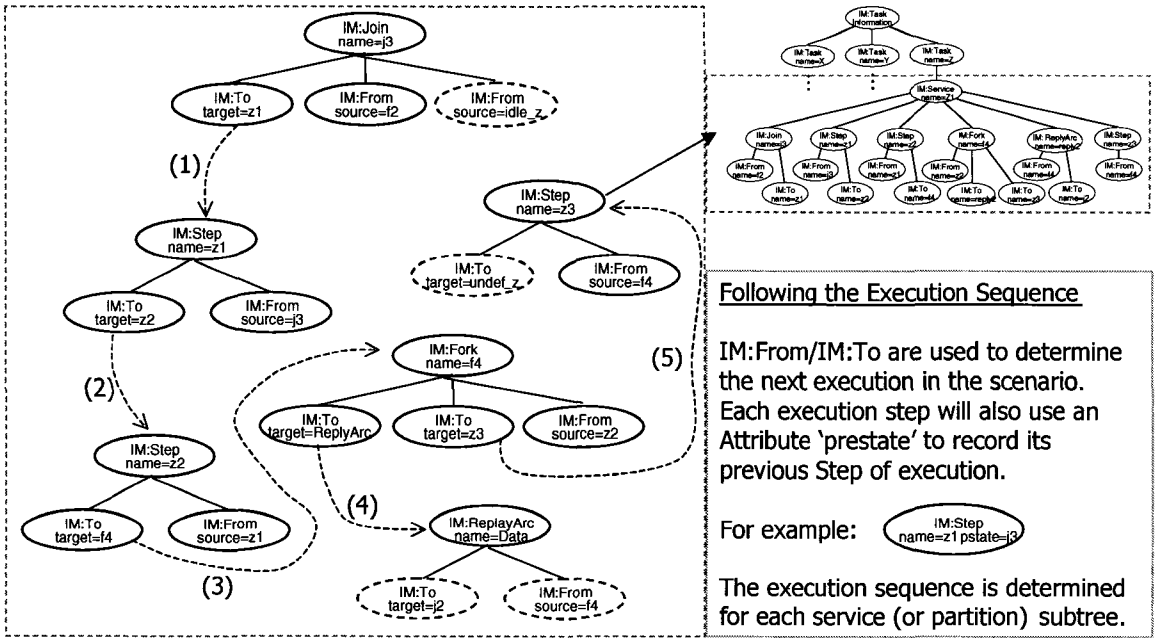


Figure 4.18: The execution sequence of Database (Z) partition.

4.5.4 Intermediate Model Output Algorithm

From the extraction and transfer operations, we obtained a template set $\{T\} = \{T_1, T_2, T_3, \ldots, T_i\}$ where T_i is a collection set that contains the required fragments for constructing the intermediate file template. Also, we define T_{dtd} for the intermediate model template DTD. For $\forall v \in \{T\}$, v must be defined in T_{dtd}. The construction of the intermediate model then becomes a function of $\{T\} \times T_{dtd} \rightarrow T_{im}$. This means that we use the template fragments in {T} to construct the intermediate model template in accordance with a given DTD template. The operation algorithm is given below:

$T_{im} = \{\varepsilon\}$;
$\forall v \mid v \in T_{dtd}$
$f(v)$;
where
$f(v)\{$
$\{S_t\} = \{w \mid w \in \{T\}, lab_w(1) = lab_{dtd}(v)\}$;
if $\{S_t\} = \{0\}$ *return;*
else $\forall S_i \in \{S_t\}$
$p_i = findparent(\{T\}, S_i)$;
if $p_i \in T_{im}$ {
if $S_i \in T_{im}$
return;
else {
$g = \lambda^{name}(S_i)$;

```
        gapifyAChild(T_im, p_i, g);
        tplug(T_im, g, S_i);
      }
    }
    else //p_i ∉ T_im;
      f(p|lab_dtd(p) = lab_t{p_i},p ∈ T_dtd ^ p_i ∈ {T});
  }
```

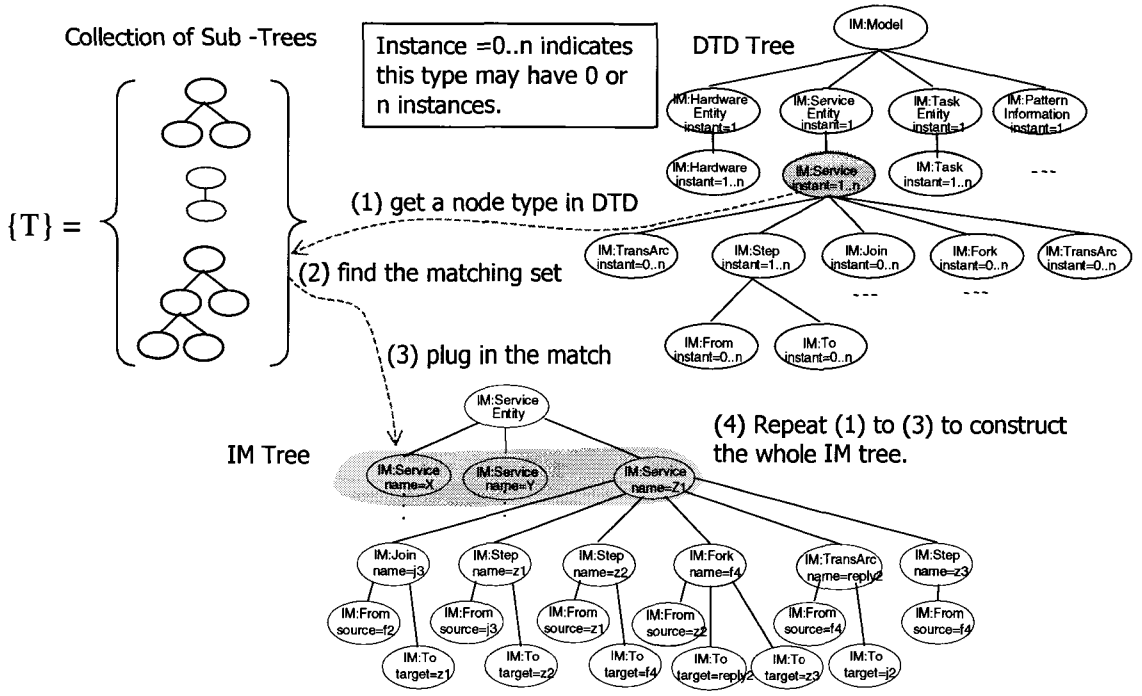


Figure 4.19: Illustration of the intermediate model construction algorithm.

This algorithm provides a generic method to construct a template from a given set of template fragments and a DTD template. All the smaller templates in {T} must comply with the given DTD. The fragments in {T} can either be vertices or small trees. The *f(v)* is a recursive function. It takes a vertex from the T_{dtd} and matches the first node of template fragments in {T}. If matches are found, it takes the parent vertex of the match vertex to check if the parent is already in the new template T_{im}. If the parent vertex exists and the match child does not exist, a gap is created and the matched vertex is plugged into the new template T_{im}; otherwise the algorithm returns. If the parent vertex is not found in the new template, then use the parent vertex as the matching vertex and repeats the steps. Moreover, The *findparent({T}, v)* function returns a vertex whose name

attribute is the same as the attribute "pname" of v, e.g., $\lambda^{name}(w) = \lambda^{pname}(v)$. A complete intermediate model for the 3-tier client server example is given as an example in Chapter 6.

Figure 4.19 illustrates the intermediate model construction algorithm. In short, the algorithm gets a node type from the DTD tree and finds the matches in the collection of sub-trees and plugs them to the IM tree. For example, the "IM:Service" is obtained from DTD tree and this node type has three matches in {T}, these three "IM:Service" sub-trees then are plugged into the output tree as steps (1) to (3) shown in Figure 4.19. If a node type has no match in the collection set or is already in the output tree, the algorithm goes for the next node type. Steps (1) to (3) are repeated until all node types in DTD are traversed.

4.6 GENERIC TRANSFORMATION ALGORITHM

This section introduces a generic algorithm that allows template conversion if a set of transformation rules is given. The purpose of this method is to separate the mapping rules from the transformation implementation details. This algorithm can convert a template (defined by an input DTD) to another template (defined by an output DTD, which can be identical or different from the input DTD). The conversion algorithm takes an input template "t_1" and the rule set "m" to construct an output template "t_2". The essence of this algorithm is to match each vertex in the input template with the mapping rule set (via the "select-apply" function), to create a gap in the output template and to plug in a tree (template) returned from the "select-apply" function. The operation calls itself for each child vertex recursively.

$T_{(\Sigma,A,G)} \times M \rightarrow T_{(\Sigma,A,G)}$ conversion algorithm:

```
conversion(t1,m){
  t2 = {ε};
  internalconversion(t1,m,t2);
 return t2;
}
internalconversion(t1,m,t2){
  v=first(t1)
  t' = select-apply(v);
  if(t'≠null)
     v'= gaplocation(v);
```

```
        g = lab(v);
        gapifyAChild(t2, v', g);
        tplug(t2, g, t');
       }
      if (subtree_t1(v)= {0} return;
      else{
       S = t_i | t_i ∈ subtree_t1(v);
       ∀t_i ∈ S
        internalconversion (t_i,m,t2);
       return;
       }
     }
```

The above conversion algorithm does not require any specific information of the target template. The '*select-apply*' function provides what to plug in by looking at the rule table and the '*gaplocation*' function returns the parent node of the currently transformed node, where the return of the '*select-apply*' function can be plugged in. In brief, the *select-apply(v)* takes input of a vertex and returns null, a vertex or a tree. A null return means that the input vertex has no match in the mapping rules. If it is a one-to-one mapping, a vertex is returned. If a tree is returned, it indicates a one-to-n mapping. A null or one-to-n mapping will result in template structure changes. On the other hand, the *gaplocation(v)* returns a vertex where a new vertex is to be plugged in. It should be noted that '*select-apply*" function is target template specific since each target template has its different transformation rules and plug-in rules. Other two functions used in the above algorithm are *first(t)* and $subtrees_t(v)$. The former returns the first vertex of a given template, i.e., $v = first(t) = v(1) \in t$. The latter returns all the child templates of v and the function definition is given below:

$$subtrees_t(v) = \{select(t, w_i) \mid w_i \in children(v))\} \text{ and}$$

$$children(v) = \{(c_1, c_2, \ldots, c_n)1, \ldots, (c_1, c_2, \ldots, c_n)k\}$$

The function returns a set of sub-templates of a given vertex in a template. Since each of the resulting templates is a sub-template that has domains dom_{ti} which contains the logical root ε, the i'th child vertex and the all descendant vertices of i in t. The labels and attributes remain unchanged.

4.7 TRANSFORMATION FROM INTERMEDIATE MODEL TO PERFORMANCE MODELS

In this section, we continue to describe how to transform an intermediate model to LQN and CSIM performance models, respectively. Before getting into the transformation details and rule definitions, it is necessary to understand the correspondences and relations between the intermediate model and the target performance model. Table 4.1 provides a mapping from the intermediate model components to both LQN and CSIM performance model components. It should be noted that the mapping from the intermediate model to the performance model is performance model specific. For instance, the task of the intermediate model is mapped to task and facility in LQN and CSIM, respectively. A request arc in the intermediate file is represented by a call with service demand in LQN and the number of calls is defaulted to one if not specified, while the same call is modeled as a message in CSIM. However, the mapping relations (shown in Table 4.1) do not reveal the architecture and behavior of a system design. The structure and scenario behavior of the design system must be reconstructed by traversing the scenario (service) trees of the intermediate model. Transformation rules are defined to assist the transformation of the intermediate model to performance models as discussed in the following sections.

Table 4.1: Mapping UML components to intermediate file, LQN and CSIM components

Intermediate File Components	LQN Component	CSIM Component
Hardware Resource	Hardware Device	Facility
Task	Task	Facility and Mailbox
Service	Entry	Process
Initial Step	Initial Step	Initial Step
Transition Arc	Calls (receiver, service provider)	Message
Step	Activity or Phase	Demand
Join and Fork	"AND" Join, Fork and Second Phase	"AND" Join and Fork
Merge and Branch, Loop	"OR" Join or Fork, Iteration	"OR" Join or Fork, Iteration

4.7.1 TRANSFORMATION OF INTERMEDIATE MODEL TO LQN MODEL

4.7.1.1 Difficulties in LQN Transformation

One of the difficulties in constructing a LQN model is to identify whether an entry has "phases" or "activities". This is an abstraction-raising problem, where a variable number of steps, depending on the concrete model, have to be aggregated together and mapped to one LQN element (i.e., phase or activity). This is done by traversing the intermediate model to determine if a fork or a branch operation exists (except for a reply). The steps are aggregated into a "phase" if there is no fork operation inside the partition, otherwise an "activity" is considered. The fork operation is further examined to determine if it is an inter-fork or intra-fork. The service demand is set to zero for the inter-fork. Moreover, a task entry is generated for each kind of service offered by the corresponding software component instance. The set of all services offered by an instance is determined by looking at all the messages received by this instance in all the scenarios considered for performance analysis. More details of the LQN elements are generated as follows:

a) Each task starts with one entry only. A new entry is added to the task if a new type of request is received. If a request is received more than once with the same message ID, its number of repetitions is increased by one.

b) All entries start in phase 1. When a server sends a reply back to the client or forwards it to another server, it moves to the second phase within the entry.

c) LQN activities are created if a conditional or non-conditional branching state is encountered. In the case of conditional branching, a LQN "OrFork" is created to connect alternate activities. An "OrJoin" is created to end the conditional branching, corresponding to a "merge" state in the activity diagram.

d) In the case of non-conditional branching, a LQN "AndFork" is created whenever a "Fork" state is used in the activity diagram to create a concurrent thread. However, the "Fork" state used by the servers for sending a reply at the end of phase 1 is an exception to this rule, and no explicit LQN "AndFork" is created in this case.

e) A LQN request arc is generated when a communication is detected between a phase or activity of a component playing the role of client, and the entry of another server, according to the corresponding high-level pattern. The visit ratio of the arc is given by the number of repetitions of the scenario step originating the request multiplied by

the number of requests made in that step. If more scenario steps contained in the same phase are sending a request to the same entry, we have to add the visit ratio contributions of all these scenario steps.

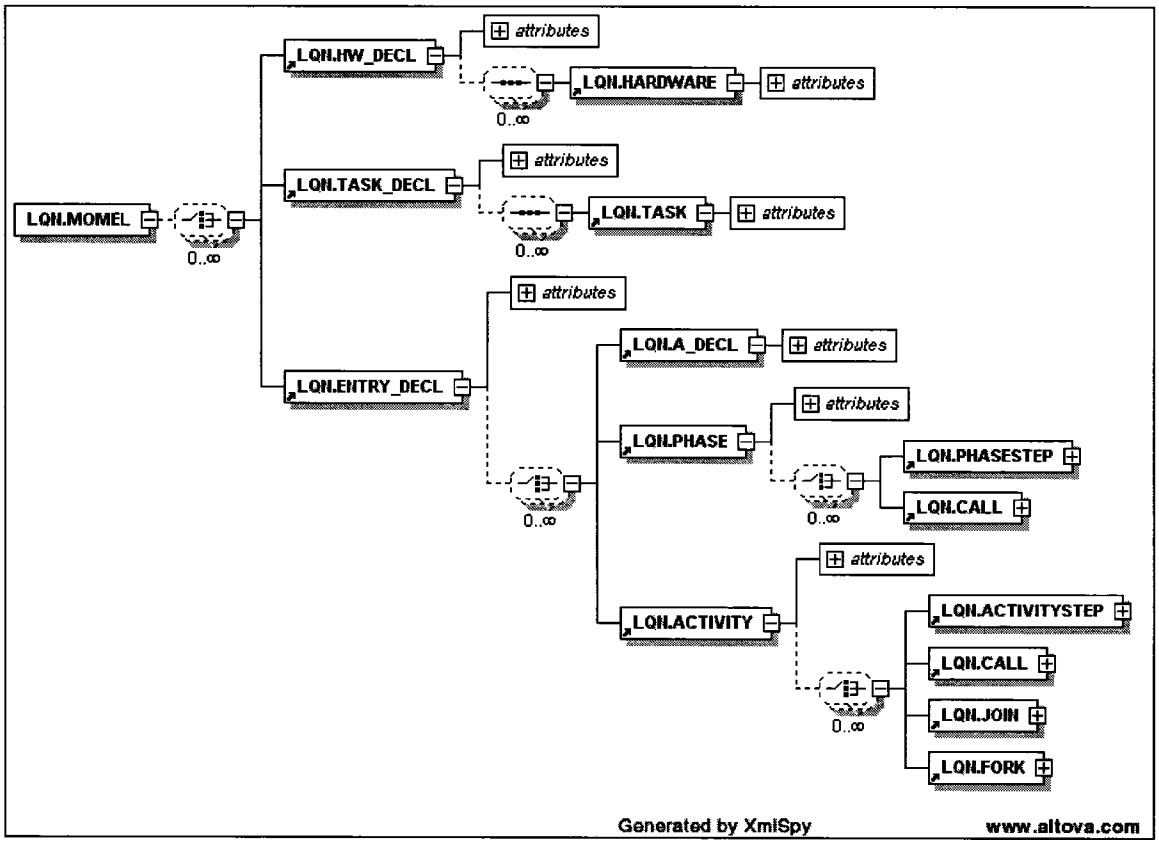


Figure 4.20:XML schema of LQN Model.

4.7.1.2 LQN DTD

The schema of the LQN DTD is given in Figure 4.20. It has three major components (hardware, task and entry) that are corresponding to hardware, task and service of the IM model (Figure 4.13). This LQN metamodel (which is derived from the DTD) is similar to others [122] in the conceptual level regardless to the differences. The LQN DTD template was defined in this work to contain essential performance information, to retain the system core scenarios of the original design, and to facilitate the final LQN text description file construction. It also adopted some of the definitions from LQN file grammar [39]. The LQN model contains Hardware, Task and Entry sections. The Entry has Activity or Phase sections to capture concurrent executions. "LQN:A_DECL" is an

activity declaration and its details are in "LQN:ACTIVITY", which consists of activity step (LQN:ACTIVITYSTEP), call (LQN:CALL), join (LQN:JOIN) and fork (LQN:FORK). A completed LQN DTD is given Appendix II.

T(IM)→T(LQN) Transformation Rules

{IM:Model(modelname)→LQN:MODEL(modelname,comment);
IM:HardwareEntity(name)→LQN:HW_DECL(name,number_of_processor);
IM:Hardware(name,taskname,comment)→LQN:HARDWARE(proc_id,comment);
IM:TaskEntity(name)→LQN:TASK_DECL(name,number_of_task);
IM:Task(name,hardwareref,comment)→LQN:TASK(task_id,proc_id,entry_id,comment);
IM:ServiceEntity(name)→LQN:ENTRY_DECL(name,number_of_entry);

IM:Service(name,hardwareref,think-time)
→{ LQN:PHASE(name,entry_id,think-time); ^ph_ser_time=Σservicetime.
  <LQN:A_DECL(task_id,entry_id,activity_id); LQN:ACTIVITY(task_id,entry_id,activity_id);>
IM:Step(name,taskname,servicetime,prestate,pname)
→{ LQN:PHASESTEP(entry_id,ph_ser_time);if phase.
  LQN:ACTIVITYSTEP(activity,serv_time);
IM:Fork(name,taskname,prestate,pname)
→{ LQN:Null(); if phase.
  LQN:FORK(name,fork_type);
IM:Join(name,taskname,prestate,pname)
→{ LQN:Null(); if phase.
  LQN:JOIN(name,join_type);
IM:Branch(name,taskname,prestate,pname)
→{ LQN:Null(); if phase.
  LQN:FORK(name,fork_type);
IM:Merge(name,taskname,prestate,pname)
→{ LQN:Null(); if phase.
  LQN:JOIN(name,join_type);
IM:TransitionArc(name,sourcetask,targettask,source,target,arctype,calltype)
→LQN:CALL(call_name,from_entry,to_entry,from_activity,to_activity,call_type,comment);if request
IM:TransitionArc(name,sourcetask,targettask,source,target,arctype,calltype)→LQN:Null();if reply.
IM:CompositeStep(name,taskname,serviceref,prestate,pname)--LQN:ACTIVITYSTEP(activity,serv_time);
IM:Iteration(name,taskname,begin,end,repetition)→Null();
IM:From(transname,fromStep)
→{LQN:From(transname,fromStep);
 LQN:Null(); if phase.
IM:To(transname,toStep)
→{ LQN:To(transname,toStep);
  LQN:Null();} if phase.}

4.7.1.3 IM → LQN Mapping Rules

There are fundamental procedures in the implementation to transform the intermediate model to a LQN performance model. The first procedure is to convert the intermediate model template to a LQN template. The naming and structure of the two XML templates are different; therefore, mapping rules between the two domains must be defined. These rules are explicit and static and they specify the corresponding conversion of the names and attributes between the two domains. The T(IM) → T(LQN) set of mapping rules is given above. It should be denoted that the above rules do not always represent a one-to-one mapping. In some cases, there is more than one choice that depends on the input scenario. For instance, the "IM:Service" can be mapped either to "LQN:PHASE" or "LQN:A_DECL + LQN:ACTIVITY". The choice is decided by the transformation function '*select-apply*', which takes a vertex from the input template (i.e., intermediate model) and returns a corresponding tree for the new template (i.e., LQN model). The above rules do not dictate the dynamic aspect of the transformation and provide no information on the template structure changes. These aspects are handled by the '*select-apply*' function, which contains specific information that is directly associated with this transformation.

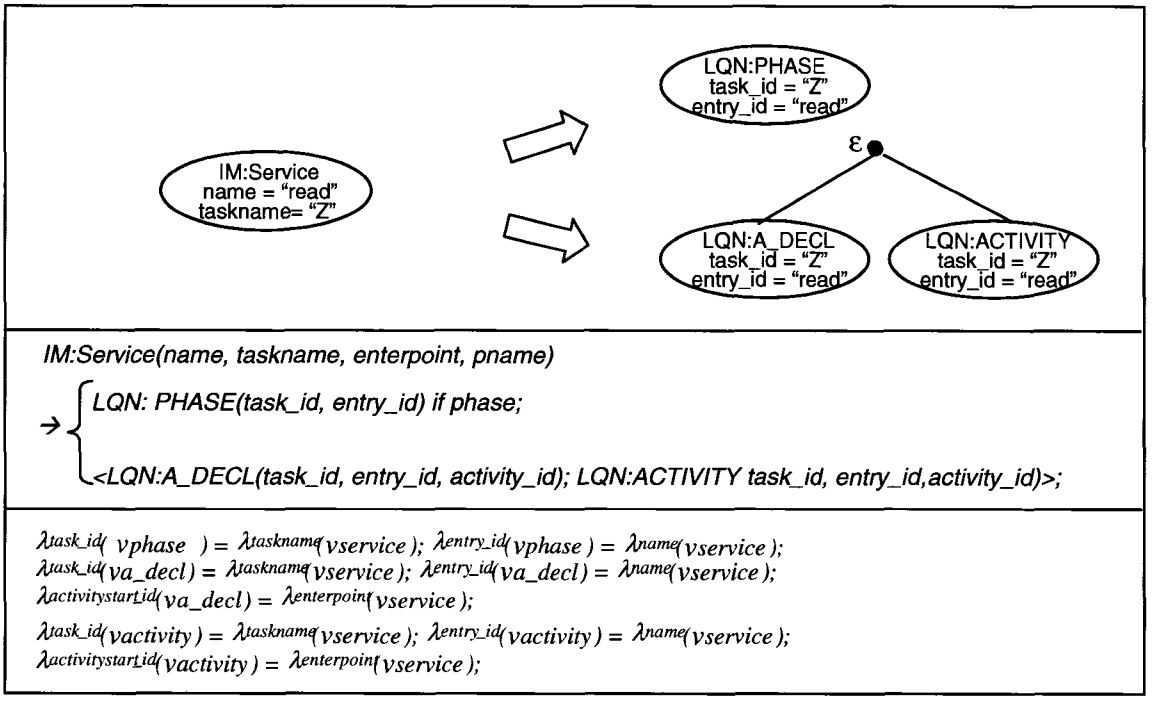


Figure 4.21: "IM:Service" → "LQN:PHASE" or "LQN:ACTIVITY" mapping.

One of the most challenging rules from IM to LQN is to deal with identifying if "IM:Step" is mapped to "LQN:PHASE" or "LQN:ACTIVITY", depending on rather complex conditions. This is an example of "abstraction-raising" transformation, where a group of elements from the source model have to be mapped to a single element in the target model. The number of model elements that have to be aggregated and their relationship vary widely from case to case. This problem was inherited from the UML model, but the abstraction raising is not solved in the UML-to-IM transformation. The solution is encapsulated in the rule for "IM:Service".

Figure 4.21 provides more details on attribute assignments of above transformation rule. This transformation has two alternatives. The choice is further delegated to the transformation function *select-apply*, which takes a vertex from the input template (in the IM domain), verifies the application conditions and returns the corresponding vertex or vertices, which is rooted by an empty node ε (that is not shown in Figure 4.21). The purpose of the *select-apply* function is even more far-reaching. Its goal is to encapsulate the verification of conditions for rule applications, which are specific to the performance target model, leaving the transformation algorithm that invokes *select-apply* and controls the application of the IM→LQN transformation rules as generic as possible.

Select-Apply function for LQN

The *select-apply(v)* function is specific to a target performance model. In this case, the select-apply function not only makes the choice of the transformation rule to be applied, but also performs the aggregation of steps (abstraction-raising) when necessary. The following is the definition of *select-apply(v)* function to convert $T_{IM} \rightarrow T_{lqn}$:

```
select-apply(v)
{
  if (v ∉ V) return null;
  if (lab(v) = 'IM:Service'){
    if (∃v_i | v_i ∈ descendants(v), u=lab(v_i)='IM:Fork'
          ^ ∃u_i | u_i ∈ descendants(u), λ^calltype(u_i) ≠ reply)
      return w=lookup(v(1));//it is an activity
    else return w=lookup(v(0));//it is a phase
    }
  else{
    if (p = parentService(v)==null)
```

```
        return w=lookup(v);
    else{
      if (lab(select-apply(p) = 'LQN:ACTIVITY)
        return w=lookup(v(1));
      if (lab(select-apply(p) = 'LQN:PH_DECL'){
        if(lab(v) = 'IM:Step' ^ ∃ v_c, | v_c =children(v)
          ^ lab(v_c) = 'IM:From' ^λ^name(v_c) = λ^pname(v))
          while(u = p.i ∈ Tim){
            if(lab(u) = 'IM:Step' ^ λ^pname(u)= λ^pname(v))
              st = st + λ^servicetime(u);
           i=i+1;
          }
          λ^servicetime(w)=st;
          if(λ^name(v_c) = 'Join') λ^name(w)=phase1;
          if(λ^name(v_c) = 'Fork') λ^name(w)=phase2;
          return w;
        }
       else
         return null;
       return w=lookup(v(0));
      }
    }
  }
}
```

The *select-apply* function takes an input node and returns one of the following: null, a node or a tree, in accordance with the defined IM → LQN rules. The choice of whether to convert an "IM:Service" template to phase or activity is made as follows.

The IM input model is traversed to determine whether a fork operation, which does not involve a reply, exists. A conversion to "phase" is applied if there is no such fork, otherwise a conversion to "activity" is considered. The fork operation is further examined to determine if it is an inter-fork (inside a partition) or an intra-fork (one that sends a message across partition). The service demand is set to zero for the inter-fork. Moreover, a task entry is generated for each kind of service offered by the corresponding software component instance. The service demands of the aggregated steps are added together to produce the service demand for the generated phases or activities. The above function can also distinguish between LQN activity and phase by checking if there is a fork operation that does not involves a reply. If the condition is true, then the corresponding entry is treated as having LQN activities; otherwise it is transformed to phases.

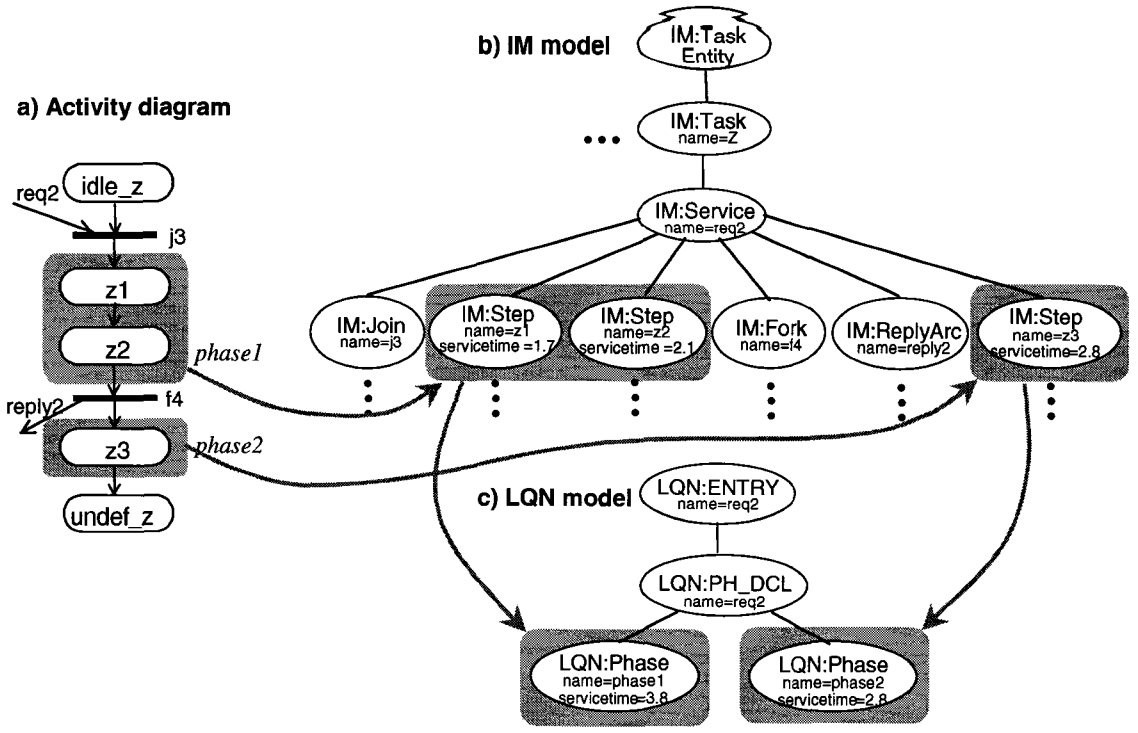


Figure 4.22: From UML to IM to LQN: aggregating scenario steps into LQN phases.

Figure 4.22 traces the problem of deciding on activities and phases from UML to IM, and then to LQN. It illustrates how a partition of the activity diagram is transformed to the corresponding IM, and then further to LQN. The steps z1 and z2 are grouped into phase1, while step z3 becomes phase2. Once an XML LQN model is obtained, it can be further formatted to a LQN text file using an XSLT stylesheet "lqn2text.xsl". The textual output then can be evaluated by LQN engine for performance [42].

4.7.2 IM to CSIM Transformation

4.7.2.1 High-Level UML to CSIM18 Transformation

Before getting in the details of how to transform IM to CSIM model, it is worthwhile to discuss two important aspects of the UML design to CSIM transformation. The first aspect is that the CSIM model structure including the software processes, logical and hardware resources, hardware devices and the connecting elements (mailbox, buffers etc.) are generated in accordance with the UML design, e.g. the software architecture and deployment diagrams. This is called structural mapping. The second aspect is that the

execution details, which correspond to service time executions, process interaction are derived from scenario descriptions. It is also known as behavior mapping.

The behavioral mapping of UML to CSIM must preserve the execution details described in the UML scenario. Each step that requires service time can be simulated with a "time holding" in CSIM to represent the service granted by the server. The execution time demands and visit ratio parameters of each step are obtained from annotated UML scenarios. Figure 4.23 gives a graphical presentation of the 3-tier client server CSIM model, in which the inter-nodes communication is represented by the CSIM mailbox messaging. A circle inside a rectangle represents a process and a circle itself represents a step. MBS and MBR stand for mailbox send and receive, respectively. For example, Px sends a message to Py and waits for receiving the replying message, while Py waits for message coming in. By comparing Figure 4.23 (a) and (b), CSIM simulation model retains most of execution details in the original design.

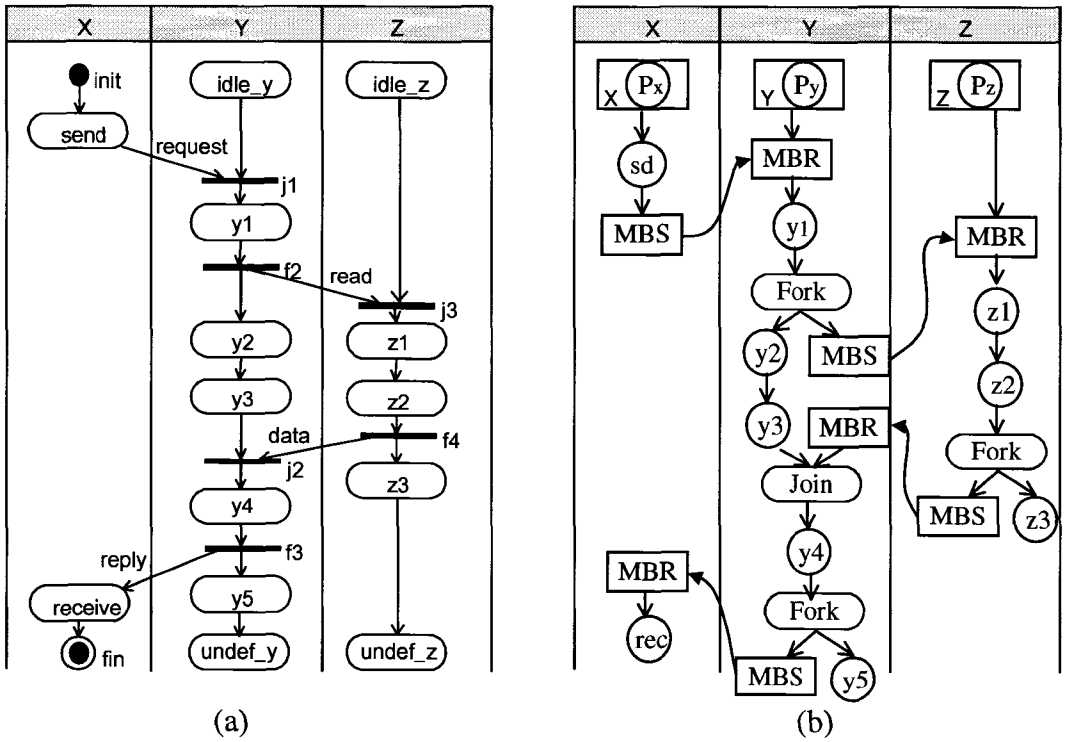


Figure 4.23: (a) A modified activity diagram, and (b) A CSIM model for 3-tier client server. Tasks X, Y and Z represent the client, application and database, respectively.

To transform UML design shown in Figure 4.23(a) to CSIM performance model (shown in Figure 4.23(b), our approach is UML design → IM then IM → CSIM. In the early

sections of this chapter, many discussions have been given to UML design → IM, which we will not repeat here. In the sections below, our focus rests on how to transform IM → CSIM. We will introduce the CSIM DTD first and then follow by the IM→CSIM transformation rules and algorithms. We will leave the output detail to Chapter 6.

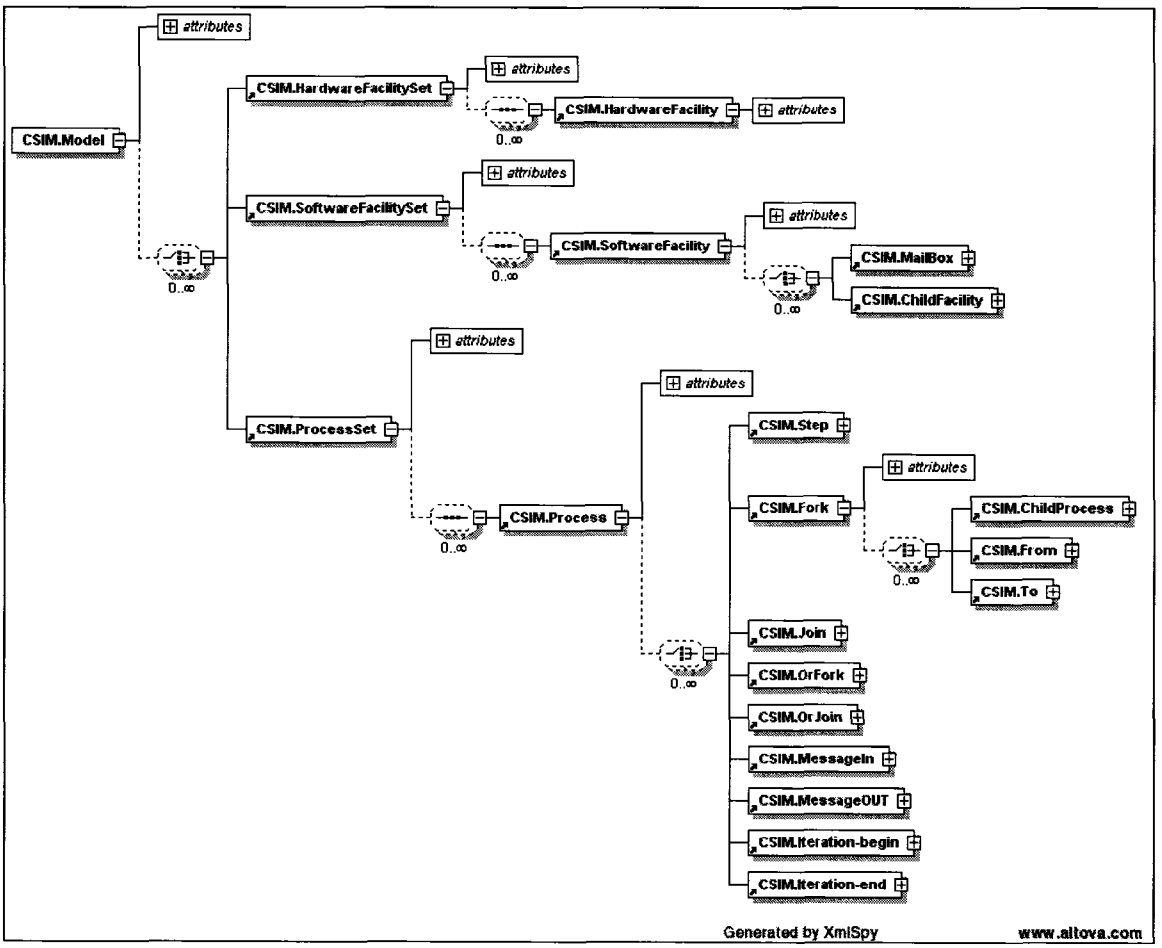


Figure 4.24: Schema for CSIM Model.

4.7.2.2 CSIM DTD

Similar to constructing a LQN template, the IM → CSIM model transformation relies on the transformation rules defined between IM and CSIM DTDs. The CSIM DTD template is designed to contain essential performance information as defined in the SPT performance profile to retain the system core scenarios of the original design. Figure 4.24 is a schema of CSIM DTD, which contains "CSIM:HardwareFacility", "CSIM:Software Facility", and "CSIM:Process" etc. The "CSIM:Process" is associated with a

"CSIM:MailBox" that is used to simulate inter-process messaging, and it contains detail execution scenario of a process. A "CSIM:Process" has "CSIM:Step", "CSIM:Join", "CSIM:Fork" and other operations.

4.7.2.3 IM to CSIM Conversion Rules

Mapping the intermediate template to CSIM template is done by following the T(IM) → T(CSIM) mapping rules. The "IM:Task" is mapped to "CSIM:SoftwareFacility", which is always associated with a "CSIM:MailBox" to simulate its queue. The fork operation may lead to a new child process creation if its operation is within the parent process domain. In such as case, a new "CSIM:ChildProcess" must be generated during the IM→CSIM transformation. If the fork operation is associated with message out operation to another process (that has been created), then there is no new process being created.

T(IM)→T(CSIM) Transformation Rules

{IM:Model(modelname,comment)→CSIM:Model(modelname,comment);
IM:HardwareEntity(name)→CSIM:HardwareFacilitySet(name);
IM:Hardware(name,taskname,comment)→CSIM:HardwareFacility(name,comment);
IM:TaskEntity(name)→CSIM:SoftwareFacilitySet(name);
IM:Task(name,hardwareref,comment)→
CSIM:SoftwareFacility(name,hardwareref,comment);
<CSIM:MailBox(name,softwareref);CSIM:ChildFacility(name,hardwareref,comment);>
IM:ServiceEntity(name)→CSIM:ProcessSet(name);
IM:Service(name,hardwareref,think-time)→CSIM:Process(name,hardwareref,softwareref);
IM:Step(name,taskname,servicetime,prestate,pname)→CSIM:Step(name,servicetime);
IM:Fork(name,taskname,prestate,pname)→CSIM:Fork(name,condition);<CSIM:ChildProcess(startStep,endStep);>
IM:Join(name,taskname,prestate,pname)→CSIM:Join(name,condition);
IM:TransitionArc(name,sourcetask,targettask,source,target,arctype,calltype)→
CSIM:MessageOut(name,fromProcess,toProcess); if request.
IM:TransitionArc(name,sourcetask,targettask,source,target,arctype,calltype)→
CSIM:MessageIn(name,fromProcess,toProcess); if reply.
IM:Branch(name,taskname,prestate,pname)→CSIM:OrFork(name,condition);
IM:Merge(name,taskname,prestate,pname)→CSIM:OrJoin(name,condition);
IM:Iteration(name,taskname,begin,end,repeatition,prestate,pname)→
<CSIM:Iteration-begin(startStep,repeatition);CSIM:Iteration-end(endStep);>
IM:From(transname,fromStep)→CSIM:From(transname,source);
IM:To(transname,toStep)→CSIM:To(transname,target);}

The above IM→CSIM rules are relatively straightforward and they are mostly one-to-one mappings, except for three special cases (in fact, two are very similar cases). One of them is:

IM:Task(name,hardwareref,comment)→
CSIM:SoftwareFacility(name,hardwareref,comment);
<CSIM:MailBox(name,softwareref);CSIM:ChildFacility(name,hardwareref,comment);>

Figure 4.25 provides the transformation rules for "IM:Task" → "CSIM:SoftwareFacility" and "CSIM:MailBox" if there is no intra-fork operation inside the task. The "IM:Task" is transformed to a tree containing a "CSIM:SoftwareFacility" node and a child node "CSIM:MailBox". If there is an intra-fork operation, then an additional child node "CSIM:ChildFacility" is generated to represent a child process. However, the child process would share the same mailbox with its parent process.

Using the transformation algorithm described in section 4.6 and the mapping rules shown above, we can construct a CSIM performance model template. The transformation function *select-apply* for IM→CSIM, which takes a vertex from the input template (e.g., intermediate file) and returns a null, vertex or a tree in new template (e.g., CSIM model), is defined below.

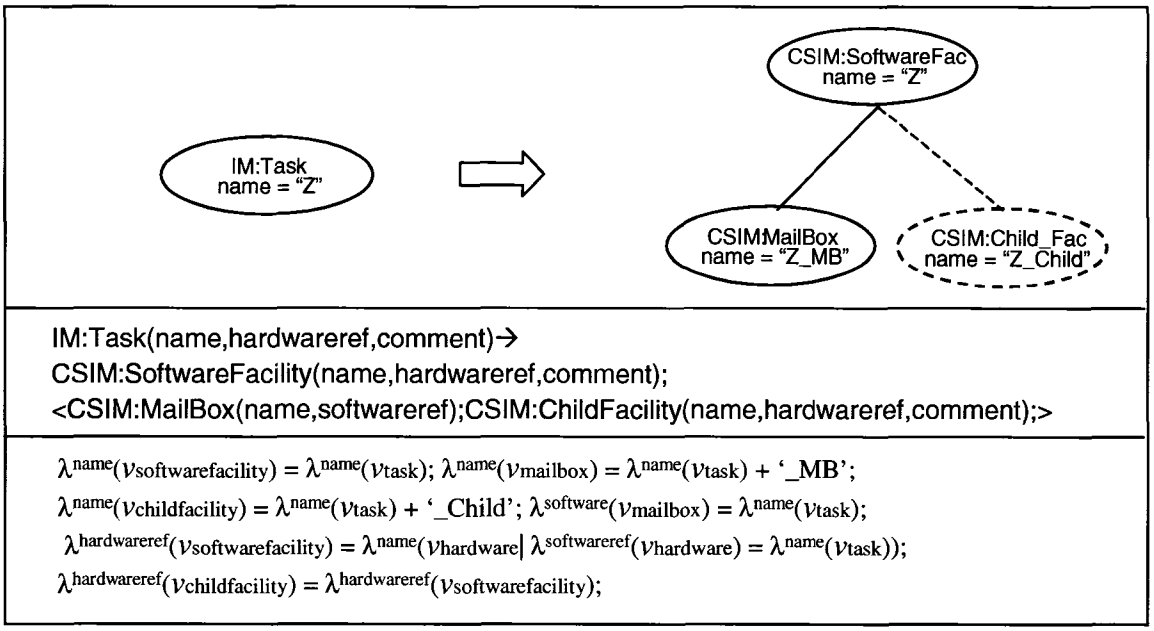


Figure 4.25: "IM:Task" → "CSIM:Process" and "CSIM:MailBox" .

select-apply(v)
{ n=0;
if (v ∉ V) return null;
if (lab(v) = 'IM:ServiceEntity'){
$S_w = (w_1 , w_2 , \ldots w_i) | w_i \in descendants(w | lab(w) = 'IM:Service' \wedge \lambda^{taskname}(w) = \lambda^{name}(v)),$
$\wedge lab(w_i) = 'IM:Fork' \wedge \exists u_i | u_i \in descendants(w_i), lab(u_i) \neq 'IM:TransitionArc';$
$n = count(S_w);$

```
  }
  if (lab(v) = 'IM:Task'){
   return lookup(v(n));//will create n child processes
  }
  else if(lab(v)='IM:Fork' ^ ∃u_i | u_i ∈ descendants(w_i), lab(u_i)≠ 'IM:TransitionArc'){
   return lookup(v(n)); //will have a child function
  }
 else
  return lookup(v(0));//the rest is one-to-one mapping
}
```

The above *select-apply* function is rather simple. It consists of two special cases that are one-to-n mappings, the "IM:Task" and "IM:Service". In both cases, we need to decide if there is a child process to be created. In "IM:Service"→"CSIM:Process", if a fork operation within a process (intra-fork) involves concurrent execution (e.g., excluding any reply or request operations), the *select-apply* returns a tree with *n* "CSIM:ChildProcess" where *n* is the number of forks that satisfy the condition. The "CSIM:ChildProcess" contains attributes of starting and ending steps. When transforming the "IM:Task", a similar check is also applied and the *select-apply* returns a tree with *n* "CSIM:ChildFacility" nodes. The *n* must be the same in both cases. Once the CSIM template is available, it can be converted to a compile-ready C/C++ code that can be run in CSIM18 engine using XSLT stylesheet [42]. The output of this 3-tier client server model transformation is given in Appendix IV.

Code implementation of the transformation rules including intermediate model to both LQN and CSIM models, can be added, modified or removed to facilitate performance model generation. The template conversion algorithm, however, is generic. To transfer an intermediate file template to a performance model, the *select-apply* function and mapping rules are specific.

4.8 PATTERN TRANSFORMATION

Design pattern plays important role in today's software industrial practice as it allows reuse of code and standardizes design architectures. Many well-tested patterns can be built into modules and used as building blocks in large software system development. Therefore, it is necessary to preserve the use of patterns in UML design while building

the performance models. Regardless of the fact that this is not in the main scope of the thesis, in this section we address briefly how to retain pattern information extracted from the UML design into the intermediate model, which can be used for building the performance models. Our goal is to discuss a potential approach that could be developed in future work.

There are two kinds of patterns of interests in this work: architectural patterns and use of resources. The software architecture view may be described in the form of architectural patterns [40] which identify frequently used architectural solutions, such as client/server, pipeline and filters, client/broker/server, master-slave and blackboard etc. Use of resources, particular passive resource such as storage buffer, is characterized by its acquire/release behavior, which guarantees mutual exclusion. An application must acquire a resource before use and must release the resource after use. The behavior of a critical section can also be described in a similar way, as a passive resource that must be acquired by a process before entering, and released after exiting the critical section.

Design patterns are frequently embedded in the software design. Identifying and recording patterns in the transformation is important as the pattern facilitates the performance model building. To store the pattern information, we need to define a pattern element in the IM DTD. Such a definition must be specific enough to contain necessary information, yet it must be generic in order to represent a variety of patterns. The following is the proposed DTD element for recording pattern information in IM.

Pattern Definition in IM DTD:

```
<!ELEMENT IM:Pattern (IM:PatternAssociate|IM:PatternInteraction)*>
<!ATTLIST IM:Pattern name                CDATA #REQUIRED>
<!ELEMENT IM:PatternAssociate EMPTY>
<!ATTLIST IM:PatternAssociate name       CDATA #REQUIRED>
<!ELEMENT IM:PatternInteraction EMPTY>
<!ATTLIST IM:PatternInteraction name     CDATA #REQUIRED
                        interactiontype  CDATA #REQUIRED
                        actionsource     CDATA #REQUIRED
                        actiontarget     CDATA #REQUIRED>
```

The "IM:Pattern" node has an attribute "name" and it could have a number of children nodes "IM:PatternAssociate" and "IM:PatternInteraction". The "IM:PatternAssociate" is

the task involved in the pattern and "IM:PatternInteraction" is the interaction between the tasks described in the pattern.

4.8.1 Client-Server Pattern

The first and simplest pattern addressed here is the client server pattern. Both LQN and CSIM can represent very well this pattern. Figure 4.26(a) illustrates a case where a client communicates directly with a server through synchronous requests. Both client and server tasks are denoted as <<PAresource>> as defined in performance profile [78] and a synchronized call is explicitly indicated. Normally, a server may offer a wide range of services with different performance attributes. From a performance modeling point of view it is important not only to identify these services, but also to find out by whom and how frequently the server is invoked. In this example, the client server pattern is explicitly expressed in the architecture diagram. It is also assumed that the server provides one service call and the call type is given in the stereotype.

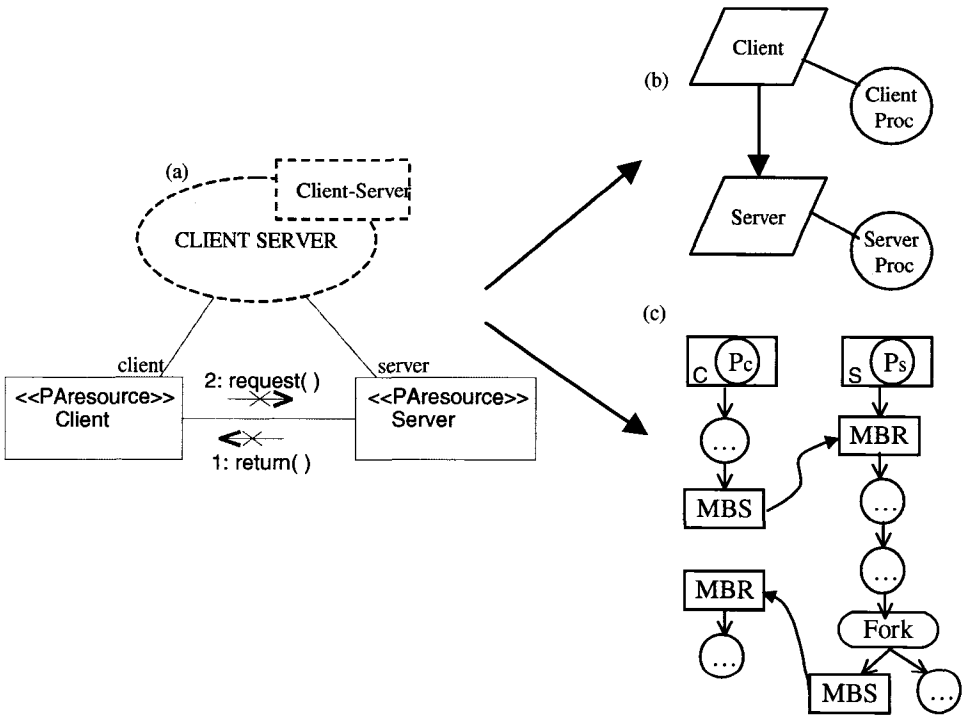


Figure 4.26: Client/Server architecture Pattern Transformation. (a) UML diagram; (b) LQN model and (c) CSIM model.

The client server pattern information output to the intermediate file follows:

```
<IM:Pattern name = "client-server">
 <IM: PatternAssociate name="Client" />
 <IM: PatternAssociate name="Server" />
 <IM: PatternInteraction  name="request" interactiontype = "synchronous"
                          actionsource = "Client"  actiontarget = "Server" />
</IM:Pattern>
```

As LQN was originally created to model client/server systems, the transformation from the client/server pattern to LQN is quite straightforward. Two tasks with corresponding hardware were modeled (Figure 4.26(b)). An arrow from client to server represents the message sent. In LQN, 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. A LQN server offers a range of services, each with its own CPU time and number of visits. A service is modeled as a service of the server task, and will contribute to the response time, utilization, and throughput of the server. A client may invoke more than one of these services at different times. The performance attributes for the clients include their average CPU time demands, and the average number of calls for each service of the server. Figure 4.26(c) shows the corresponding CSIM simulation model. Each task is modeled by a CSIM process, which uses software and hardware facilities and a mailbox. The client process sends a message to server's mailbox. The server processes the message and puts back the reply to client's mailbox. Message will be queued in the mailbox if the server is busy. The report function provided by CSIM18 records the performance parameters for analysis.

4.8.2 Pipeline and Filters Pattern

The pipeline and filters pattern can also be expressed as a UML collaboration (Figure 4.27(a)). The call type is asynchronous, which means that no reply is required. The pipeline pattern information outputted to the intermediate file is:

```
<IM:Pattern name = "pipeline">
 <IM:PatternAssociate name="filter1" />
 <IM:PatternAssociate name="filter2" />
 <IM:PatternInteraction  name="message" interactiontype = "asynchronous"
                          actionsource = "filter1"  actiontarget = "filter2" />
</IM:Pattern>
```

Figures 4.27(b) and 4.27(c) shows the corresponding transformation models in LQN and CSIM simulation, respectively. The transformation takes into account the structural and behavioral information provided by the UML architecture. To transfer the pipeline and filters pattern to LQN, each active filter becomes a LQN software server with a single service whose service time includes the processing time of the filter. A filter task receives an asynchronous message and executes its phases in response to it. Typically, the distribution of the work includes to receive and process the message, and to send it to the next filter. Messages in pipeline and filters pattern are modeled by CSIM process with facilities and mailbox.

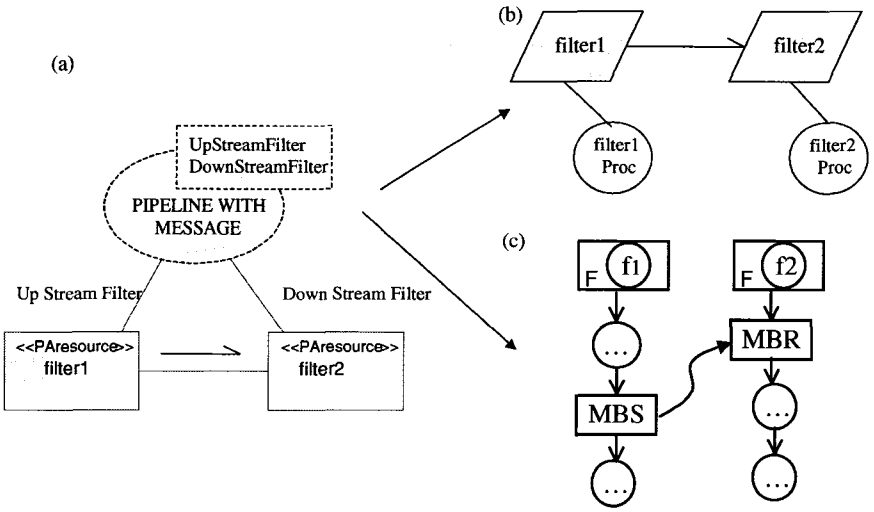


Figure 4.27: Pipeline and Filter Pattern Transformation. (a) UML diagram; (b) LQN model and (c) CSIM model.

Figure 4.27(c) shows the two filters in different nodes. If the two filters reside in the same node, they will share the same hardware facility. The message is forwarded from one filter to the other via CSIM mailbox.

4.8.3 Use of Resource Pattern

The use of resource pattern describes a behavior rather than architecture. It is easier to present the pattern in a UML activity diagram. Figure 4.28(a) depicts a scenario of an application use of a storage buffer resource. The application makes a request to acquire the buffer before use and release it after use. If there is no buffer available, the

application either waits or makes another attempt later. To extract this pattern information is difficult, as the pattern is implied and embedded in the behavior. We need to traverse the scenario to seek behavior pattern that matches the known pattern. In this case, buffer is stereotyped as <<PAresource>>. A step that allocates a buffer is also stereotyped as <<GRMacquire>>, indicating a resource acquisition; and a step that releases a buffer is stereotype as <<GRMrelease>>.

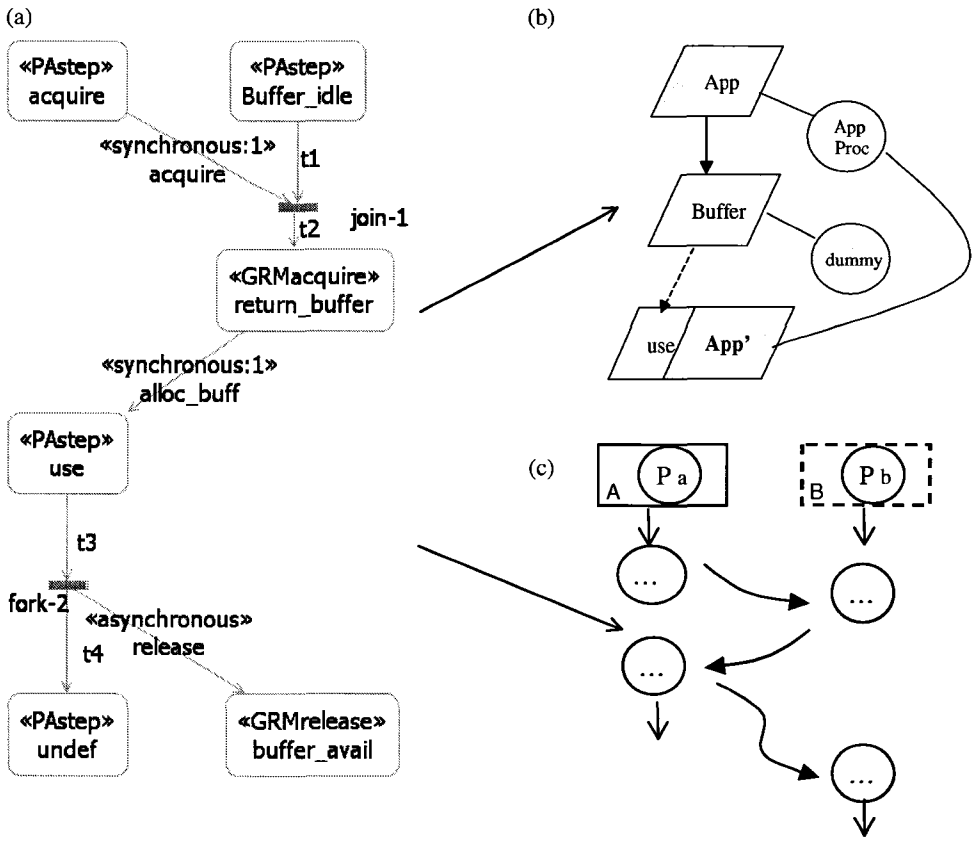


Figure 4.28: Buffer Pattern Transformation. (a) UML activity diagram; (b) LQN model and (c) CSIM model.

The use of resource pattern information outputted to the intermediate file is:

```
<IM:Pattern name = "use of resource">
 <IM:PatternAssociate name="App" />
 <IM:PatternAssociate name="Buffer" />
 <IM:PatternInteraction name="acquire" interactiontype = "asynchronous"
                        actionsource = "App" actiontarget = "Buffer" />
 <IM:PatternInteraction name="release" interactiontype = "asynchronous"
                        actionsource = "Buffer" actiontarget = "App" />
</IM:Pattern>
```

To transform the use of the resource pattern to the LQN model, the buffer is represented by an active task that does not use CPU and receives a forwarding call from the application. Figure 4.28(b) shows the corresponding LQN model where task App and App' represent the same task in the activity diagram. On the other hand, the CSIM simulation modeling is relatively simple in this case as shown in Figure 4.28(c). CSIM process can directly acquire the buffer if available otherwise, the request will be queued. After use, the buffer must be released explicitly.

The above discussion mentions a few examples of software pattern transformations in building the LQN and CSIM simulation performance models. However, a software design system normally is far more complicated and patterns may not be obvious as they are embedded in the design. In transforming a UML design to performance model, the pattern information alone is not enough in constructing the performance model as many of the execution details are embedded in the behavior (in activity diagram). Therefore, the pattern information must be used along with the scenario behavior information. For example, the client server pattern information does not indicate in which step the synchronous call is made, such detail must be obtained from the execution behavior indicated in the design. However, templates for pattern transformations do facilitate the UML design to performance model transformation.

4.9. SUMMARY

Starting with high-level concept of transformation, step by step in this chapter, we explained the essence of transforming UML designs to performance models (LQN and CSIM). Aspects that have been addressed include (1) UML design with performance annotations; (2) extended XMLgebra; (3) Performance information extraction; (4) Intermediate model (IM) schema and construction; (5) transformation rules and generic transformation algorithms; (6) IM to LQN; and (7) IM to CSIM transformations. An example of 3-tier client-server model was used to illustrate the transformation between UML design to intermediate model, and intermediate model to performance models. In the last section of the chapter, design patterns transformation, including simple client/server, pipeline and filter, and storage buffer resource patterns, were also

discussed. In next chapter, the XSLT code implementation of the transformation steps defined here, are presented.

CHAPTER 5: XSLT IMPLEMENTATION

5.1 INTRODUCTION

The transformations described in Chapter 4 (i.e., UML-to-IM, IM-to-LQN and IM-to-CSIM) are implemented using the XSLT language. As briefly introduced in section 3.2.5, XSLT was designed as a part of XSL, a stylesheet language for XML, meant to describe how to display an XML document of a given type. Originally intended to perform styling operations, such as the generation of tables of contents and indexes, XSLT has evolved beyond its original scope and is now used as a general purpose XML processing language [59]. XSLT describes rules for transforming a source document in a tree format (such as an XML file) into a result document described also by a tree. A transformation expressed in XSLT is named a stylesheet and it contains a set of template rules. It should be noted that the "template" terminology in XSLT is similar to a function in other computer languages such as C/C++ or Java, which is different from the concept "template" described in XMLgebra earlier. An XSL template contains instructions for creating result tree fragments. When a template is instantiated for a particular source element, each instruction is executed and replaced by the result tree fragment that it creates. Instructions can select and process descendant source elements, as well. The main strength of XSLT is the matching of given features in the source tree through patterns and the generation of the result subtrees through templates.

In the previous chapter, we have defined the transformation using eXMLgebra. Their implementation in XSLT is described in this chapter. As indicated in Figure 5.1, the implementation contains four packages, including domain definition, transformation rules, transformation algorithms, and supporting templates. The transformation algorithm package is the core, which is supported by the others. In the following sections, we introduce each component one by one.

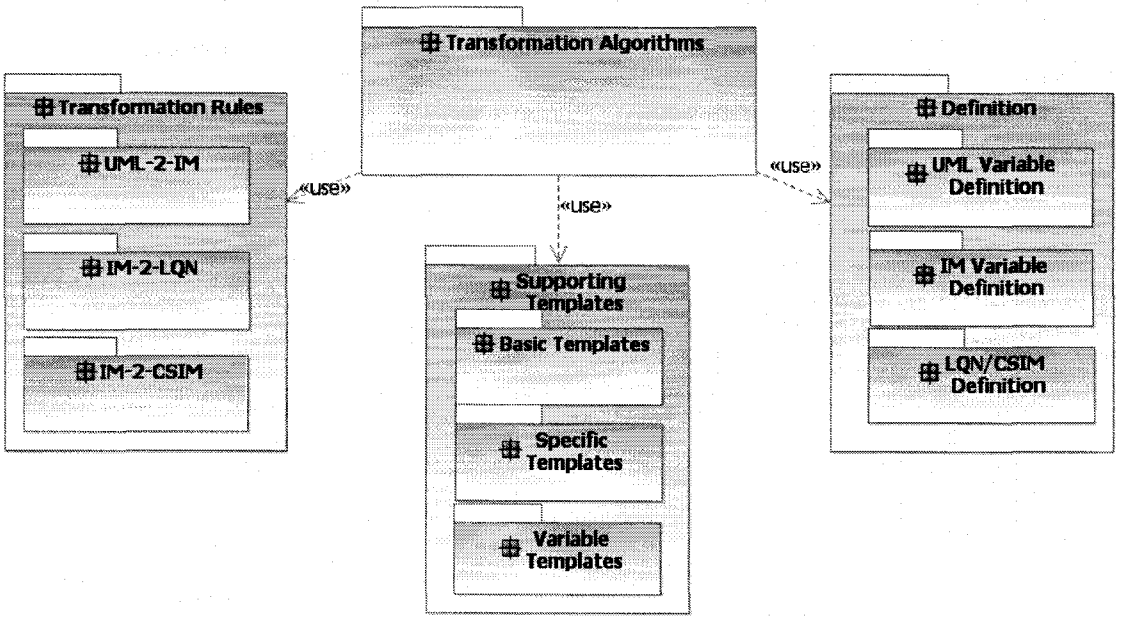


Figure 5.1: XSLT implementation modules.

5.2 DOMAIN DEFINITION

This package contains three modules of definitions for source, target and intermediate models, respectively. In our approach, we have UML, IM and LQN/CSIM variable definitions. An example of the UML2.0 variable definition is given in Figure 5.2. The variable definition of each domain has two parts, the node type and attribute. In Figure 5.2, node type and attribute names are directly taken from the UML 2.0 model exported using XMI. In XSLT, these variables are defined using <xsl:param name = "name" select = "value" />, The "name" is the variable name and "value" is its value. For example, we define: name="uml-nodetype-3" and select = "node", which means that the variable *$uml-nodetype-3 = node.*

There are two reasons to separate the variable definitions as a module. One is to make the transformation module as generic as possible and the other is to avoid hard-coding and to facilitate the adaptation to changes. This approach allows for an easy adaptation to UML2.0 from UML1.4, simply by replacing the UML variable definition module. The IM, LQN and CSIM variable definitions are also in a very similar format.

```
<!-- %% UML2.0 node type definition %% -->
<xsl:param name="uml-nodetype-3" select="'node'"/>
<xsl:param name="uml-nodetype-4" select="'edge'"/>
<xsl:param name="uml-nodetype-5" select="'group'"/>

<!-- %% UML2.0 attribute type definition %% -->
<xsl:param name="uml-attribute-1" select="'uml:Action'"/>
<xsl:param name="uml-attribute-2" select="'uml:Node'"/>
<xsl:param name="uml-attribute-3" select="'uml:Artifact'"/>
```

Figure 5.2: UML 2.0 node type and attribute definition example.

5.3 TRANSFORMATION RULES

For similar reasons as indicated earlier, the transformation rule package is also separated from others in order to allow user to easily modify, remove and add rules. In Chapter 4, the UML2.0 to IM, IM to LQM and IM to CSIM transformation rules have been defined. In the code implementation, transformation rules are defined as XSLT variables in the following format:

<xsl:variable name = "rule name">
rule expression (LHS – RHS);
rule expression (LHS – RHS);
.....
</xsl:variable>

The format of each rule contains the left and right hand sides and they are separated by the symbol "--". The left hand side (LHS) is the source node type and the right hand side (RHS) is the target node type. Both source and target node types can contain a number of attributes indicated inside the bracket. Most of the transformation rules are one-to-one mappings, as indicated below:

node-uml:Action(xmi:type,xmi:id, name,outgoing,incoming,inPartition)	--	*IM:Step(name,taskname,servicetime, prestate,pname);*

The above rule indicates that the UML "node(xmi:type = "uml:Action")" node is transformed to "IM:Step" node. If it is a one-to-zero transformation, a Null() is indicated in the RHS:

EAnnotations(xmi:id,source)	--	*IM:Null();*

Another type of transformation is called alternative transformation. It has a choice of target nodes depending on a selection condition, but it is still a one-to-one mapping when the condition is met. For example, the *"IM:Service"* node can be transformed to

"LQN:PHASE" or *"LQN:A_DECL" and "LQN:ACTIVITY"* depending on the execution scenario. *"IM:Service"* → *"LQN:PHASE"* is an alternative one-to-one transformation. To identify a choice, a condition identifier is attached at the LHS of the rule, such as "*-phase"* or "*-activity"* in the XSLT implementation.

One-to-n transformation means that more than one target nodes are in the RHS An example is *"IM:Service-activity"* to *"LQN:A_DECL"*and *"LQN:ACTIVITY"* transformation as shown below:

IM:Service-activity(name, hardwareref,think-time)	--	*LQN:A_DECL(task_id,entry_id, activity_id)+LQN:ACTIVITY(task_id,entry_ id, activity_id);*

In the case of n-to-one transformation, a condition argument is also given in LHS In transforming *"IM:Step"* to *"LQN:PHASESTEP"*. A condition check is performed to determine if the value of attribute 'prestate' remains the same (prestate is a glueing attribute that records previous fork operation). If yes, this node will be merged with the previous one, otherwise it is converted to a new phase node. The attributes are also transformed to the new nodes as defined in the rules.

IM:Step-phase (name, taskname,servicetime,prestate,pname)	--	*LQN:PHASESTEP(entry_id, ph_ser_time);*

5.4 SUPPORTING TEMPLATES

This package also contains three modules, including basic templates, variable templates and specific templates. The basic templates are utility templates. Variable templates are utility variables that are used by the transformation algorithms. The specific templates are specific only for a given target model.

5.4.1 Basic Templates

Basic templates may be called utility functions (as they interact like functions) and they facilitate XSLT stylesheet writing. These basic templates themselves are also XSLT stylesheets but they are used similar to C/C++ fashion. Templates discussed here included "select", "tplug", "splug", "remove" and "gapifyAChild", which are defined in eXMLgebra descriptions and given in previous chapters.

"select" template:

There are a few slightly different versions of "select" template in the XSLT implementation and the difference is in the input argument. The template could take a node structure, a path, or a node name as the input identifier. As shown in Figure 5.3, the "select" template example shown below takes a *"$selectname"* as an input and traverses the input tree starting from the given path. The parameter *$path* is normally denoted as the root of the input tree, but can also be an absolute path of a node in the input tree. The "select" function basically seeks nodes whose name is identical to the *$selectname* and copies the entire subtree starting with the selected node (as shown in flow chart in Figure 5.3). The "copynodes" template copies the element name and attributes as well as all the child nodes via a recursive call.

```
<xsl:template name="select">
  <xsl:param name="selectname" select="$selectname"/>
  <xsl:for-each select="$path">
   <xsl:if test="name()=$selectname">
    <xsl:call-template name="copynodes"/>
   </xsl:if>
  </xsl:for-each>
</xsl:template>

<xsl:template name="copynodes">
   <xsl:element name="{name()}">
    <xsl:for-each select="@*">
     <xsl:attribute name="{name()}">
      <xsl:value-of select="."/>
     </xsl:attribute>
    </xsl:for-each>
    <xsl:for-each select="./*">
     <xsl:if test="not(name()='')">
      <xsl:call-template name="copynodes"/>
     </xsl:if>
    </xsl:for-each>
  </xsl:element>
</xsl:template>
```

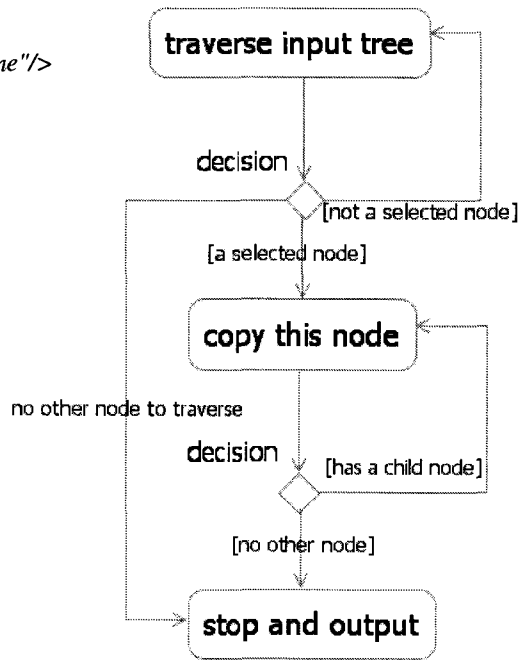


Figure 5.3: XSLT code fragment and flow chart of "*select*" template.

"*tplug*" template:

The "tplug" template takes two input trees and plugs the second tree into a GAP node in the first tree. The gap name *$g* can be omitted if there is only one gap in the first tree. The XSLT code shown in Figure 5.4 simply does the following. (1) if it finds the GAP, then plug in the second tree via *<xsl:apply-templates select="document($secondfile)/*"/>*;

and (2) output the node that is not GAP. The flow chart (shown in Figure 5.4) depicts the execution steps. The "tplug" template examines each node in the first input tree and looks for the GAP node. If it is not a GAP, the template outputs the node. If the GAP is found, the template plugs in and outputs the second tree. The template stops when there is no node left to examine.

```
<xsl:template name="tplug">
  <xsl:param name="g" select="$g"/>
  <xsl:choose>
   <xsl:when test="name()='GAP'">
    <xsl:if test="@name=$g">
      <xsl:apply-templates
        select="document($secondfile)/*"/>
    </xsl:if>
   </xsl:when>
   <xsl:otherwise>
    <xsl:element name="{name()}">
     <xsl:for-each select="@*">
      <xsl:attribute name="{name()}">
        <xsl:value-of select="."/>
      </xsl:attribute>
     </xsl:for-each>
     <xsl:for-each select="./*">
      <xsl:if test="not(name()='')">
        <xsl:call-template name="tplug">
          <xsl:with-param name="g" select=
        </xsl:call-template>
      </xsl:if>
     </xsl:for-each>
    </xsl:element>
   </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

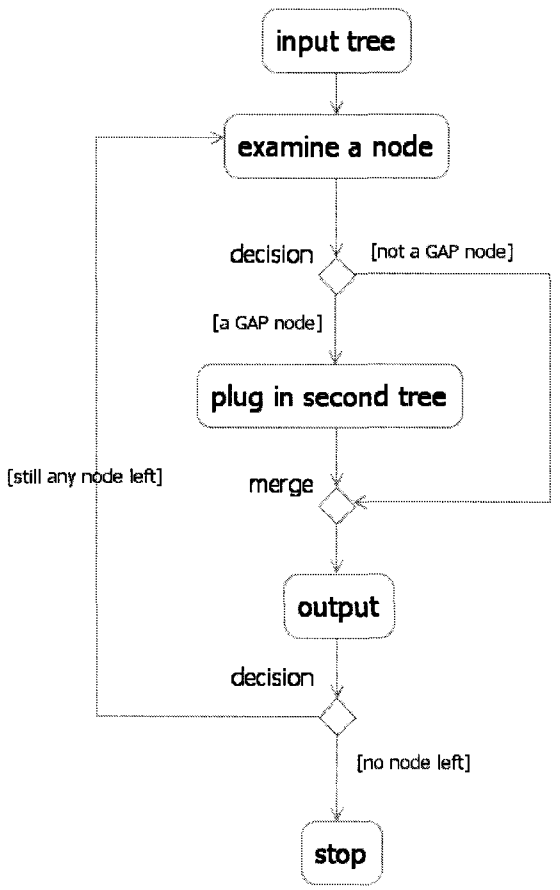


Figure 5.4 XSLT code fragment and flow chart of "*tplug*" template.

"*splug*" template:

In the XSLT implementation, the "splug" template is very similar to the "tplug" template. The only difference is that it looks for a GAP (either attribute or node GAP) and plugs in the value rather than a tree. The "splug" examines each node of the input tree and to find the attribute GAP. If there is no attribute GAP found, it outputs the node directly. If an attribute GAP is identified, the template assigns the value to the attribute GAP.

"remove" template:

The "remove" template (shown in Figure 5.5) is a strict version, which removes only a given node *v* and its descendants since both node type name and node name are checked (a node is expressed as *<NODETYPENAME name="nodename".../>*). A more relaxed version is only to check the node type name, which would remove all the nodes with the same node type name. Figure 5.5 also shows the flow chart of "remove" template, which checks if a given node is the one to be removed. If yes, it returns, which results in no output for this node and its descendents. For the rest nodes, the template simply outputs them as they are. The template stops when all nodes are traversed.

```
<xsl:template name="remove">
  <xsl:param name="nodetypename"
       select="$nodetypename"/>
  <xsl:param name="nodename" select="$nodename"/>
  <xsl:choose>
   <xsl:when test="(name()=$nodetypename) and
       (@name=$nodename)">
   </xsl:when>
   <xsl:otherwise>
    <xsl:element name="{name()}">
     <xsl:for-each select="@*">
      <xsl:attribute name="{name()}">
        <xsl:value-of select="."/>
      </xsl:attribute>
     </xsl:for-each>
     <xsl:for-each select="./*">
      <xsl:if test="not(name()='')">
        <xsl:call-template name="remove">
          <xsl:with-param name="nodetypename"
       select="$nodetypename"/>
          <xsl:with-param name="nodenmae"
       select="$nodename"/>
        </xsl:call-template>
      </xsl:if>
     </xsl:for-each>
    </xsl:element>
   </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

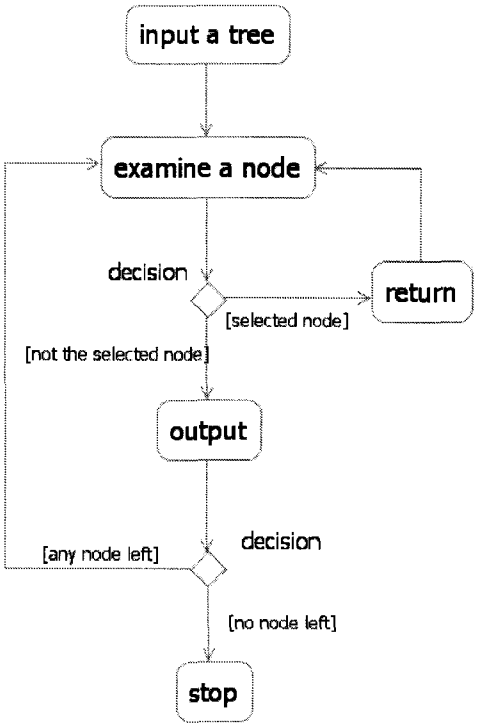


Figure 5.5: XSLT code fragment and flow chart of "*remove*" template.

"gapifyAchild" template:

The "gapifyAChild" template adds a new child GAP node to a selected parent node. The new node is always placed at the most right if there is an existing child node. The XSLT code (not shown) is relatively long but the execution logic is simple. The template looks for the "selected node" (which becomes the parent node of the gap). If it is not the "selected node", the template outputs the node. If it is, then it checks if any child node

exists. If not, the gap node is the only child node. If yes, then the gap node is placed in the most right position, e.g., the last child node.

"dom" template:

As defined in Chapter 4, an XML tree can be expressed as $t = (dom_t ; lab_t ; (\lambda_t^a)\ a \in A)$ where dom_t is a finite array expressed of each node in a domain over N. The lab_t: $dom_t \rightarrow \Sigma$ is a labeling function, and for every attribute name $a \in A$, λ_t^a: $dom_t \rightarrow D \cup G$, is an attribute value function. The XSLT implementation of "dom" template shown in Figure 5.6 not only implements the "dom" function but the "lab" functions as well. Also, the attribute in XSLT makes the implementation of the λ_t^a function easy. For example, *<xsl:value-of select="@name"/>* returns the value of attribute "name", which is equal to λ_t^{name}.

```
<xsl:template name="dom">
 <xsl:param name="number" select="$number"/>
   <xsl:value-of select="$number"/>
   <xsl:value-of select="concat('-->',name())"/>
   <xsl:value-of select="$newline"/>
   <xsl:for-each select="./*">
     <xsl:if test="not(name()='')">
      <xsl:call-template name="dom">
        <xsl:with-param name="number"
              select="concat($number,',',position())"/>
      </xsl:call-template>
     </xsl:if>
   </xsl:for-each>
</xsl:template>
```

Figure 5.6: XSLT code of "*dom*" template.

```
1-->xmi:Root
1,1-->ownedMember
1,1,1-->eAnnotations
1,1,1,1-->details
1,1,2-->ownedAttribute
1,1,3-->deployment
1,2-->ownedMember
1,2,1-->eAnnotations
1,2,1,1-->details
1,2,2-->ownedAttribute
1,2,3-->deployment
```

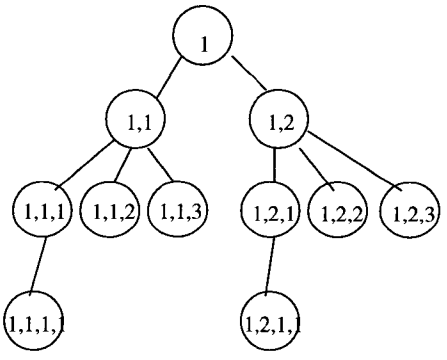


Figure 5.7: An output example of "*dom*" template.

Figure 5.7 shows the output result of the "dom" template (left side) by taking in an input tree structured as in the right side of the figure.

5.4.2 Variable Templates

Variable templates are used to generate variable sets that hold values of different kind of nodes and their attributes. For example, a "node" variable set is a collection of "node" names and attributes or a "step" variable set is a collection of "step" names and attributes. These templates, along with the basic templates, provide a foundation for the transformation templates. Figure 5.8 shows a hardware variable template, which creates a variable called *$hardware-content* that contains a collection of hardware names and attributes. This template basically traverses the UML XML model and copies the names and attributes of all node named "uml:Node" ("uml:Node" is a node type that corresponding to hardware node expressed in uml deployment diagram). There are more variable templates, including *$node-content*, *$transition-content* and *$group-content* etc.

Variable templates create run time variables to facilitate the transformation. These variables have a comment expression as follows:

$$(NODETYPE\ ;\ (attribute\ name:\ attribute\ value;)_{n=1,2,3...n}{}^{\wedge *})_{m=1,2,3...m}$$

```
<xsl:variable name="hardware-content">
  <xsl:apply-templates select="document($hardwarefile)/xmi:Root/ownedMember"/>
</xsl:variable>
<xsl:template match="xmi:Root/ownedMember">
    <xsl:if test="@xmi:type='uml:Node' ">
      <xsl:value-of select="name()"/><xsl:text>:name:</xsl:text>
      <xsl:value-of select="@name"/><xsl:text>;xmi-type:</xsl:text>
      <xsl:value-of select="@xmi:type"/><xsl:text>;xmi-id:</xsl:text>
      <xsl:value-of select="@xmi:id"/>
            <xsl:for-each select="./*">
              <xsl:if test="name()='deployment'"><xsl:text>;deployedArtifact:</xsl:text>
              <xsl:value-of select="@deployedArtifact"/><xsl:text>;</xsl:text>
       </xsl:if>
      </xsl:for-each>
     <xsl:text>^</xsl:text>
    </xsl:if>
</xsl:template>
```

Figure 5.8: XSLT code of a variable template.

* ^ is a seperator.

5.4.3 Specific Templates

Specific templates are target model specific and they handle attributes and other specific transformations. It is important to decouple this module from the rest since it is target model dependent. The main task of these templates is to address attribute transform, including both names and values. Figure 5.9 is an example how to assign attribute values for *"IM:Hardware"* node which has two attributes "name" and "taskname". The "name" attribute value is directly assigned by taking the name attribute value of the source node ("ownedMember" of the uml model) as denoted by step (a). However, the request value "taskname" can be obtained via a reference in the hardware set and the latter has a reference to artifact set that contains the value of "taskname". A supporting template "get-value" is called to sort out the referencing relation between nodes. Figure 5.9 illustrates how to obtain the value for target attribute "taskname". The "taskname" value assignment takes three steps (b1 to b3). Step (b1) calls the "get-value" template that takes a reference "xmi:id" to find the "deployedArtifact" id. Step (b2) "get-value" template again by taking the "deployedArtifact" id as a reference to find the "Artifact" name. Step (b3) assigns the found name to "taskname". In this particular example, the "get-value" template has been called twice, indicating an indirect referencing. The XSLT code fragment (1) and (2) shown in the figure are corresponded to step (a) and (b) in the attribute transformation.

The referencing relations between a source and target attributes are defined by the source model. Each attribute has different a referencing relation, which makes the attribute transformation specific. In this study, there are three specific templates for UML model to IM, IM to LQM, and IM to CSIM transformations, respectively. The specific templates define every attribute transformation between the source and target models.

5.5 TRANSFORMATION ALGORITHMS

Transformation of a UML design model to a performance model contains steps including data extraction, sequence sorting and domain transformation. This package implements transformation algorithms that are defined earlier in eXMLgebra. Templates contained in this package are data extraction, sequence sorting and transformation templates.

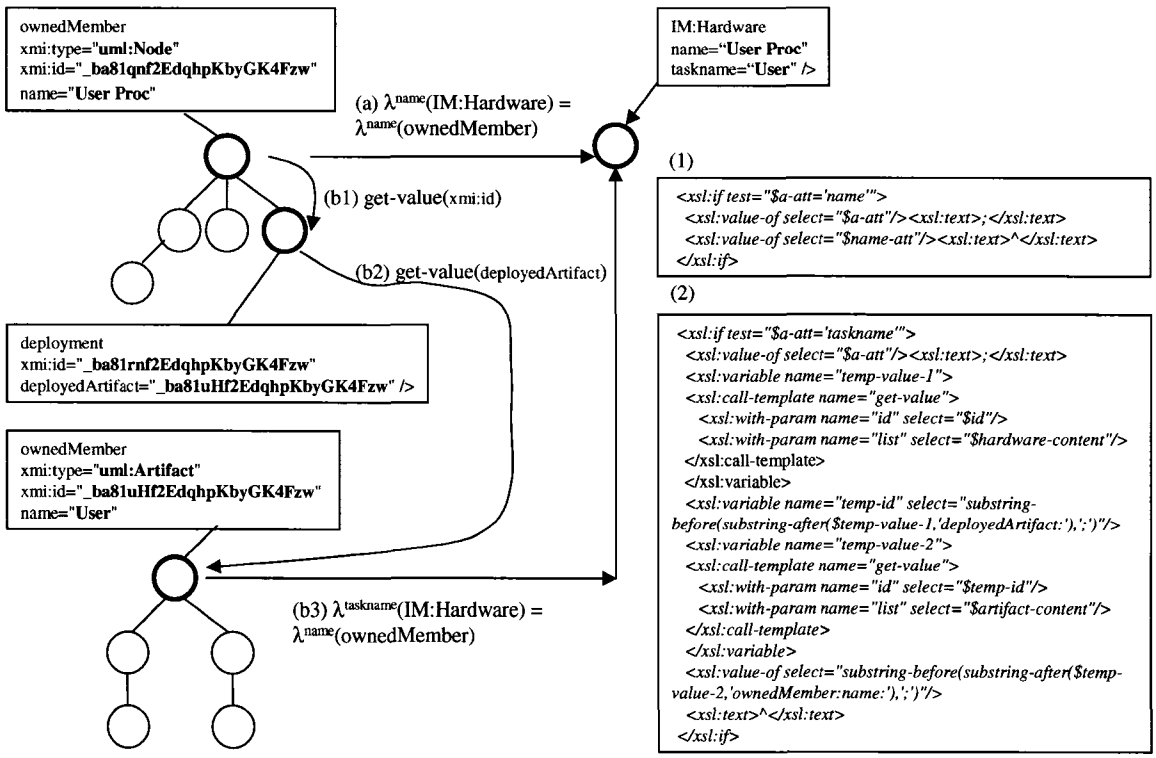


Figure 5.9: An example of how to assign attribute values in a specific template.

5.5.1 Data Extraction Templates

Extraction data is rather simple as indicated by the XSLT code below. It uses the "select" template and passes in the node type parameter. During UML to IM transformation, many subtrees (in UML XML tree) are extracted as these subtrees contain information necessary for building the performance model. Subtrees include "group", "node", "ownedMember" and others. The XSLT code below extracts any subtree if a "nodetype" name is provided. For example, if *$nodetype* is equal to *$node*, this template extracts all the subtrees of "node".

```
<xsl:template match="/">
<xsl:element name="xmi:Root">
   <xsl:call-template name="select">
      <xsl:with-param name="nodetype" select="$nodetype"/>
   </xsl:call-template>
</xsl:element>
</xsl:template>
```

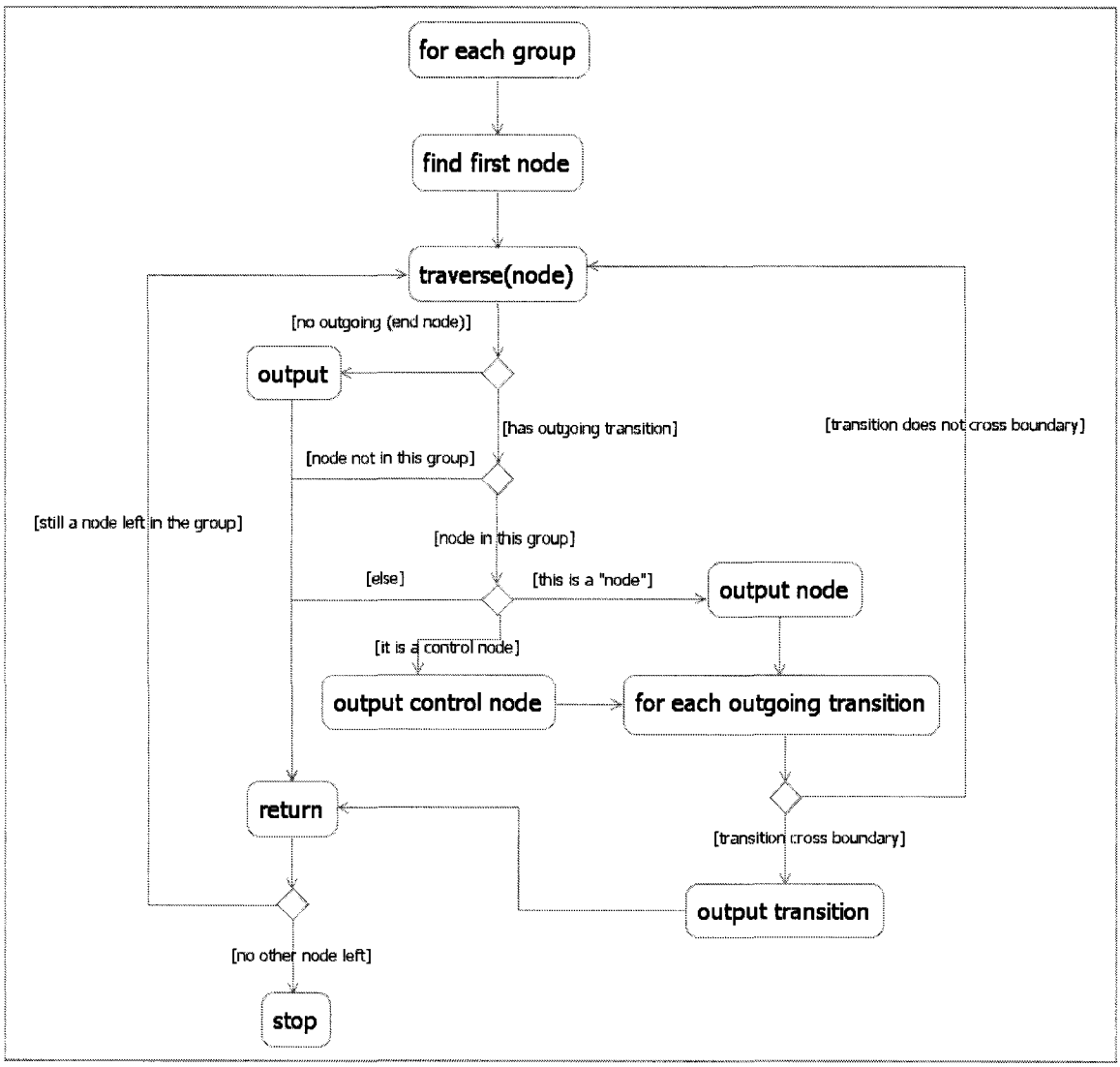


Figure 5.10: The algorithm of the sequence-sorting template.

5.5.2 Sequence Sorting Template

This template ensures that the execution sequence is the same as shown in the UML activity diagram. When the graphical presentation of an activity diagram is converted to XML format, the scenario execution sequence is hidden and implied by cross-references. Figure 5.10 shows the algorithm of the XSLT code implementation, which reconstructs the "*IM:Service*" subtree by re-examining the execution sequence. The template loops through all the "*IM:Service*" subtrees one by one and calls "sortsequence" template (where flow chart is shown in Figure 5.10). Starting from the initial step of each partition, the template traverses each node. If the current node has an outgoing transition, and its

target node is within the same partition (either a node or control node, e.g., fork and join), the template outputs this node. Then for each outgoing transition, do output the transition if it crosses the partition boundary. If the transition is within the same partition, there is no output. However, the algorithm goes back to traverse the next node. If a node has no outgoing transition, it must be the end node. The template outputs it and stops if there is no other node left to traverse.

The algorithm works well if every execution step is in sequence. In a concurrent case, it outputs the first execution path, which means that the output sequence would be path by path. To explain this point, let us take a look at a fork concurrent case shown in Figure 5.11. The output sequence will be (1) to (4) then (5) to (6). Such output sequence does not affect the further transformation since there are crossing references existing between the "fork" and "Action3".

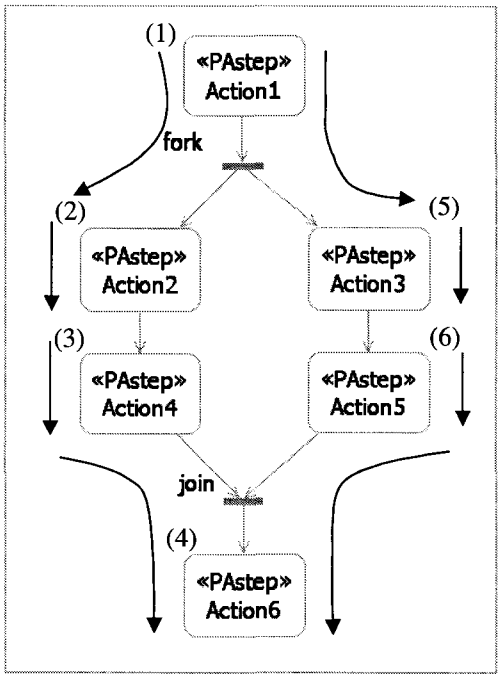


Figure 5.11: Output sequence for a fork concurrent case.

5.5.3 Transformation Template

This module converts an XML tree in one domain to a new XML tree in another domain. This process is done by the "transform" template (flow chart is shown in Figure 5.12), which contains two parts. The first part is to send each node in the input XML file to an

"apply-select" template. The latter looks up the transformation rules and returns the new node. The second part is to output the new node to an XML format. This transformation template is used to transform the extracted XML subtrees (complied with UML domain to IM subtrees complied with IM domain definition; IM XML model to LQN XML model; and IM XML model to CSIM XML model.

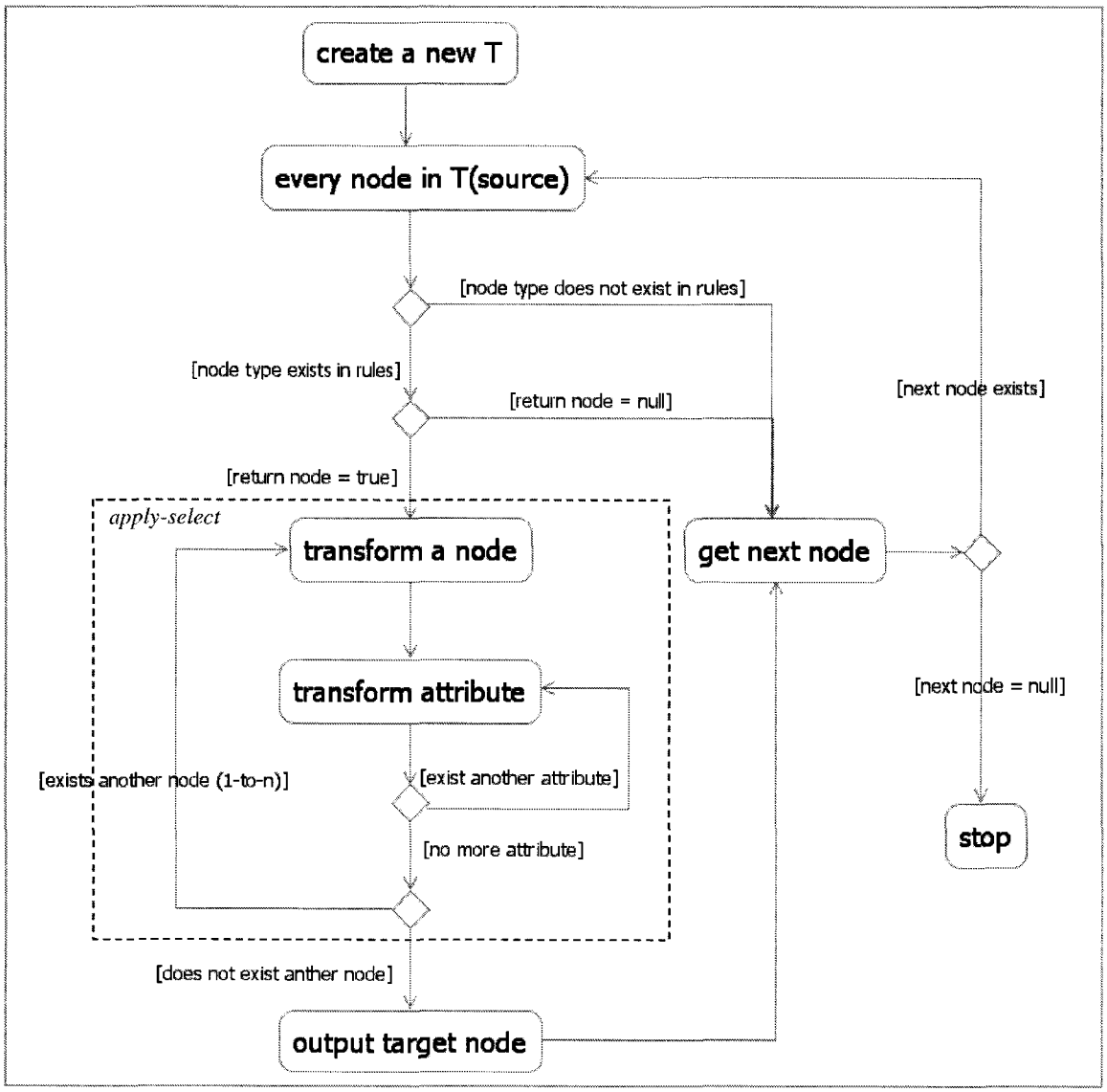


Figure 5.12: Flow chart of "*transform*" template algorithm.

Figure 5.12 briefly illustrates the action flow chart of the "transform" template. The transformation starts from the input source tree. Each node from the source tree is subjected to transformation. If the source node type is not registered in the rules or the

returned target node is null, the transformation goes to next source node; otherwise, each attribute of the target node is transformed. The transformation is repeated until there is no source node left.

5.5.4 IM XML Model Construction Template

Figure 5.13 is a flow chart of IM XML model construction template. Such a construction requires a DTD that defines the target tree and a collection of smaller trees that are readily to be plugged into the target tree. The construction starts with a node type from a DTD in XML format. If the node type already exists in the target tree (which means that it is already plugged in), then next node type is considered. If not, a gap is created and subtree of same type is plugged in. The algorithm repeats itself until all the node type in DTD is browsed.

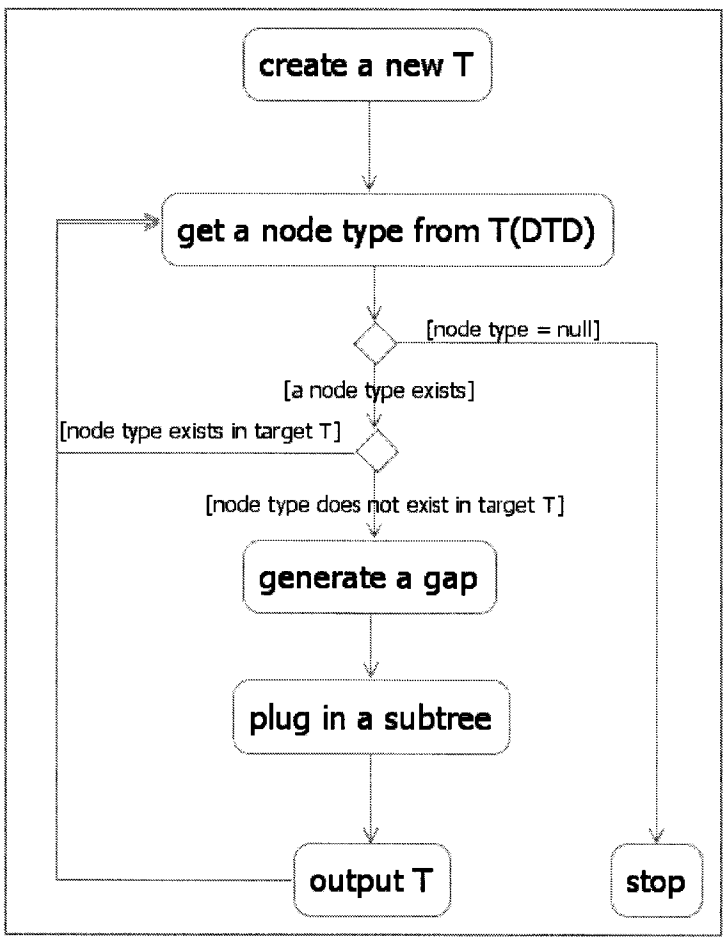


Figure 5.13: Flow chart of IM XML model construction algorithm.

5.5.5 From IM Model to LQN and CSIM Models

To transform IM to LQN, we require IM →LQN rules, a transformation algorithm (Figure 5.12) and an "apply-select" template. The transformation algorithm is generic, but the "select-apply" template is different for each transformation since specific cases are required to handle differently. In IM to LQN transformation, one of the special cases is to determine if an *"IM:Service"* is transformed to activity or phase as the execution steps are treated differently. The XSLT code deals with the transformation of *"IM:Step"* either to *"LQN:PHASESTEP"* or *"LQN:ACTIVITYSTEP"*. A variable *$phase-or-activity* is used to identify whether a task is a phase or activity if transformed; and it is determined by checking if this task makes a request call to other tasks or a concurrent fork operation. If the step belongs to an activity then the *"IM:Step"* is transformed to the *"LQN:ACTIVITYSTEP"*. If it is a phase step, it is handled in *"IM:Service"* to *"LQN:PHASE"* transformation, where the *"get-phase-value"* template handles the addition of service time within the same phase (as discussed in Chapter 4). Similar to LQN, the IM to CSIM transformation requires also rules, generic transformation algorithms, and a "select-apply" template that is specific for the CSIM model. Since the CSIM simulation model retains most of the execution details, the IM to CSIM transformation involves mainly one-to-one transformation, and in some cases, additional information must be introduced. For example, CSIM:MailBox is introduced to simulation inter-processor communication.

The XSLT code implementations for both IM→LQN and IM→CSIM are relative tedious; therefore, they are not shown in this section. For more details, one may refer to the study report [42]. Also, the entire transformation of UML design to performance model is automatic. All the transformation templates mentioned above are packaged into a make file, which can be run together. Many intermediate results are output and they could be examined for verification purpose.

5.6. SUMMARY

In this chapter, we discussed the XSLT implementation based on the definitions provided in eXMLgebra notation in the previous chapters. The implementation contains four

packages, including domain definition, transformation rules, transformation algorithms, and supporting templates. The transformation algorithm package is the core, which is supported by others. In the next chapter, a set of test cases is designed to verify the correctness of the code implementation. Test cases include both behavior testing and end-to-end model transformation testing.

CHAPTER 6: VERIFICATION AND CASE STUDY

6.1 INTRODUCTION

This thesis focused on the development of a transformation methodology from UML 2.0 design models to performance models (LQN and CSIM). The transformation algorithms were defined (in eXMLgebra) and implemented (using XSLT) and tested out. Six simple models representing different scenarios and two end-to-end model test cases were used to verify the correctness of the transformation.

Behavior scenario models included are:

1. Synchronous Messaging
2. Asynchronous Messaging
3. Optional Combined Fragment
4. Parallel Combined Fragment
5. Loop Combined Fragment
6. Passive Resource

The two end-to-end models are:

7. 3-tier Client Server Application
8. The Building Security System (BSS) Application

The initial simple test cases (1–6) from Table 6.1 are designed to test transformation rule elements such as step, join, fork, transition and others in various combinations. The test procedure includes the following steps:

a) Construction of a test case using IBM Rational Software Architect (which complies with the UML 2.0 specification) with performance annotation.
b) The UML model is exported to a file in XML format and a primary error check is applied.
c) The input UML model then is run through data extraction, transformation to generate the intermediate model.
d) The intermediate model is used as input to generate LQN and CSIM model.
e) Each output model is compared with its high-level graphical presentation (derived from scenario analysis) to examine its correctness.

The end-to-end test cases (7-8) were tested out using a mixed combination of some or all transformation rules. Test 7 is a 3-tier client/server application model, which has been discussed in Chapter 4 extensively. Test Case 8 includes both Access Control and Acquire/Store Video Scenarios of a Building Security System presented in [135], which are using different behavioral features such as synchronous messaging, asynchronous messaging and looping etc.

Both LQN and CSIM models produced by the transformation algorithms are in XML format. These XML documents are being produced using the SAXON transformation engine built in Java [59]. The output XML files were verified using XMLSpy, which discovers errors based on the given XMI Schema. XMLSpy confirmed the valid format of the produced XML documents.

Table 6.1 List of Test Cases Summary

	Synchronous Message	*Asynchronous Message*	*Step (one-to-one transformation)*	*Optional Combined (Or-fork/or-join) Fragment*	*Parallel Combined (fork/join) Fragment*	*Composite step*	*N-to-one transformation*	*One-to-n transformation*	*Loop Combined Fragment*
1. Synchronous Messaging	X		X		X		X		
2. Asynchronous Messaging		X	X				X		
3. Optional Combined Fragment			X	X					
4. Parallel Combined Fragment			X		X				
5. Loop Combined Fragment			X			X			X
6. Passive Resource	X	X	X						
7. 3-tier Client Server Application	X		X		X		X	X	
8. The Building Security System	X	X	X	X	X	X	X	X	X

In the following sections, we only compare the performance model outputs with their corresponding high-level graphical models. All the transformation details are not presented since the transformation steps have been explained in the previous Chapters 4

and 5. Also, due to the large size of the output XML model, it is not convenient to represent the entire XML performance model; therefore, fragments of the output results are normally shown in this chapter. In fact, to avoid overwhelming the presentation with details, the LQN and CSIM XML models are presented in Appendix IV. For the test case (8), the entire output is given in a study report [42].

6.2 ERROR CHECKING

To ensure a successful transformation, an "error checking" template is run to examine the input UML XML model file. This template can catch errors that are due to incorrect graphical modeling or errors that are introduced due to the XML conversion. For example, a deleted element in the graphical representation may appear in the XML file due to a "dirty" delete (an element is deleted from the graph but not from the model). Other errors may be introduced by incorrect modeling. Examples of these errors include missing a reply transition for a synchronous call, a transition node without a source or target node, an element without name, etc. Error checking is also carried out during the transformation. Each node being transformed is evaluated against the rules. If a node type cannot be recognized by a rule set, an error message of "unrecognized node type" will be given. Table 6.2 lists a number of error types (both errors and warnings, depending on the severity) that are checked, and their corresponding error messages.

Table 6.2: List of error types that are checked

Error Type	***Error Message***	***Description***
Error	*Unknown Node name*	*An element without name*
Error	*Reference missing*	*A component fails to link to the referenced node*
Error	*Synchronous call mismatch*	*Reply synchronous transition can not be found*
Error	*Duplicate initial node*	*Duplicate initial node*
Error	*Unrecognized node type*	*Node type can not be recognized by rule sets*
Error	*Annotation is missing*	*Performance annotation is not found*
Warning	*Pattern not found*	*Stated pattern can not be identified*
Warning	*Zero service time*	*no service time is found*

"Error" must be corrected or removed from the input file to ensure structural correctness of the output XML file. "Warning" can sometimes be ignored, as normally it does not terminate the transformation. However, careful examination is recommended. Table 6.2 lists only few of the common errors encountered in this study.

6.3 TEST CASE 1: SYNCHRONOUS MESSAGING

This test case uses a simple client/server model to test the synchronous messaging behavior. Figure 6.1(a) is a UML 2.0 activity diagram showing a blocking synchronous messaging case. Both client and server are stereotyped as <<PAresource>> while action steps involved are stereotyped as <<PAstep>> in accordance with the SPT Profile [78]. The service time of each step is also documented in the property of each action step (for example, {PAdemand=('est','mean',3,'ms')} is annotated for both "send" and "process" steps while {PAdemand=('est','mean',0.7,'ms')} is for "receive" step.). The calls ("send" and "reply") are stereotyped as <<synchronous>>. Stereotype <<asynchronous>> is used for asynchronous calls (see later sections). This annotation facilitates the transformation. However, the XSLT implementation in this work also handles the situation where a transition stereotype is not specified, in which case an algorithm is invoked to check if a reply call exists when a request call is identified.

The client/server model is transformed to an intermediate model first, then to LQN and CSIM performance models. The intermediate model, as discussed in Chapter 4, records every step and transition that crosses the partition boundary in the UML activity diagram. Figure 6.1(b) shows what a UML activity diagram (6.1(a)) would become, if transformed to a LQN performance model. The steps in each task (client and server) are grouped into phases through an abstraction-raising aggregation (the shaded area).

Similarly, Figure 6.1(c) is the execution diagram of Figure 6.1(a) when it is transformed to CSIM simulation model. As mentioned previously, the circle inside a rectangle represents a process. "MBS" and "MBR" are mailbox send and receive operations. A mailbox is used to simulate inter-process communication. Elements (steps) under a process execute in that process. In this example, the client sends a call (via mailbox) to server and waits for a reply. The server is blocked by its mailbox receive. If a message arrives, the server executes the "process" step, sends a reply, and waits at the mailbox again for the next coming message.

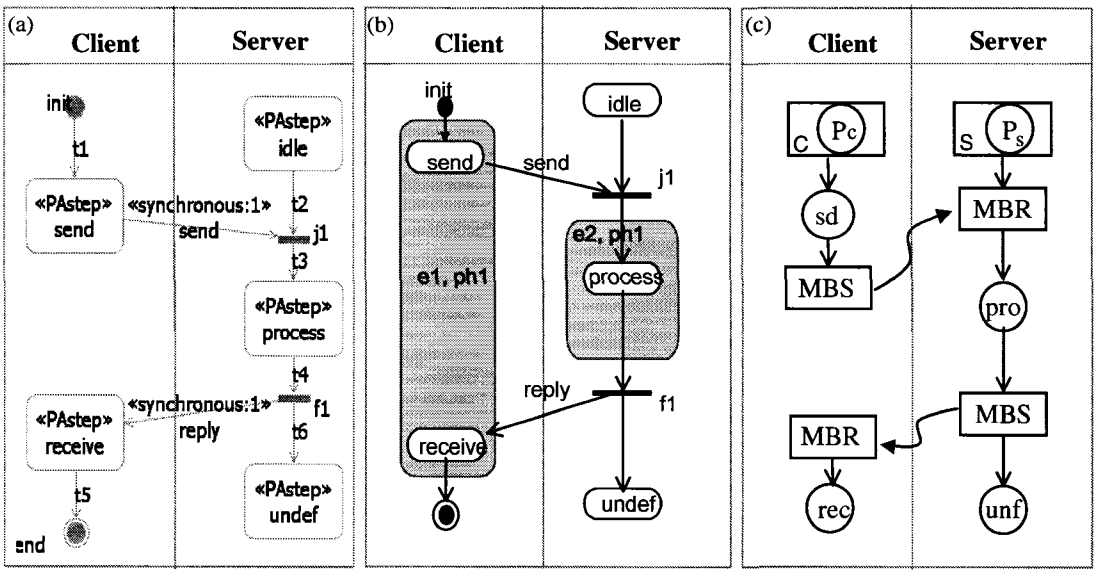


Figure 6.1: A blocking synchronous messaging. (a) UML activity diagram, (b) assignment to LQN entries and phases, and (c) CSIM execution diagram.

The client/server XML output model for IM, LQN and CSIM are given in Appendix IV. The IM model basically contains every activity steps from the UML design. There are two "IM:Task" trees and each of them has an "IM:Service". The elements shown in the Figure 6.1(a) are well represented inside the "IM:Service" trees. Some of the attributes have no value since this test is subjected to the activity diagram only. The LQN XML model is relatively simple; and steps are grouped into phases as indicated by shaded areas in Figure 6.1(b). In the CSIM XML model, the "fork" and "join" operations (that are for sending and receiving message) shown in the UML diagram (Figure 6.1(a)) are not explicitly shown. This is because the CSIM18 engine provides concurrent execution for mailbox "send" and "receive". However, a fork or join operation that is not specifically for mailbox operations will have to be explicitly expressed (see examples discussed later).

6.4 TEST CASE 2: ASYNCHRONOUS MESSAGING

This test case uses a pipeline and filters pattern to test asynchronous messaging behavior. As shown in Figure 6.2(a), Filter1 sends an asynchronous call while continuously

executing remaining steps. Filter2, however, waits for the coming call then executes the steps in sequence. It should be noted that the asynchronous call is stereotyped as <<asynchronous>> to distinguish it from a synchronous call. If a call is not stereotyped and it is without reply, it will be assigned as an asynchronous call. In this test case, the XSLT implementation works both ways. If the pipeline and filters pattern is specified in a UML diagram, an asynchronous call will be expected. To transform this model to LQN, all the steps are grouped into phases, as indicated in Figure 6.2(b). LQN implies that Filter2 waits for the coming asynchronous call. The XML outputs, shown in Appendix IV for both LQN and CSIM, are in agreement with the high-level graphical representations shown in Figure 6.2(b) and Figure 6.2(c).

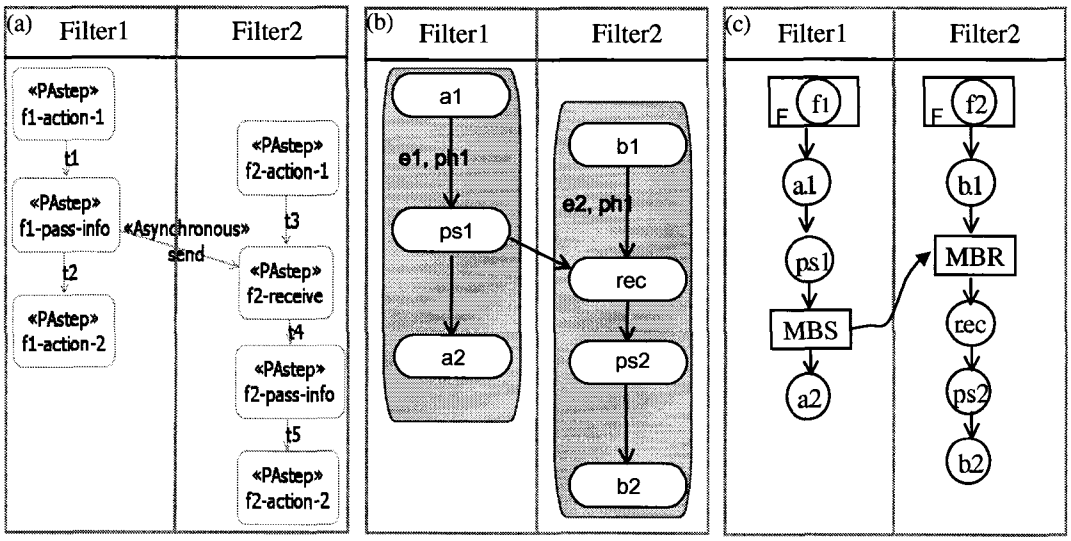


Figure 6.2: An asynchronous messaging pipeline example. (a) UML activity diagram, (b) assignment to LQN entries and phases, and (c) CSIM execution diagram.

6.5 TEST CASE 3: PARALLEL COMBINED FRAGMENT

This is a simple concurrent case to test the join and fork operations. This is a case where activities are introduced in LQN due to the presence of an internal "fork". Figure 6.3(a) is a UML activity diagram with performance annotations (in stereotypes). Because of the "fork" operation, the corresponding LQN model contains activities instead of phases; the shaded steps are activity steps as indicated in Figure 6.3(b).

In the CSIM model, a "fork" operation creates a new child process that executes concurrently with the main process. In Figure 6.3(c), process "Cp" is denoted as the child

process of the main process "App". The LQN XML output has eight elements including six activity steps, a fork and a join operation. The CSIM XML output model is also similar. The IM, LQN and CSIM XML output models are given in Appendix IV.

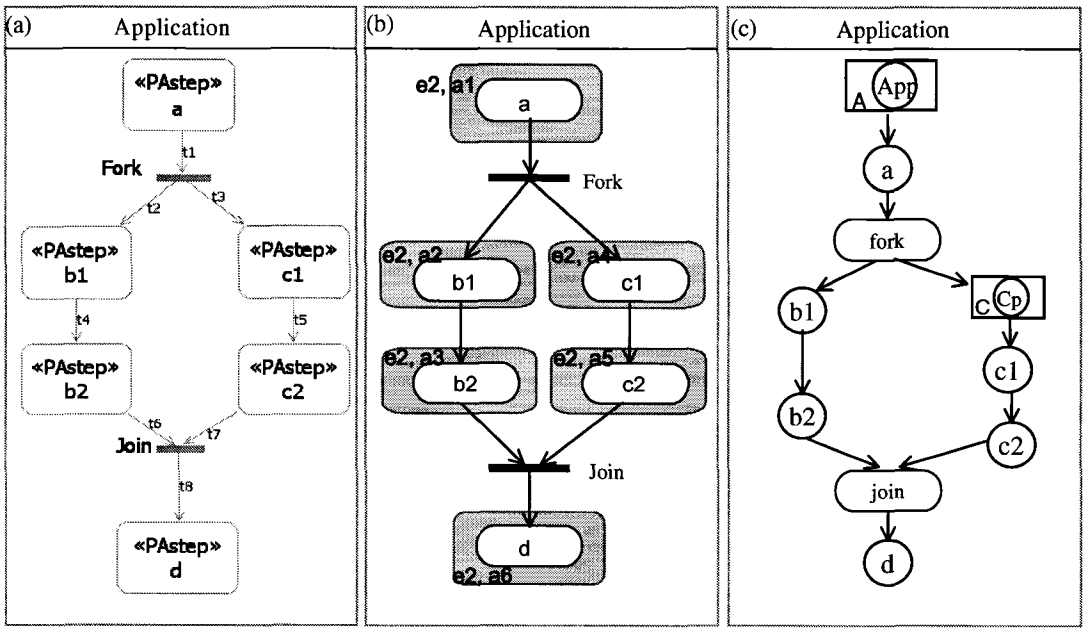


Figure 6.3: A fork/join fragment example. (a) UML activity diagram; (b) assignment to LQN entries and activities, and (c) CSIM execution diagram.

6.6 TEST CASE 4: "OR-FORK/JOIN" FRAGMENT

An "or-fork" is a decision node that has one incoming edge and multiple outgoing activity edges. Guards of the outgoing edges are evaluated to determine which edge should be traversed. On the other hand, an "or-join" is a merge node that has multiple incoming edges and a single outgoing edge. An outgoing edge from "or-fork" normally has a guard condition so that only one path is allowed at a given condition. The "or-join" normally is used with "or-fork". Figure 6.4(a) shows a UML activity diagram example containing "or-fork" and "or-join". Probabilities can also be assigned to "a1->a2" and "a1->a3" transitions. There is no waiting time in the "or-join".

Figures 6.4(b) and 6.4(c) are high-level execution flow chart of LQN and CSIM models. LQN treats the "or-fork/or-join" similar to "fork/join" and the partition "Application" is treated as containing LQN activities. The transitions "a1->a2" and "a1->a3" are guarded

with condition or probabilities. The IM, LQN and CSIM XML outputs are given in Appendix IV.

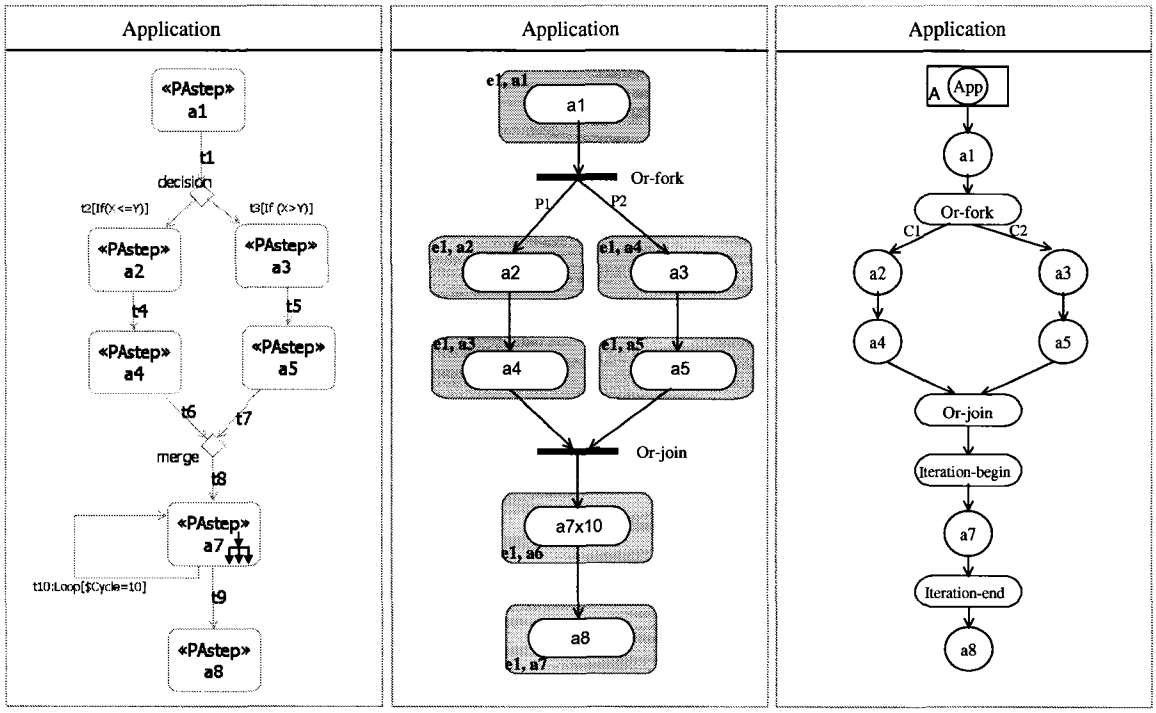


Figure 6.4: An example of "or-fork" and "or-join".

6.7 TEST CASE 5: LOOP COMBINED FRAGMENT

In Figure 6.4(a), step "a7" has a loop operation repeated $Cycle=10 times. The loop section can be considered as a composite step (the arrow symbol displayed in step "a7" indicating an aggregation of steps). The aggregated scenario can be treated separately as indicated in Figures 6.5(a) to (c). Step "a7" consists of three steps (b1, b2 and b3). In fact, the sub-scenario "a7" is drawn in a separated activity diagram with the name "a7". During the transformation, a composite step will be identified and its aggregated scenario will be matched. The loop transition is transformed to *"IM: Iteration"* in the intermediate model, with attributes to record begin-node and end-node. It is convenient to separate the steps in a loop as a sub-scenario, which allows the loop operation to be handled separately. In the LQN transformation, the sub-scenario is treated as a phase where all steps are aggregated (Figure 6.4(b)) and the iteration of step "a7" becomes a

multiplication of its service demand with the number of repetitions (which is shown in Figure 6.4(b) as "a7 x 10").

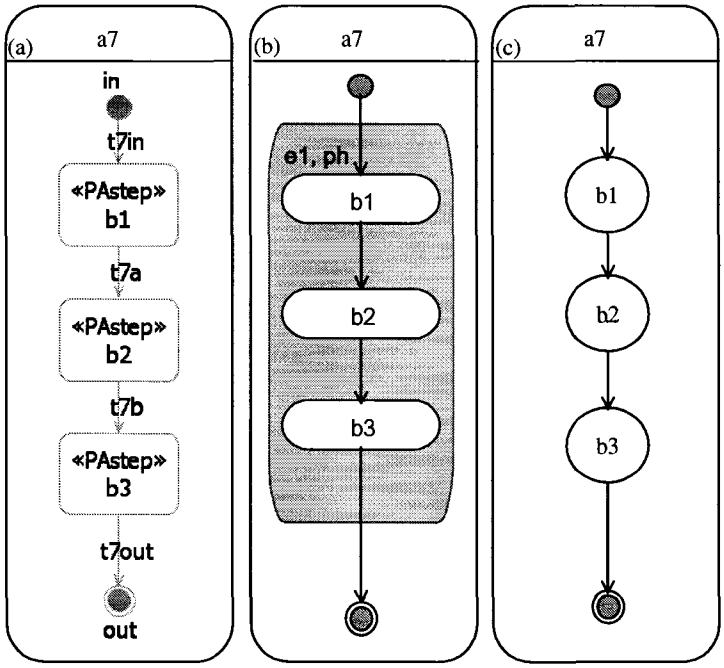


Figure 6.5: Sub scenario of composite step "a7".

In the CSIM transformation, *"IM:Iteration"* is transformed to *"CSIM:Iteration-begin"* and *"CSIM:Iteration-end"* as indicated in Figure 6.4(c). The aggregation details are to replace the composite step. In this case, steps b1 to b3 replace the composite step "a7". The following steps illustrate how the above example is handled.

a) During the construction of the intermediate model, in the case of more than one activity diagrams, the main activity diagram is identified (by text attribute of the diagram) and processed first. In the above sample, the main "IM:ServiceEntity" contains an iteration node and a composite step node that has an attribute referencing to the sub-scenario (or "IM:ServiceEntity" named "a7");

b) In the LQN model, the sub-scenario becomes a phase node and the iteration and composite step are simply grouped in to a LQN phase node with a service time value of "a7x10".

c) In the CSIM model, the composite step is replaced by its sub-scenario steps and "IM:Iteration" becomes "CSIM:Iteration-begin" and "CSIM:Iteration-end" nodes.

The output IM, LQN and CSIM XML models for Figures 6.4 and 6.5 are given in Appendix IV.

6.8 TEST CASE 6: PASSIVE RESOURCE

A buffer is a passive resource and normally managed by the buffer manager. Use of a resource is considered as a pattern, as discussed in Chapter 4 and should always be denoted in UML design diagrams. In the proposed transformation, the use of resource pattern is a special case since it is treated differently in the LQN and CSIM models. In the activity diagram from Figure 6.6(a), a simple use of a buffer is presented, which includes the acquiring of the buffer indicated by the stereotype <<GRMacquire>> and its release by <<GRMrelease>>. In this example, $Nbuf is defaulted to 1. Once a buffer is acquired, the application uses it and releases it when completed. The activity diagram (Figure 6.6(a)) shows that these operations are in the same task (App) in the software. However, they are separated in the LQN model into a pseudo task (App', Figure 6.6(b)), which executes while App is blocked. The reason to introduce a pseudo task (App') is to avoid cyclic calling, which is not allowed in LQN. It is noted that the introduction of a pseudo task does not affect the behavior of the model.

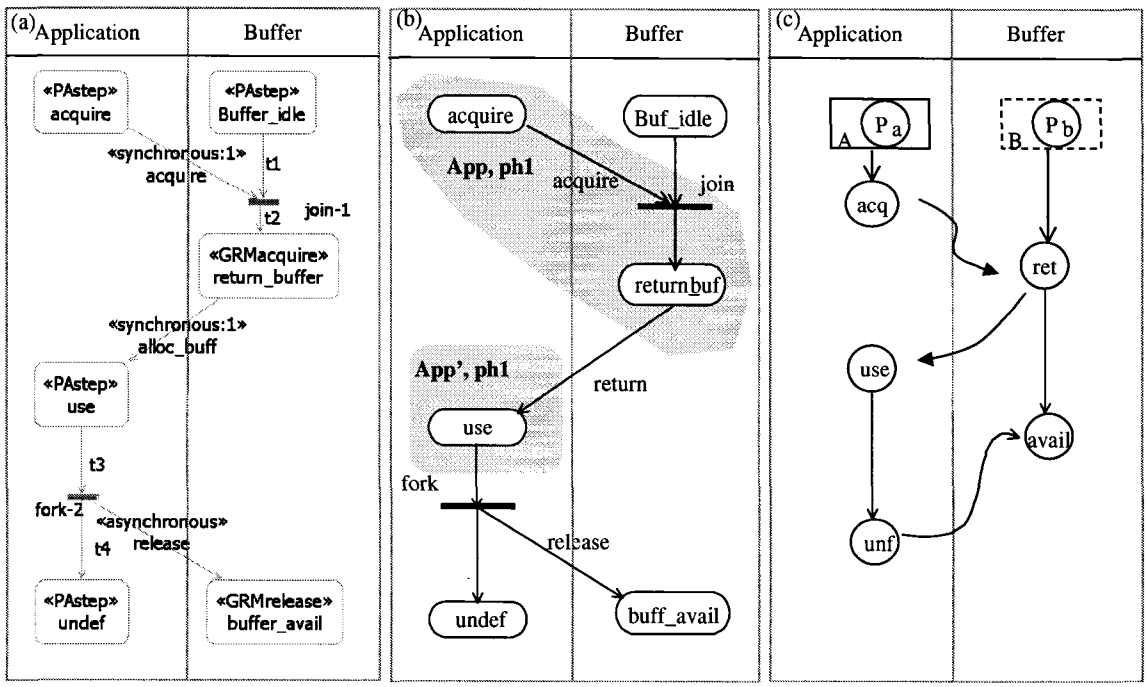


Figure 6.6: An example of passive resource or buffer.

In CSIM, Buffer is a resource (dashed line rectangle is used to differentiate it from a process) and there is no inter-process communication. The Buffer allocation is executed under the App process, which acquires, uses, and releases the Buffer as indicated in Figure 6.6(c).

The XML output models including IM, LQN and CSIM are given in Appendix IV. The correctness of both files was verified against their high-level presentations.

6.9 TEST CASE 7: 3-TIER CLIENT SERVER APPLICATION

This example has been discussed in Chapter 4 extensively. The high-level software components (User, Application and Database), and their relationships in the form of the Client-Server Pattern are shown in Figure 4.3(a). The deployment diagram, where the hardware components are linked according to the design blueprint, is given in Figure 4.3(b). The activity diagram that represents a key scenario of the 3-tier client-server model is given in Figure 4.3(c). The performance annotations are detailed in Table 6.3.

Table 6.3: List of some of performance annotations for the 3-tier client-server model

Task	***Step***	***Performance Annotation***
X (User)	*Send*	*{PAdemand=('est','mean',3,'ms')}*
	Receive	*{PAdemand=('est','mean',0.7,'ms')}*
Y (Application)	*Y1*	*{PAdemand=('est','mean',1.5,'ms')}*
	Y2	*{PAdemand=('est','mean',1.2,'ms')}*
	Y3	*{PAdemand=('est','mean',1.1,'ms')}*
	Y4	*{PAdemand=('est','mean',0.5,'ms')}*
	Y5	*{PAdemand=('est','mean',1.0,'ms')}*
Z (Database)	*Z1*	*{PAdemand=('est','mean',5.0,'ms')}*
	Z2	*{PAdemand=('est','mean',5.0,'ms')}*
	Z3	*{PAdemand=('est','mean',2.0,'ms')}*

For comparison purposes, the UML 2.0 activity diagram of the 3-tier client-server model is shown again in Figure 6.7(a). The partitions X, Y and Z represent the User, the Application and the Database, respectively. The corresponding execution diagrams for LQN and CSIM are shown in Figure 6.7(b) and (c). In Figure 6.7(b), the partition Y is treated as containing LQN activities instead of phases, since there exists a fork operation with a request call.

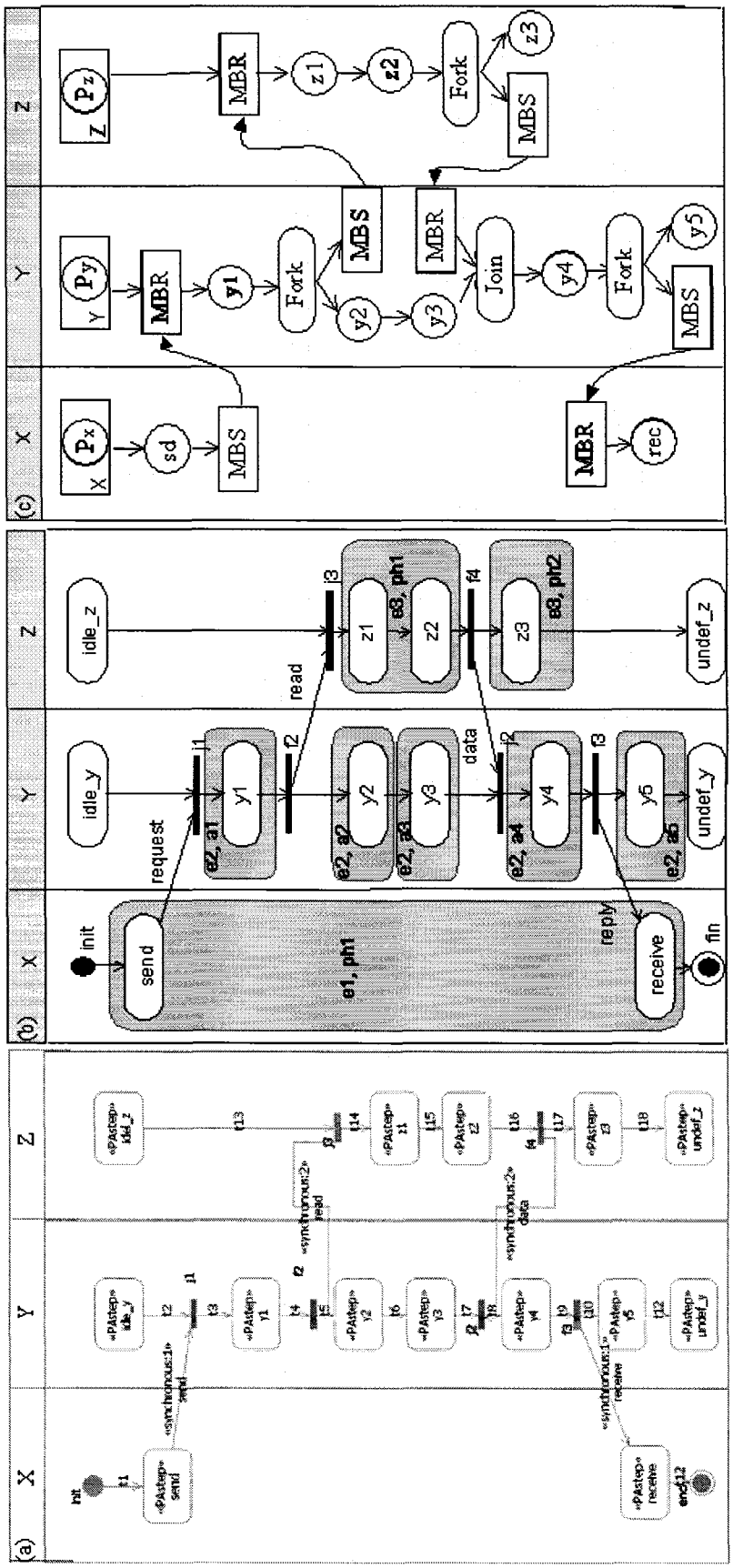


Figure 6.7: 3-tier client server application. (a) UML activity diagram, (b) assignment to LQN entries, phases and activities, and (c) CSIM execution diagram.

In Figure 6.7(c), the CSIM model retains the execution details described in the UML activity diagram (Figure 6.7(a)). Each execution step is reflected in the CSIM XML model and a synchronous call (either request or reply) is represented by a pair of "MBS" and "MBR". All the details of this example transformation can be referenced to Chapter 4 and the output XML models for IM, LQN and CSIM are also given in Appendix IV.

6.10 TEST CASE 8: CASE STUDY OF A BUILDING SECURITY SYSTEM

In order to test thoroughly the correctness of the transformation implementation, a more complicated example, a Building Security System (BSS), was used. The Building Security System is intended to control access and to monitor activity in a building such as a hotel or a university laboratory. This system was also analyzed elsewhere [135]. In here, two scenarios, related to control of door locks by access cards and video surveillance, are considered. In the Access Control scenario, a card is inserted into a card-reader, read and transmitted to a server, which verifies the access rights associated with the card in a database, and then either triggers the lock to open the door, or denies access. In the Acquire/Store Video scenario, video frames are captured periodically from a number of web cameras located around the building, and stored in the database. The system may have other use cases, but for simplicity we assume that the main performance concerns relate to the two use cases described above. Both scenarios have delay requirements. The access control scenario has a target completion time, and the surveillance cycle has a target time between consecutive polls of a given camera. In both cases it is supposed that 95% of responses, or of polling cycles, should meet the target delay.

In the design of such a system, the deployment diagram is used to define the assignment of software artifacts to nodes. A UML 2.0 deployment diagram consists of Nodes, Artifacts and Components. Nodes are connected through communication paths to create network systems of arbitrary complexity.

Artifacts represent concrete elements in the physical world that are the result of a development process such as model files, source files, scripts, and binary executable files

etc. Components are the concrete physical renderings of one or more model elements by an artifact. Artifacts are deployed to Nodes and components are manifested.

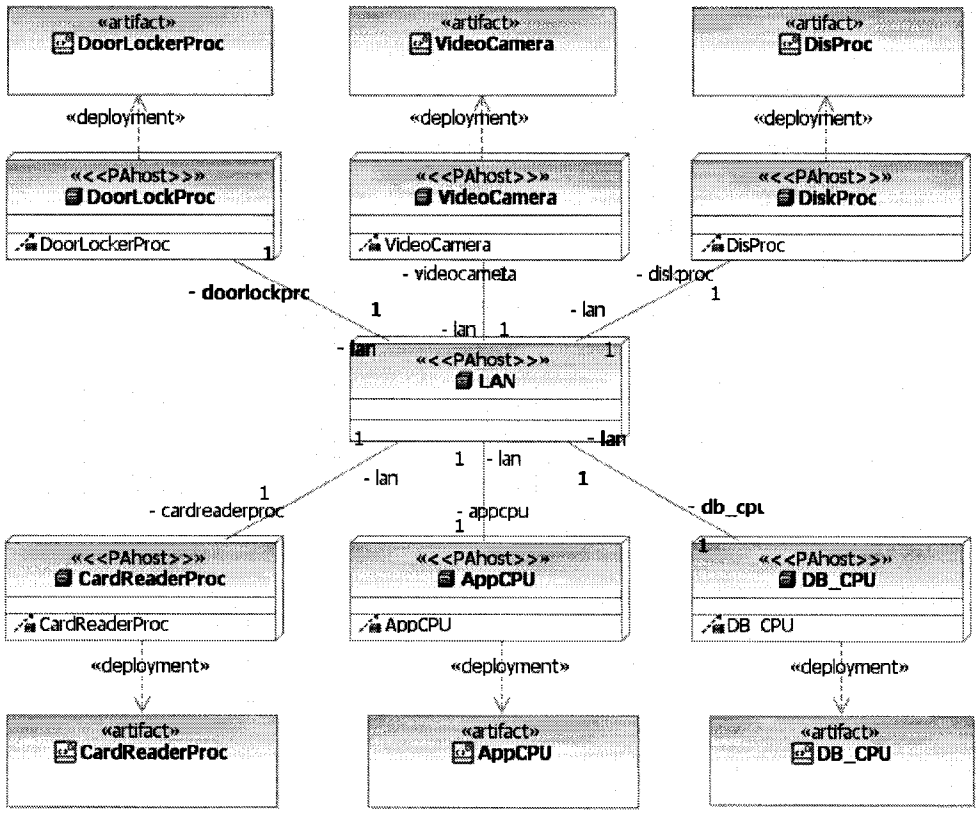


Figure 6.8: A hardware deployment diagram of BSS.

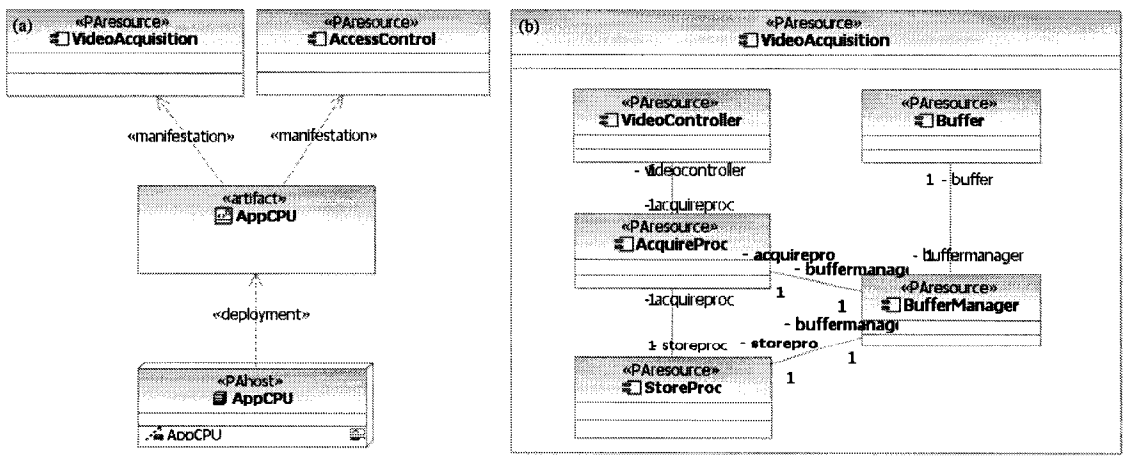


Figure 6.9: (a) AppCPU deployment diagram and (b) component structure of video Acquisition.

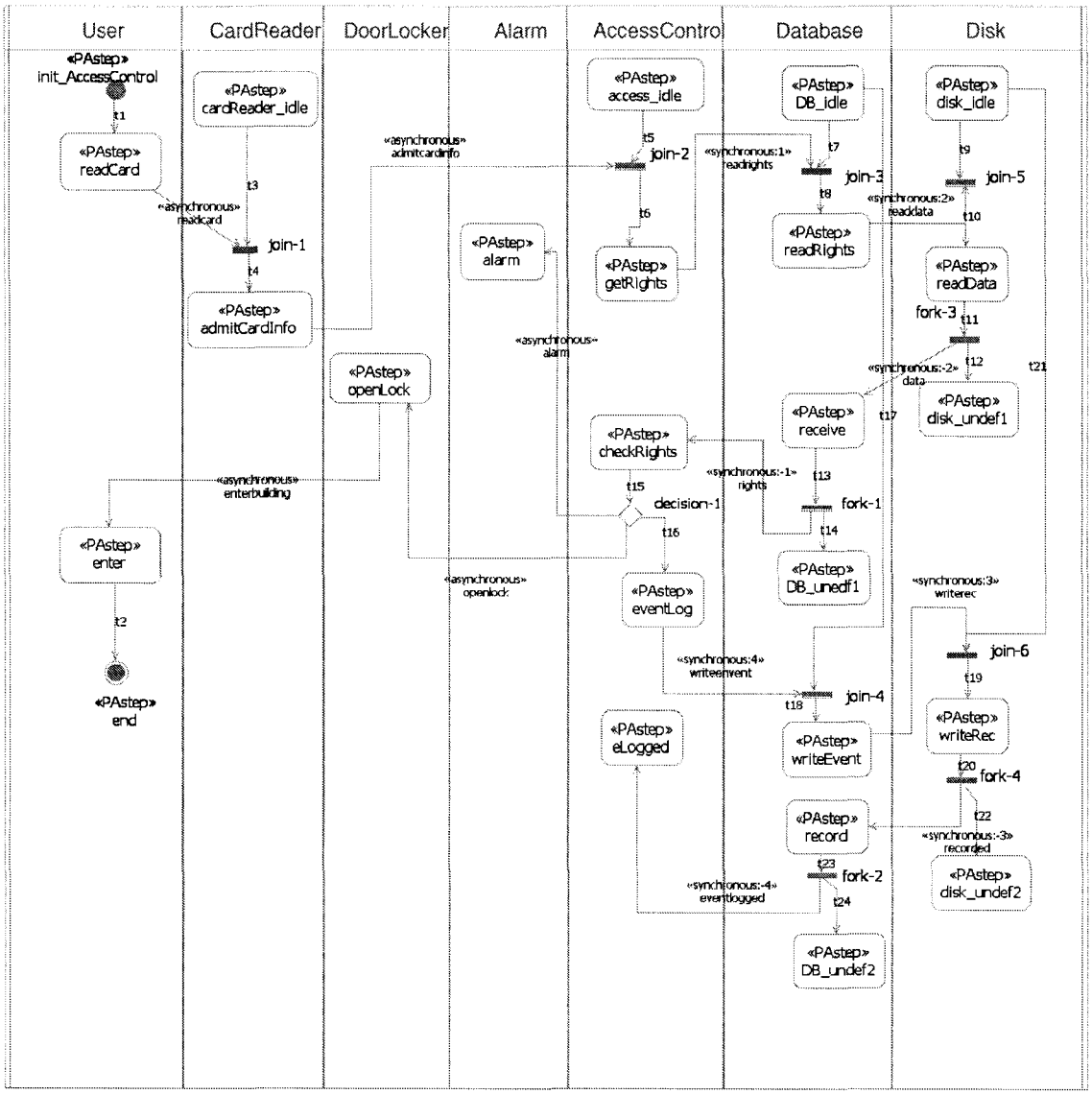


Figure 6.10: Activity diagram of BSS Access Control scenario.

Figure 6.8 shows the Nodes deployment of the Building Security System, including an application processor, a database processor, and other peripheral devices accessed over a LAN. The Node-Artifact-Component relation for all Nodes is not shown since they are basically one-to-one relations except for AppCPU. The deployment relation for AppCPU is given in Figures 6.9(a) and (b), where two components, Video Acquisition and Access Control, are shown. The former is a composite component that consists of five other components including VideoController, AcquireProc and StoreProc etc. All the devices and processors are stereotyped as <<PAresource> and <<PAhost>>, and components are stereotyped as <<PAresource>>.

Figure 6.10 describes the Access Control scenario using stereotypes and tagged values defined in the SPT Profile [78]. These stereotypes are, respectively, <<PAcontext>>, <<PAstep>>, <<PAopenLoad>> and <<PAclosedLoad>> for workloads.

As shown in Figure 6.10, the User provides an open workload, meaning a given arrival process. The tagged values define it as a Poisson process with a mean inter arrival time. The steps are tagged with a demand value for processing time (tag PAdemand), which is the CPU demand for the step. The request goes from the card reader to the Access Controller software task, to the database and its disk, and then back to execute the check logic and either allow the entry or not. The openDoor is a conditional step, which can be tagged with a probability (PAprob) which here is set to 1. The devices are stereotyped as <<PAresource>>, as in the deployment diagram, and so are the software tasks Access Controller and Database; this is because a task has a queue and acts as a server to its messages. A resource can be tagged as having multiple copies, as in a multiprocessor or a multithreaded task.

The scenario for the Acquire/Store Video is shown in Figure 6.11. There is a single Video Controller task that commands the acquisition of video frames from n cameras in turn, by a process AcquireProc. The initial step is the Video Controller which is a closed workload source with one instance, with a required cycle time having 95% of cycles below 1 second (Figure 6.11(a)), and a predicted value $N cycles to represent the model result. In Figure 6.11(b), the AcquireProc is a concurrent process (<<PAresource>>). It acquires a Buffer resource by a step allocBuf which is stereotyped as <<GRMacquire>>, indicating a resource acquisition. Buffer is a passive resource shown in the deployment diagram with a multiplicity $Nbuf, managed by BufManager. In the activity diagram the use of Buffer is indicated by the stereotype <<GRMacquire>>. In the base case, $Nbuf is set to 1. Once a buffer is acquired, AcquireProc requests the image from the camera, receives it and passes the full buffer to a separate process StoreProc, which stores the frame in the database and releases the buffer. The Database process can be tagged with the number of threads, and its disk subsystem is tagged as having two disks. The writeImg operation on the Database has a tag PAextOp to indicate the time demand for

an operation writeBlock, which is not defined in the diagram. This operation can be filled in, in the performance model, by a suitable operation to write one block of data to disk.

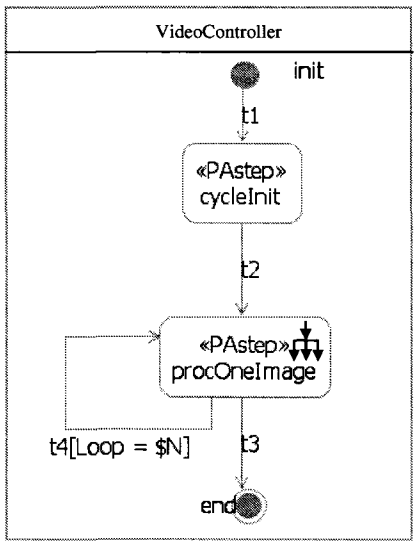


Figure 6.11 (a) Top-level activity diagram for Acquire/Store Video scenario.

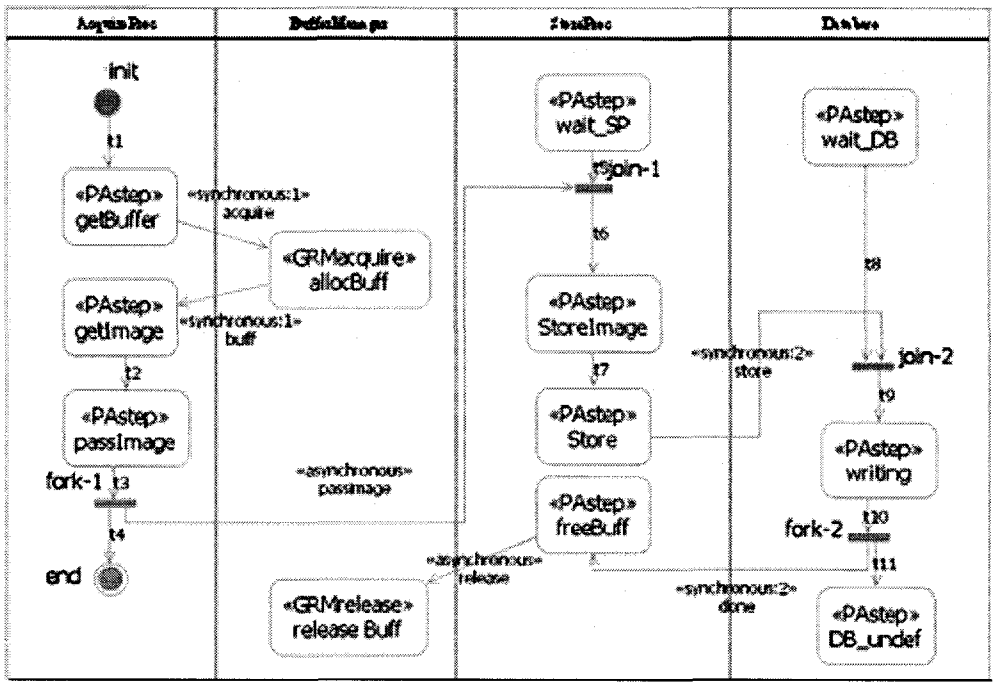


Figure 6.11 (b) Composed activity procOneImage.

Transformation to Performance Models:

After the construction of a performance annotated UML design model, the transformation of the BSS example is done via the same steps as described previously. In constructing

the intermediate model, each scenario expressed in UML activity diagram is processed separately. The Access Control and Acquire/Store Video are difference scenarios and they are transformed to two "IM:ServiceEntity" trees. However, the composite step "procOneImage" (Figure 6.11(a)) is treated as a difference "IM:Service" and it has a reference to the sub-scenario, in this case, which is the Acquire/Store Video scenario.

To LQN:

Access Control Scenario

Figure 6.12 shows a high-level aggregation of steps of the LQN model for the Access Control Scenario, where the shaded areas are either phases or activities. The transformation is done partition by partition; each partition is converted to a LQN task and service. Service that does not contain a "fork" operation is transformed to phase. Only AccessControl is mapped to LQN activities in this scenario since it has an "or-fork" operation.

There are five pairs of blocking synchronous calls (one in User, two in Access Control and Database each). They are handled as the synchronous messaging example given in Figure 6.1 earlier. The number of entries in each task is the number of services they offer (for which requests are received). In this example, Database and Disk tasks have two entries corresponding to "read" and "write" operations.

A server task can carry out part of its work after replying to its client; this is termed a "second phase" of service. For each entry the host demand is represented by first and second phase CPU demand in time units. A request arc in the model can have a mean number of calls per entry invocation, or a deterministic integer number. Here Most of the calls are given as averages.

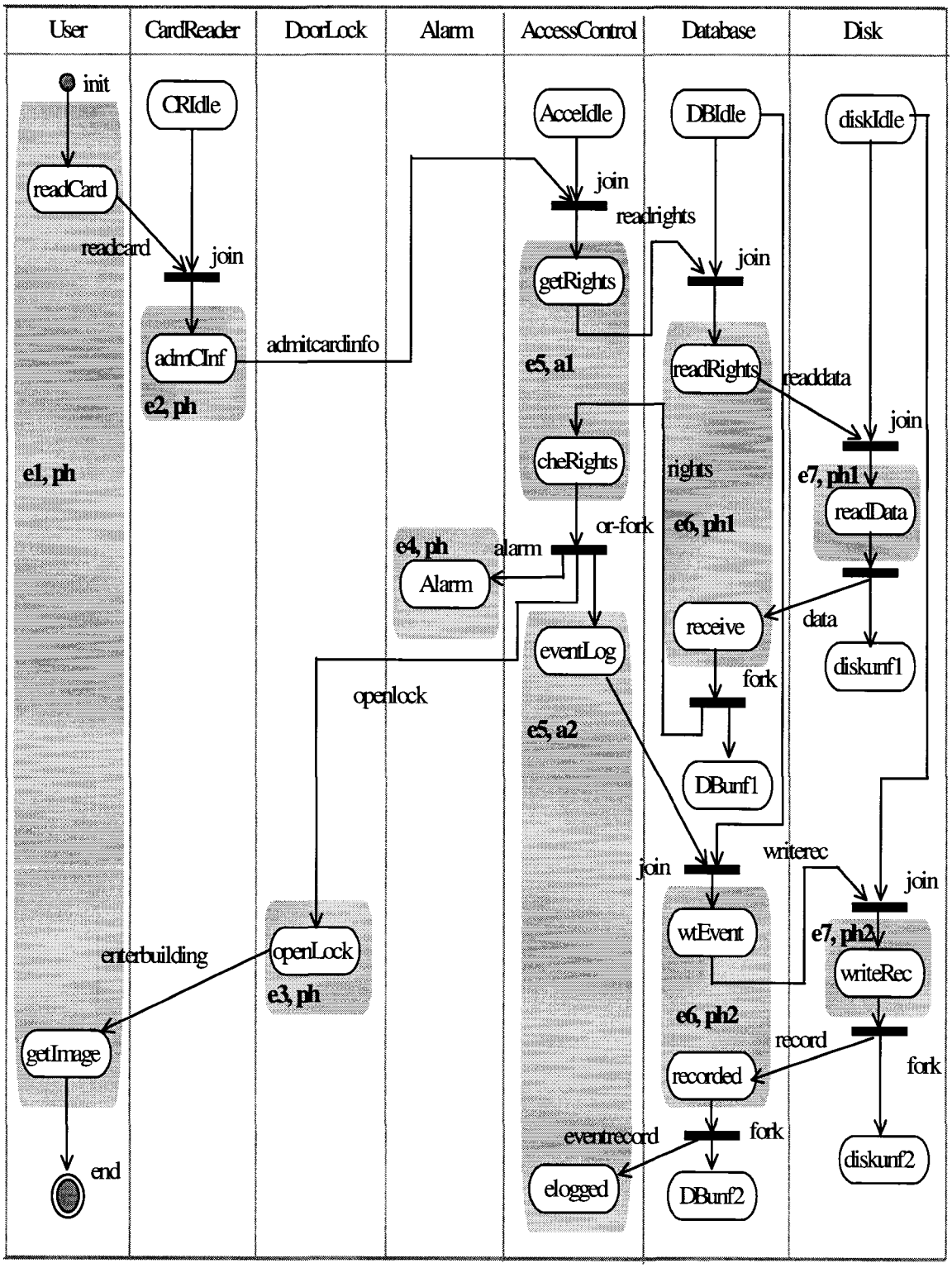


Figure 6.12 Assignment of UML activities for the Access Control Scenario to LQN entries, phases and activities.

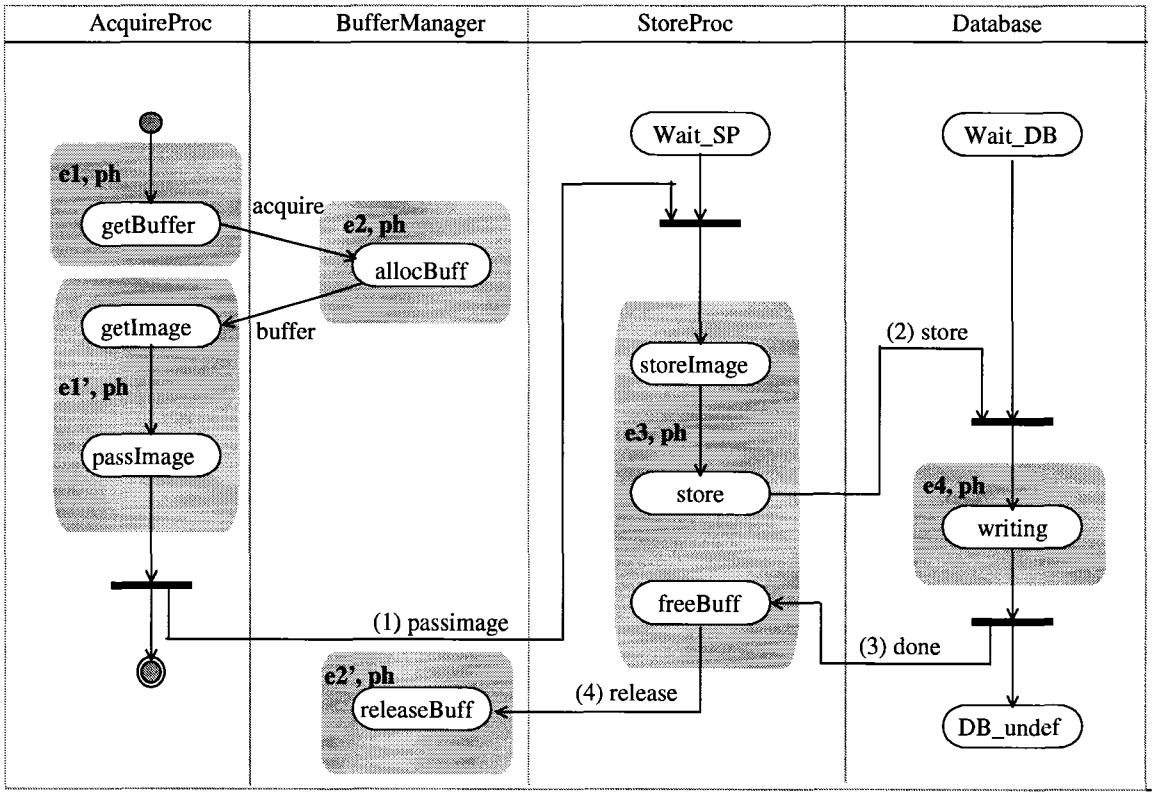


Figure 6.13: Assignment of UML activities for the Video Scenario to LQN entries and phases.

Acquire/Store Video Scenario

This scenario is more complicated than the Access Control scenario despite of having fewer steps (Figure 6.13). This scenario is actually a sub-scenario that is aggregated in the main scenario shown in Figure 6.11(a). We have discussed how to handle composite scenarios earlier. The difficulty here is to address the use of the Buffer resource via the Buffer Manager. This case is different from the buffer example that was discussed previously because the release of the buffer does not happen until the completion of the disk writing. Since the Buffer entity is implied (even if not shown in the activity diagram). Access of the buffer is through a 'bufEntry', which makes synchronous calls to invoke the operations and carries out holding the buffer. Although the activity diagram shows that these operations are in the same AcquireProc task (e1) in the software, they are separated in the model into a nested dummy task (e1'), which executes while AcquireProc is blocked. Passing the buffer to Store is also modeled as a call from the dummy task (e1') to the Store task. The Store task, at the end, calls the BufferManager

task to release the buffer; however, to avoid another calling cycle in the model, the release is again modeled as an entry of a dummy task e2'.

Combining the Two Scenarios

In the transformation process, each scenario is converted to an "IM:ServiceEntity" tree. If the same task is involved in different scenarios, the entries of that task will be grouped in IM→LQN. For example, the Database task is involved in both Access Control and Acquire/Store Video scenario and it has three entries (two in the former and one in the latter).

To CSIM Model:

Different from LQN, the CSIM model keeps most of the execution details that correspond to service time executions, and processes interaction that are derived from scenario.

Access Control Scenario

Figure 6.14 is the graphical execution diagram of the Access Control Scenario in CSIM model. Each "IM:Service" becomes a "CSIM:Process" (represented by a circle inside a rectangle). The "CSIM:Process" is linked to a hardware (a processor or a device) and software resource (facility). The inter-process communications are done via MBS and MBR. The execution flow of the Access Control scenario in CSIM starts with the User making a request to CardReader, which passes the request to AccessControl. To verify the authority, AccessControl checks the Database and Disk. Depending on the return, Access Control notifies alarm or DoorLock. This scenario has seven "CSIM:Process" sub-trees and each contains a service except for Database and Disk, which have two services (corresponding to read and write operations). The multiple services in a process is treated as a special case in current version of the code implementation and does not handled in the XML CSIM output; however, it is taken care of using "switch" in XML CSIM model to C/C^{++} code conversion.

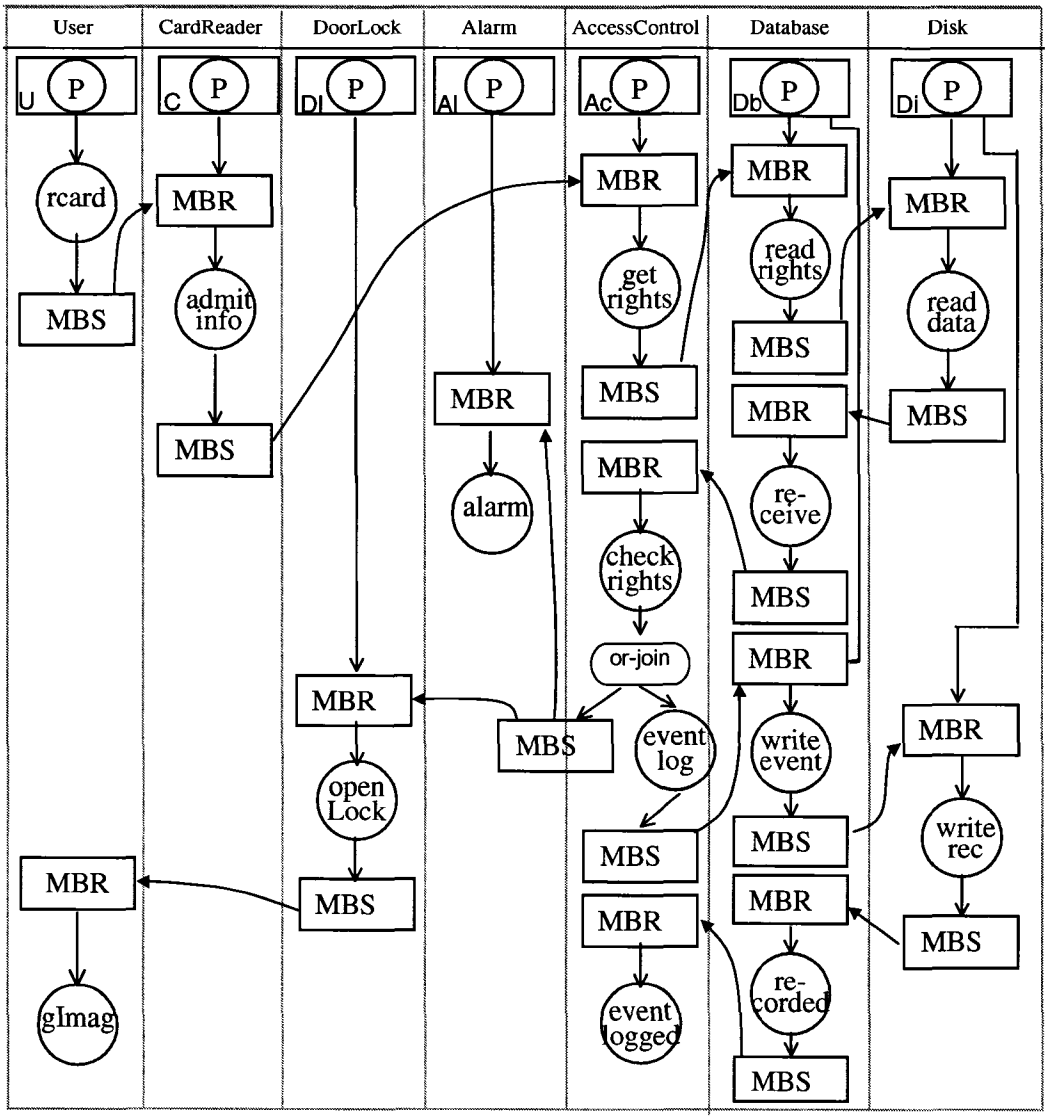


Figure 6.14: CSIM execution model for Access Control Scenario.

Acquire/Store Video Scenario

The high-level CSIM model for Acquire/Store Video Scenario is displayed in Figure 6.15. The AcquireProc ("CSIM:Process") acquires a buffer from BufferManager, which allocates a Buffer, and returns it to the AcquireProc. Once it has obtained the Buffer, AcquireProc makes a call (using MBS and MBR operations) to StoreProc and further to the Database. The Buffer is occupied until the StoreProc releases it to the BufferManager. In this scenario, the AcquireProc is free as soon as it makes the "MBS" to store image but the BufferManager is held until the image is stored. The CSIM "MBS" does not block the

process but the "MBR" does. Once the CSIM XML model is available, it can be converted to a compile-ready C/C^{++} code that can be run in CSIM18 engine.

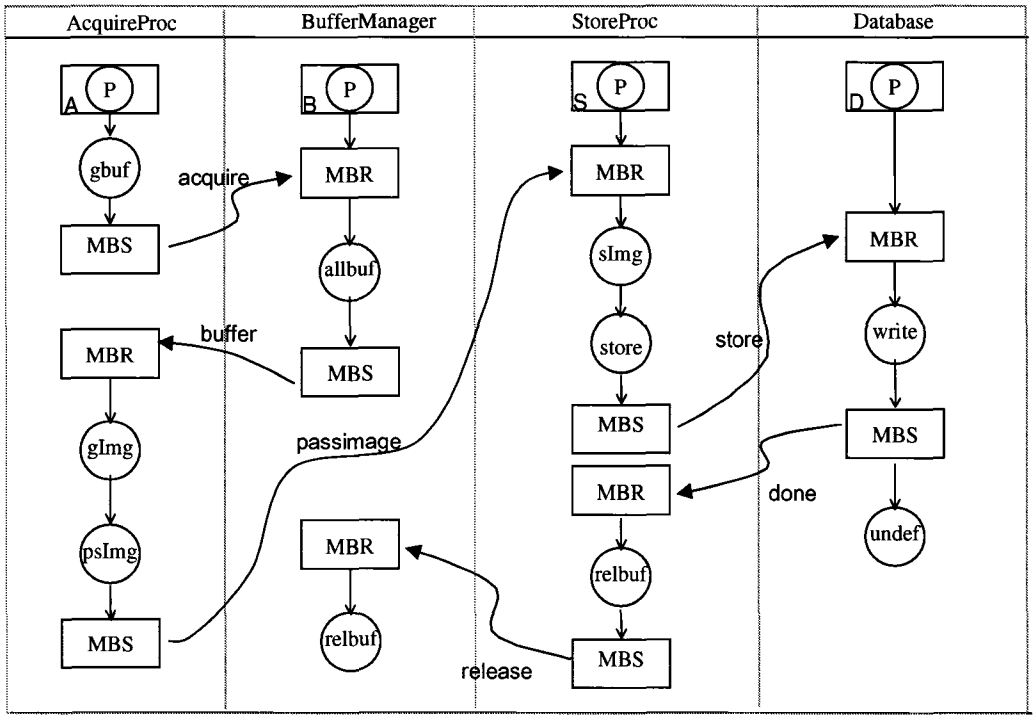


Figure 6.15: CSIM execution model for Acquire/Store Video Scenario.

Both LQN and CSIM XML outputs for Building Security System are rather complex and lengthy. A report containing XSLT style sheets, XML outputs and debugging results, particularly for the BBS test case, are provided as an additional document for this research work [42].

6.11 SUMMARY

In this chapter, we discussed the test results of six simple model fragments representing different scenarios and two end-to-end models to verify the correctness of the transformation. Behavior scenarios include, synchronous and asynchronous messaging, optional and parallel combinations, loop and use of resource. Two end-to-end models were also tested, including the 3-tier Client Server Application and Building Security System Application. All the outputs are presented in the Appendix IV of this thesis.

CHAPTER 7: CONCLUSION

7.1 CURRENT WORK STATUS

The main contribution of the thesis is a model transformation method that combines concepts from graph grammars (or graph rewriting) and XML transformation techniques, such as XMLgebra. More precisely, the proposed method contains a lower abstraction level for XML tree manipulations, and a higher abstraction level for defining the mapping between the source and target model, based on graph-grammar concepts. According to the taxonomy of model transformations from [70], the transformation proposed in this thesis is both exogenous (i.e., the source and target models are different) and vertical (abstraction-raising).

The proposed transformation fits very well within the context of Model Driven Architecture (MDA), the new vision for software development proposed by OMG. MDA promotes the idea that software development should be based on models throughout the entire software lifecycle. In the context of MDA, it becomes important to have the ability to evaluate at an early stage the performance characteristics of UML models, in order ensure that the systems under development will meet their performance requirements. The proposed transformation method was applied to generate performance models from UML software design models annotated with performance information according to the standard UML Profile for Schedulability, Performance and Time (SPT). The transformation is conducted in two steps: first a transformation from UML+SPT (in XML format) to an Intermediate Model, and then from the intermediate model to a performance model (either LQN, or CSIM simulation-based model). The role of the Intermediate Model is to allow for a more flexible transformation method, which is easy to specialize for several kinds of source models (such as UML 1.4 and UML 2.0) and different kinds of target models (queuing networks, Petri nets, simulation, etc.) The approach of introducing an Intermediate Model to solve this type of N-to-M transformation problem is similar to the solution proposed in the PUMA project [122], and was developed in parallel with it.

The proposed transformation method includes the following parts:

a) A set of model transformation rules for mapping the source model to the target model, defined by using the eXMLgebra notation. Each rule maps a template compliant with the input DTD/schema (the left-hand-side) to a template compliant with the output DTD/schema (the right-hand-side). The rules define not only the structure of the subtrees to be generated, but also how to compute the node attributes. Some of these attributes will be used later as "gluing" labels.

b) An eXMLgebra-based algorithm that traverses a given input XML template (tree) representing the source model, and chooses and applies the appropriate transformation rules. This step generates a set of XML subtrees compliant with the output DTD. If necessary, this algorithm also implements the abstraction-raising aggregation of source model elements.

c) An eXMLgebra-based algorithm to construct the complete output template (tree) by gluing the subtrees obtained in the previous step. The final result represents the generated output model.

One of the most interesting features of the proposed transformation is the "abstraction-raising" feature, needed to bridge the large semantic gap between the source and target models, where the target model is at a higher level of abstraction than the source. For instance, the QVT/Relations language described in the latest proposal for the MOF QVT standard [80] cannot easily define complex mappings as those required in an abstraction-raising transformation, where a group of source model elements (whose number and relationships varies widely from case to case) must be aggregated together according to some given aggregation rules applied repeatedly, then mapped to a single target model element.

The proposed transformation was implemented in the XML technological space. More exactly, the rules (a) and the algorithms (b) and (c) were implemented in XSLT, as well as the library of eXMLgebra functions, as discussed in chapter 5. One of the advantages of implementing the transformation at the XML level (where the XML format is obtained according to the XMI standard) is the fact that no libraries for manipulating the source and target metamodels are necessary. During the implementation, a lot of attention was

given to reusability, separating the parts that are independent of the source and target models from the parts that are dependent on them.

The thesis also discussed issues regarding to the testing of the implementation by using different scenarios that include, synchronous and asynchronous messaging, optional and parallel combination, loop and use of resource. Two complete models were also tested, the 3-tier Client Server Application and Building Security System.

7.2 FUTURE WORK

Although the proof of concept of the XSLT-based transformation from a UML model to a performance model has been successful, there are still challenges and difficulties to be overcome before the proposed process is applied to a complicated UML design. For example, errors in UML-XML file generation due to deficiencies of the current XMI tools, duplicate information generated from different diagrams and merging multiple scenarios will certainly bring in additional complication to the transformation process.

Future work in this direction includes refinement of the transformation process, particularly, using more complicated application example. Details include:

a) To extent the kind of UML models that can be transformed including more architectural pattern recognition and transformation. Patterns considered so far only included simple client/server, pipeline and filters, and buffer resource patterns; more patterns can be included in the future.

b) To develop a more general approach for defining abstraction-raising aggregation rules that can be invoked by the transformation algorithm. In the proposed solution, the abstraction-raising aggregation of steps was hard-coded in the select-apply function. The goal is to separate even more in the implementation the reusable parts from the ones that are dependent on the source or target models.

c) To generate different kinds of performance models to make sure that the proposed transformation method is flexible and easy to specialize for other performance models. The goal is to concentrate the performance model specific details in clearly identified parts, separated from the generic parts. In this study, LQN and CSIM simulation models were the primary focus, but other target models could be explored.

The aim of this work is to develop a generic model transformation method for direct UML design to multiple performance models transformation that enables the early performance evaluation of software design in early stages. The ultimate goal for this work is to further reduce the gap that exists between the software development and the performance analysis domains.

REFERENCES:

1. Ajmone, M., Balbo, G., Conte, G., "Performance Models of Multiprocessor Performance", The MIT Press, 1986.
2. Amer, H. "Automated Transformation of UML Software Specifications into LQN Performance Models using Graph-Grammar Techniques", M. Eng. Thesis, Carleton University, 2001.
3. Amer, H. and Petriu, D.C., "Software Performance Evaluation: Graph Grammar-based Transformation of UML Design Models into Performance Models", submitted for publication.
4. Ammar, H.H., Cortellessa, V., Ibrahim, A., "Modeling resources in a UML-based simulative environment", http://www.csee.wvu.edu/~ammar/papers/2001/camera_ready1.pdf .
5. Amyot, D. "Use Case Maps as a Feature Description Language", in: S. Gilmore and M. Ryan (Eds), Language Constructs for Designing Features. Springer-Verlag. 27-44, 2000.
6. Amyot, D. and Andrade, R. "Description of Wireless Intelligent Network Services with Use Case Maps", in: SBRC99, 17° Simpósio Brasileiro de Redes de Computadores, Salvador, Brazil, May 1999, 418-433.
7. Amyot, D. and Mussbacher, G. "On the Extension of UML with Use Case Maps Concepts", in: <<UML>>2000, 3rd International Conference on the Unified Modeling Language, York, UK, October 2000. LNCS 1939, 16-31.
8. Amyot, D., Buhr, R.J.A., Gray, T., and Logrippo, L. "Use Case Maps for the Capture and Validation of Distributed Systems Requirements", in: RE'99, Fourth IEEE International Symposium on Requirements Engineering, Limerick, Ireland, June 1999, 44-53.
9. Andolfi, F., Aquilani, F., Balsamo, S., Inverardi, P., "Deriving Performance Models of Software Architectures from Message Sequence Charts", Workshop on Software and Performance, Proceedings of the second international workshop on Software and performance, WOSP2000, Ottawa, Ontario, Canada, pp 47-57, 2000.
10. Aquilani, F., Balsamo, S., Inverardi, P., "Performance Analysis at the software architecture design level", Performance Evaluation, Vol. 45, No.4, pp. 205-221, 2001.
11. ArgoUML: An Open-Source UML-Tool, See http://argouml.tigris.org/.
12. Arief, L.B., and Speirs, N.A., "A UML Tool for an Automatic Generation of Simulation Programs" in Proc.WOSP'2000, pp. 71-76.
13. Baccelli, F., Balbo, G., Boucherie, R.J., Campos, J., Chiola, G., "Annotated bibliography on stochastic Petri nets". In Performance Evaluation of Parallel and

Distributed Systems - Solution Methods, CWI Tract, 105, pp. 1-24, Amsterdam, 1994.

14. Balsamo, S. and Simeoni, M., "Deriving Performance Models from Software Architecture Specifications", Proceedings of the 15th European Simulation Multi-conference (ESM2001), SCS - Society for Computer Simulation, 2001.

15. Balsamo, S., Grosso, M. and Marzolla, M., "Towards Simulation-Based Performance Modeling of UML specifications", Technical Report CS-2003-2 pp1-22. http://www.dsi.unive.it/~marzolla/papers/tr-cs-2003-2.pdf.

16. Balsamo, S., Inverardi, P., Mangano, C., "An Approach to Performance Evaluation of Software Architectures" Workshop on Software and Performance, WOSP'98, Santa Fe, New Mexico, Oct 12-16, 1998.

17. Balsamo, S., Marco, A.D., Inverardi, P. and Simeoni, M., "Software Performance: state of the art and perspectives", Technical report, Sahara project, Universit` a dell'Aquila, http://di.univaq.it/~inverard/papers/techReport-perf.pdf .

18. Balsamo, S., Personè, V.D.N. and Inverardi, P., "A review on queuing network models with finite capacity queues for software architectures performance prediction", Performance Evaluation 974 (2002) 1-20.

19. Bernardi, S., Donatelli, S. and Merseguer, J., "From UML Sequence Diagrams and State charts to analyzable Petri Net models", Workshop on Software and Performance, Proceedings of the third international workshop on Software and performance, Rome, Italy pp35 - 45, 2002.

20. Bernardo, M., Ciancarini, P., Donatiello, L., "AEMPA: A Process Algebraic Description Language for the Performance Analysis of Software Architectures" in Proc. of the 2nd Int. Workshop on Software and Performance (WOSP 2000), ACM Press, pp. 1-11, Ottawa (Canada), September 2000.

21. Blaha, M. and Premerlani, W., A Catalog of Object Model Transformations; Presented at 3rd Working Conference on Reverse Engineering, Monterey, California; November 1996.

22. Bolch, G., Greiner, S., Meer, H., Trivedi, K.S., "Queuing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications", John Wiley and Sons, August 1998.

23. Booch, G., Rumbaugh, J. & Jacobson, I., The Unified modeling language user guide, Reading, Mass., Addison-Wesley. 1999.

24. Bray, T., et al, "eXtensible Markup Language (XML)" version 1.0, W3C, 10 Feb 1998, See http://www.w3c.org/TR/REC-xml.

25. Buhr, R.J.A. and Casselman, R.S., "Use Case Maps for Object-Oriented Systems", published by Prentice Hall, 1996.

26. C++SIM, http://cxxsim.ncl.ac.uk/.

27. Cassidy, T., Cordy, J., Dean, T., Dingel, J., "Source Transformation for Concurrency Analysis". Fifth Workshop on Language Descriptions, Tools and Applications (LDTA 2005), Edinburgh, Scotland. April 2005.

28. Christensen, A.S., Kirkegaard, C. and Møller, A., "A Runtime System for XML Transformations in Java", In Proceedings of the Second International XML Database Symposium (XSym 2004).

29. Christensen, A.S., Møller, A., Schwartzbach, M.I., "Static Analysis for Dynamic XML", http://www.brics.dk/~mis/papers.html.

30. Cortellessa, V. and Mirandola, R., "Deriving a Queuing Network based Performance Model from UML Diagrams" in Proc. of WOSP'2000, pp. 58-70, 2000.

31. Cortellessa, V., Singh, H. and Cukic, B., "Early reliability assessment of UML based software models", Workshop on Software and Performance, Proceedings of the third international workshop on Software and performance, Rome, Italy, pp302-309, 2002.

32. CSIM, http://www.atl.external.lmco.com/proj/csim/.

33. CSIM18, Mesquite Software, Inc. http://www.mesquite.com.

34. Czarnecki, K. and Helsen, S., "Classification of Model Transformation Approaches". In online proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA. Anaheim, October, 2003

35. Czarnecki, K., "Overview of Generative Software Development", In J.-P. Banâtre et al. (Eds.): Unconventional Programming Paradigms (UPP) 2004, Mont Saint-Michel, France, LNCS 3566, pp. 313–328, 2005

36. D'Ambrogio, A., "A model transformation framework for the automated building of performance models from UML models", Software and Performance Proceedings of the 5th international workshop on Software and performance, Palma, Illes Balears, Spain, Pages: 75 - 86, 2005.

37. Enterprise Architect (EA), http://www.sparxsystems.com/ea.htm.

38. Franks, G., Hubbard, A., Majumdar, S., Petriu, D.C., Rolia, J., Woodside, C.M., "A toolset for Performance Engineering and Software Design of Client-Server Systems". Performance Evaluation, Vol. 24, No. 1-2, pp. 117-135, 1995.

39. Franks. G., Performance Analysis of Distributed Server Systems, PhD thesis, Department of Systems and Computer Engineering, Carleton University, 1999.

40. Gamma, E., Helm, R., Johnson, R. and Vlissides, J., "Design Patterns", Addison Wesley Professional Computing Series, Addison-Wesley, 1994.

41. Gomaa, H. and Menascè, D.A., "Design and Performance Modeling of Component Interconnection Patterns for Distributed Software Architectures", in Proc. of WOSP'2000, pp. 117-126.

42. Gu, G.P. and Petriu, D.C., "Implementation of UML To Performance Models by XSLT Transformations", Research Report, Carleton University, 2006.

43. Gu, G.P. and Petriu, D.C., "Early evaluation of Software Performance based on the UML Performance Profile, Proc. of CASCON'2003, pp.214-227, Toronto, ON, Canada, October 2003.

44. Gu, G.P. and Petriu, D.C., "Software Performance Evaluation Based on the UML Performance Profile", Proc. of 7th World Multi-Conference on Systemics, Cybernetics and Informatics SCI'2003, pp.249-254, Orlando, Florida, USA, July 27-30, 2003.

45. Gu, G.P. and Petriu, D.C., "XSLT Transformation from UML Models to LQN Performance Models", Proc. of 3rd Int. Workshop on Software and Performance WOSP'2002, pp.227-234, Rome, Italy, 2002.

46. Harold, E.R., "XML Bible", http://www.mmg.vmei.acad.bg/xml/.

47. Harrison, P.G., Hillston, J., "Exploiting Quasi-Reversible Structures in Markovian Process Algebra Models", Computer Journal, Vol.38, No.7, pp. 510-520, 1995.

48. Hermanns, H., Herzog, U., Katoen, J.P., "Process Algebra for Performance Evaluation", Theoretical Computer Science, Vol 274 (1-2), pp. 43-87, 2002.

49. Hillston, J. and Thomas, N., "Product Form Solution for a Class of PEPA Models", Performance Evaluation, Vol.35, No.3, pp. 171-192, 1999.

50. Hoeben, F., "Using UML Models for Performance Calculation", Workshop on Software and Performance, Proceedings of the second international workshop on Software and performance, WOSP2000, Ottawa, Ontario, Canada, pp. 77-82.

51. IBM Rational Software Architect (RSA), http://www306.ibm.com/software/rational/.

52. Jacobson, I., Booch, G. & Rumbaugh, J., The Unified Software Development Process, MA: Addison Wesley Longman Inc., 1998.

53. Jacobson, I., The road to the unified software development process. Rev. and updated by Stefan Bylund. Cambridge, UK, Cambridge University Press, 2000.

54. JSIM: A Java-Based Simulation and Animation Environment, http://chief.cs.uga.edu/~jam/jsim/.

55. Kähkipuro, P., "Performance Modeling Framework for CORBA Based Distributed Systems", Department of Computer Science, Series of Publications A, Report A-2000-3.

56. Kähkipuro, P., "UML based performance modeling of object-oriented distributed systems", Proc. of Second International Conference on the Unified Modeling Language, October 28-30, 1999, USA, LNCS, Springer Verlag, vol.1723, 1999.

57. Kähkipuro, P., "UML-based Performance Modeling Framework for Component-Based Distributed Systems", Proceedings of Performance Engineering, Springer LNCS 2047, pp. 167-184, 2001.

58. Kant, K., "Introduction to Computer System Performance Evaluation", McGraw-Hill, 1992.

59. Kay, M., SAXON XSLT processor, http://saxon.sourceforge.net

60. King, P. and Pooley, R., "Derivation of Petri Net Performance Models from UML Specifications of Communication Software" Proc. of XV UK Performance Engineering Workshop, 1999.

61. Kirkegaard, C., "Dynamic XML Processing with Static Validation (Masters Thesis), University of Aarhus, 2003.

62. Kirkegaard, C., Møller A. and Schwartzbach, M.I., "Static Validation of XML Transformations in Java", In IEEE Transactions on Software Engineering, 30(3), March 2004 (TSE).

63. Kleinrock, L., "Queuing Systems. Vol. 1: Theory", John Wiley and Sons, 1975.

64. Kleppe, A., Warmer, J., Bast., W., "MDA Explained, The Model-Driven Architecture: Practice and Promise". Addison Wesley (2003).

65. Lazowska, E.D., Zahorjan, J., Scott Graham, G. S., Sevcik, K.C., "Quantitative System Performance: Computer System Analysis Using Queuing Network Models", Prentice-Hall, Englewood Cliffs, 1984.

66. Mazzocca, N., Rak, M. and U. Villano, U., "The MetaPL approach to the performance analysis of distributed software systems", Workshop on Software and Performance, Proceedings of the third international workshop on Software and performance, Rome, Italy, pp142-149, 2002.

67. Menascè, D.A., "A Framework for Software Performance Engineering of Client/Server Systems" Proc. of the 1997 Computer Measurement Group Conference, Orlando, Florida (1997). http://cs.gmu.edu/~menasce/papers/cmg97-menasce-spe.pdf .

68. Menascè, D.A., and Gomaa, H., "On a Language Based Method for Software Performance Engineering of Client/Server Systems" Proc. of WOSP'98, Santa Fe, New Mexico, USA, pp. 63-69 (1998).

69. Menascè, D.A.and Gomaa, H., "A Method for design and Performance Modeling of Client/Server Systems", IEEE Transaction on Software Engineering, Vol. 26, No. 11, pp. 1066-1085, 2000.

70. Mens, T., Czarnecki, K., and Gorp, P.V., "A Taxonomy of Model Transformations", http://drops.dagstuhl.de/opus/volltexte/2005/11/pdf/04101.SWM2.Paper.pdf

71. Mens, T., Gorp, V.P., Varró, D., Karsai, G., "Applying a Model Transformation Taxonomy to Graph Transformation Technology", Electr. Notes Theor. Comput. Sci. 152: 143-159 (2006), http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2005/gramot05_mvkv.pdf

72. Mens. T., Graph transformation support for software refactoring [Conf. Slides, June 2004]. http://w3.umh.ac.be/~infofs/preprints/index.php?page=list.

73. Miga, A., Amyot, D., Bordeleau, F., Cameron, C. and Woodside, M. (2001) Deriving Message Sequence Charts from Use Case Maps Scenario Specifications. In: Tenth SDL Forum (SDL'01), Copenhagen, Denmark, June 2001. LNCS 2078, 268-287.

74. Miguel, M.D., Lambolais, T., Hannouz, M., Betgè-Brezetz, S., Piekarec, S., "UML Extensions for the Specification and Evaluation of Latency Constraints in Architectural Models" in [WOSP2000] pp. 83-88.

75. Mussbacher, G. and Amyot, D. "A Collection of Patterns for Use Case Maps", in: First Latin American Conference on Pattern Languages of Programming, Rio de Janeiro, Brazil, October 2001

76. Neilson, J.E., Woodside, C.M., Petriu D.C., Majumdar, S., "Software Bottlenecking in Client-Server Systems and Rendez-vous Networks", IEEE Trans. On Software Engineering, Vol. 21, No. 9, pp. 776-782, Sept. 1995.

77. Nickel, U.A. and Wagner, R., "Graph-Grammar Based Completion and Transformation of SDL/UML-Diagrams", Position Paper, http://ase.arc.nasa.gov/wtuml01/submissions/nickel-wagner.pdf .

78. Object Management Group, "UML Profile for Schedulability, Performance, and Time Specification", OMG Adopted Specification ptc/02-03-02, July 1, 2002.

79. OMG: MDA Guide Version 1.0.1, http://www.omg.org/docs/omg/03-06-01.pdf

80. OMG: Revised Submission for MOF 2.0 Query/View/Transformations RFP, QVT-Merge Group, ver. 1.0, April 2004.

81. OMG: Unified Modeling Language Specification, version 2.0, http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML

82. OMG: Unified Modeling Language Specification, version 1.4, http://www.omg.org/cgi-bin/doc?formal/01-09-67 , Sep 2001.

83. OMG: XML Metadata Interchange specification, version 2.0, http://www.omg.org/cgi-bin/doc?formal/2003-11-04 , 2003.

84. Petriu, D. and Woodside, C.M., "A Metamodel for Generating Performance Models from UML Designs, Proc UML 2004, Lisbon, v. 3273 of Lecture Notes in Computer Science (LNCS 3273), pp 41-53, Oct 2004.

85. Petriu, D. Woodside, C.M., (2001), Generating a Performance Model from a Design Specification. http://www.usecasemaps.org/pub/index.shtml .

86. Petriu, D., "Layered Software Performance Models Constructed from Use Case Map Specifications", M. Eng thesis, Department of System and Computer Science, Carleton University, May, 2001.

87. Petriu, D., and Woodside, C.M., "Analyzing Software Requirements Specifications for Performance", Proc. Third Int. Workshop on Software and Performance, Rome, July 2002.

88. Petriu, D., and Woodside, C.M., "Software Performance Models from System Scenarios in Use Case Maps", Proc. 12 Int. Conf. on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation (Performance TOOLS 2002), London, April 2002.

89. Petriu, D.B. and Woodside, C.M.,"Software Performance Models from System Scenarios", Report SCE-03-06, Dept. of Systems and Computer Engineering, Carleton University, Oct. 2002.

90. Petriu, D.C. and Shen, H., "Applying the UML Performance Profile: Graph Grammar based derivation of LQN models from UML specifications", in Computer Performance Evaluation: Modeling Techniques and Tools, (T. Fields, P. Harrison, J. Bradley, U. Harder, Eds.) Lecture Notes in Computer Science 2324, pp.159-177, Springer, 2002.

91. Petriu, D.C., "Deriving Performance Models from UML Models by Graph Transformations" Tutorial in [WOSP2000].

92. Petriu, D.C., Amer, H., Majumdar, S., Fatah, I.A., "Using Analytic Models for Predicting Middleware Performance", Proceedings of the Second International Workshop on Software and Performance, Ottawa, Canada, pp 189-194, Sept. 2000.

93. Petriu, D.C., Shousha, C. and Jalnapurkar, A., "Architecture-Based Performance Analysis Applied to a Telecommunication System", IEEE Transactions on Software Engineering, Volume 26 , Issue 11 (November), pp1049-1065, 2000.

94. Petriu, D.C., Wang, X., "Deriving Software Performance Models from Architectural Patterns by Graph Transformations", in Theory and Applications of Graph Transformations, TAGT'98 (H.Ehrig, G.Engels, H.J. Kreowski, G. Rozenberg, Eds.) Lecture Notes in Computer Science 1764, pp.475-488, Springer Verlag, 2000.

95. Petriu, D.C., Wang, X., "From UML descriptions of High-Level Software Architectures to LQN Performance Models" in Proc. of AGTIVE'99, Springer Verlag LNCS 1779, pp. 47-62 (1999).

96. Petriu, D.C., Woodside, C.M., "Performance Analysis with UML", Chapter in UML for Real: Design of Embedded Real-Time Systems (L. Lavagno, G. Martin and B.Selic, Eds.), ISBN 1-4020-7501-4, Kluwer Academic Publishers, 2003.

97. Pooley, R., and King, P., "The Unified Modeling Language and Performance Engineering". Proceedings of IEE Software, pp. 2-10, 1999.

98. Pooley. R., "Using UML to Derive Stochastic Process Algebra Models" Proc. of XV UK Performance Engineering Workshop (1999). http://www.cs.bris.ac.uk/Events/UKPEW1999/proceedings/POOL.ps.gz

99. Poseidon UML, http://www.gentleware.com/ .

100. Rational Rose Win Evaluation, http://www-306.ibm.com/software/rational/ .

101. Rolia, J.A., and Sevcik, K.C., "The Method of Layers", IEEE Transaction on Software Engineering, Vol. 21/8, pp. 682-688, 1995.

102. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W., Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs.1991.

103. Rumbaugh, J., Jacobson, I. & Booch, G., The Unified modeling language reference manual, Reading Mass., Addison-Wesley. 1999.

104. Sabetta, A., Petriu, D.C., Grassi, V., Mirandola, R., "Abstraction-raising Transformation for Generating Analysis Models", Satellite Events at MoDELS 2005, Revised Selected Papers, (Jean-Michel Bruel, Editor), Lecture Notes in Computer Science Vol.3844, pp. 217 - 226, Springer, 2006.

105. Schönberger, S., Keller, R.K. and Khriss, I., "Algorithmic Support for Model Transformation in Object-Oriented Software Development", http://www.iro.umontreal.ca/~keller/Publications/Papers/ccpe-2001.pdf .

106. Schürr, A., "Introduction to PROGRES, an attribute graph grammar based specification language", in Graph-theoretic Concepts in Computer Science, M. Nagl, editor, Vol 411 of Lecture Notes in Computer Science, pp 151-165, 1990.

107. Schürr, A., "PROGRES for Beginners", Lehrstuhl für Informatik III, RWTH Aachen, Ahornstr. 55,D-52074 Aachen, Germany, 1997.

108. Schwetman, H. and Brumfield, J., "Data analysis and automatic run-length control in CSIM18", Proceedings of the 29th conference on Winter simulation, p.687-692, December 07-10, 1997, Atlanta, Georgia, United States.

109. Schwetman, H., "CSIM: a C-based process-oriented simulation language", Proceedings of the 18th Winter Simulation Conference, Washington, D.C., United States, 387 - 396, 1986.

110. Schwetman, H., "CSIM18/OptQuest: optimizing simulations with CSIM18/OptQuest: finding the best configuration", Proceedings of the 32nd conference on Winter simulation, December 10-13, 2000, Orlando, Florida

111. Schwetman, H., "CSIM18-the simulation engine", Proceedings of the 28th conference on Winter simulation, Coronado, California, United States, p.517-521, December 08-11, 1996.

112. Schwetman, H., "CSIM19: a powerful tool for building system models", Proceedings of the 33nd conference on Winter simulation, December 09-12, 2001, Arlington, Virginia, United States.

113. Schwetman, H., "Model, then build: a modern approach to systems development using CSIM18", Proceedings of the 31st conference on Winter simulation: Simulation---a bridge to the future, p.249-254, December 05-08, 1999, Phoenix, Arizona, United States.

114. Schwetman, H., "Model-based systems analysis using CSIM18", Proceedings of the 30th conference on Winter simulation, p.309-314, December 13-16, 1998, Washington, D.C., United States.

115. Sereno, M., "Towards a Product Form Solution for Stochastic Process Algebras", Computer Journal, Vol 38, No.7, pp. 622-632, 1995.

116. Siddiqui, K.H. and Woodside, C.M., "Performance Aware Software Development (PASD) Using Execution Time Budgets", Proc MICON 2001, Ottawa, Aug. 23,24, 2001.

117. Siddiqui, K.H. and Woodside, C.M., "Performance Aware Software Development (PASD) Using Resource Demand Budgets", Proc. Third Int. Workshop on Software and Performance, Rome, July 2002.

118. Smith, C. U. Performance Engineering of Software Systems, Reading, MA, Addison-Wesley, 1990.

119. Smith, C.U. and Williams, L.G., "Performance Engineering Evaluation of Object Oriented Systems with SPE.ED" in 'Computer Performance Evaluation: Modeling Techniques and Tools' LNCS 1245 (R. Marie et alt. Eds.), Springer-Verlag, 1997.

120. Smith, C.U. and Williams, L.G., Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software, Boston, MA, Addison-Wesley, 2002.

121. Telelogic Tau and Rhapsody, http://www.ilogix.com/homepage.aspx.

122. The PUMA Project, http://www.sce.carleton.ca/rads/puma/ .

123. Together Solo, http://www.borland.com/together/solo/index.html .

124. Trivedi, K.S., "Probability and Statistics with Reliability, Queuing, and Computer Science Applications", John Wiley and Sons, 2001.

125. Verdickt, T., Dhoedt, B. and Gielen, F., "Incorporating SPE into MDA: Including Middleware Performance Details into System Models", Workshop on Software and Performance, Proceedings of the fourth international workshop on Software and performance, Redwood Shores, California, pp120-124, 2004.

126. Verdickt, T., Dhoedt, B., Gielen, F. and Demeester, P., "Modeling the performance of CORBA using Layered Queuing Networks", 29th Euromicro Conference (EUROMICRO'03), September 01 - 06, 2003, Belek-Antalya, Turkey, pp117.

127. Visual Paradigm for the Unified Modeling Language (VP-UML), http://www.visual-paradigm.com/product/vpuml/.

128. W3C, "XSL Transformations (XSLT)" Version 2.0, W3C Recommendation, 2004, http://www.w3.org/TR/2004/WD-xslt-xquery-serialization-20041029/ .

129. Williams, L.G. and Smith, C.U., "PASASM: A Method for the Performance Assessment of Software Architectures", pp1-13, http://www.perfeng.com/papers/pasa.pdf.

130. Williams, L.G. and Smith, C.U., "Performance Evaluation of Software Architectures" in Proc. of WOSP'98, Santa Fe, New Mexico, USA, pp. 164-177 (1998).

131. Woodside, C., Neilson, J., Petriu, D., and Majumdar, S., "The Stochastic rendezvous network model for performance of synchronous client-server-like distributed software" IEEE Transaction on Computer, Vol. 44, pp. 20-34 (1995). [WOSP2000] ACM Proceedings of Workshop on Software and Performance, WOSP2000, Ottawa, Canada (2000).

132. Woodside, C.M., "Throughput calculation for basic stochastic rendezvous networks", Performance Evaluation, Vol. 9, Nb. 2, pp. 143-160, 1989.

133. 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, pp 20-34, January 1995.

134. Woodside, C.M., Petriu, D.C., Petriu, D., Shen, H., Israr, T., Merseguer, J., "Performance by Unified Model Analysis (PUMA)", Proc. 5th Int. Workshop on Software and Performance (WOSP 2005), pp 1-12, July 2005.

135. Xu, J., Woodside, C.M., Petriu, D.C., "Performance Analysis of a Software Design using the UML Profile for Schedulability, Performance and Time", Proc. 13th Int Conf. on Computer Performance Evaluation, Modeling Techniques and Tools (TOOLS 2003), Urbana, Illinois, USA, Sept 2003, pp 291 - 310, vol. LNCS 2794, Lecture Notes in Computer Science, Springer-Verlag, 2003.

136. Zhang, H., Bradbury, J.S., Cordy, J.R., Dingel, J., "Implementation and Verification of Implicit-Invocation Systems Using Source Transformation". 5th International Workshop on Source Code Analysis and Manipulation (SCAM 2005), Budapest, Hungary, September/October 2005.

APPENDIX I: IM DTD

```
<!ELEMENT IM.Model (IM.HardwareEntity | IM.TaskEntity|IM.ServiceEntity| IM.PatternInformation)*>
<!ATTLIST IM.TaskInformation
        modelname CDATA #REQUIRED
        comment CDATA #IMPLIED
>
<!ELEMENT IM.HardwareEntity (IM.Hardware*)>
<!ATTLIST IM.HardwareEntity
        name CDATA #REQUIRED
>
<!ELEMENT IM.Hardware EMPTY>
<!ATTLIST IM.Hardware
        name CDATA #REQUIRED
        taskname CDATA #REQUIRED
        comment CDATA #IMPLIED
>
<!ELEMENT IM.ServiceEntity (IM.Service*)>
<!ATTLIST IM.ServiceEntity
        name CDATA #REQUIRED
>
<!ELEMENT IM.TaskEntity (IM.Task*)>
<!ATTLIST IM.TaskEntity
        name CDATA #REQUIRED
        hardwareref CDATA #REQUIRED
        comment CDATA #IMPLIED
>
<!ELEMENT IM.Task EMPTY>
<!ATTLIST IM.Task
        name CDATA #REQUIRED
        hardware_ref CDATA #REQUIRED
        pname CDATA #REQUIRED
>
<!ELEMENT IM.Service (IM.Step | IM.InitialStep| IM.FinalStep| IM.Fork | IM.Join | IM.Branch |
IM.Merge | IM.CompositeStep | IM.TransitionArc | IM.Iteration)*>
<!ATTLIST IM.Service
        name CDATA #REQUIRED
        hardwareref CDATA #REQUIRED
        think-time CDATA #IMPLIED
>
<!ELEMENT IM.Step (IM.From | IM.To)*>
<!ATTLIST IM.Step
        name CDATA #REQUIRED
        taskname CDATA #REQUIRED
        servicetime CDATA #REQUIRED
        prestate CDATA #REQUIRED
        pname CDATA #REQUIRED
>
<!ELEMENT IM.InitialStep ( IM.To)*>
<!ATTLIST IM.InitialStep
        name CDATA #REQUIRED
        taskname CDATA #REQUIRED
        prestate CDATA #REQUIRED
        pname CDATA #REQUIRED
```

```
>
<!ELEMENT IM.FinalStep (IM.From )*>
<!ATTLIST IM.FinalStep
        name CDATA #REQUIRED
        taskname CDATA #REQUIRED
        prestate CDATA #REQUIRED
        pname CDATA #REQUIRED
>
<!ELEMENT IM.CompositeStep EMPTY>
<!ATTLIST IM.CompositeStep
        name CDATA #REQUIRED
        serviceentity_ref CDATA #REQUIRED
        prestate CDATA #REQUIRED
        pname CDATA #REQUIRED
>
<!ELEMENT IM.Join (IM.From | IM.To)*>
<!ATTLIST IM.Join
        name CDATA #REQUIRED
        taskname CDATA #REQUIRED
        prestate CDATA #REQUIRED
        pname CDATA #REQUIRED
>
<!ELEMENT IM.Fork (IM.From | IM.To)*>
<!ATTLIST IM.Fork
        name CDATA #REQUIRED
        taskname CDATA #REQUIRED
        prestate CDATA #REQUIRED
        pname CDATA #REQUIRED
>
<!ELEMENT IM.Branch (IM.From | IM.To)*>
<!ATTLIST IM.Branch
        name CDATA #REQUIRED
        taskname CDATA #REQUIRED
        prestate CDATA #REQUIRED
        pname CDATA #REQUIRED
>
<!ELEMENT IM.Merge (IM.From | IM.To)*>
<!ATTLIST IM.Merge
        name CDATA #REQUIRED
        taskname CDATA #REQUIRED
        prestate CDATA #REQUIRED
        pname CDATA #REQUIRED
>
<!ELEMENT IM.TransitionArc EMPTY>
<!ATTLIST IM.TransitionArc
        name CDATA #REQUIRED
        sourcetask CDATA #REQUIRED
        targettask CDATA #REQUIRED
        source CDATA #REQUIRED
        target CDATA #REQUIRED
        calltpye CDATA #REQUIRED
>
<!ELEMENT IM.Iteration EMPTY>
<!ATTLIST IM.Iteration
        name CDATA #REQUIRED
        taskname CDATA #REQUIRED
```

```
        begin CDATA #REQUIRED
        end CDATA #REQUIRED
        repeatition CDATA #REQUIRED
        prestate CDATA #REQUIRED
        pname CDATA #REQUIRED
>
<!ELEMENT IM.From EMPTY>
<!ATTLIST IM.From
        transname CDATA #REQUIRED
        fromStep CDATA #REQUIRED
>
<!ELEMENT IM.To EMPTY>
<!ATTLIST IM.To
        transname CDATA #REQUIRED
        toStep CDATA #REQUIRED
>
<!ELEMENT IM.PatternInformation (IM.Pattern)*>
<!ATTLIST IM.PatternInformation
        name CDATA #REQUIRED
>
<!ELEMENT IM.Pattern (IM.PatternAssociate | IM.PatternInteraction)*>
<!ATTLIST IM.Pattern
        name CDATA #IMPLIED
        comment CDATA #REQUIRED
>
<!ELEMENT IM.PatternAssociate EMPTY>
<!ATTLIST IM.PatternAssociate
        name CDATA #REQUIRED
        pname CDATA #REQUIRED
>
<!ELEMENT IM.PatternInteraction EMPTY>
<!ATTLIST IM.PatternInteraction
        name CDATA #REQUIRED
        actiontype CDATA #REQUIRED
        actionsource CDATA #REQUIRED
        actiontarget CDATA #REQUIRED
>
```

APPENDIX II: LQN DTD

```
<!ELEMENT LQN.MOMEL (LQN.HW_DECL | LQN.TASK_DECL | LQN.ENTRY_DECL)*>
<!ATTLIST LQN.MODEL
        modelname       CDATA #REQUIRED
        comment         CDATA #IMPLIED
>
<!ELEMENT LQN.HW_DECL (LQN.HARDWARE)*>
<!ATTLIST LQN.HW_DECL
        name CDATA #REQUIRED
        number_of_process CDATA #REQUIRED
>
<!ELEMENT LQN.HARDWARE EMPTY>
<!ATTLIST LQN.HARDWARE
        proc_id CDATA #REQUIRED
        comment         CDATA #IMPLIED
>
<!ELEMENT LQN.TASK_DECL (LQN.TASK)*>
<!ATTLIST LQN.TASK_DECL
        name CDATA #REQUIRED
        number_of_task CDATA #REQUIRED
>
<!ELEMENT LQN.TASK EMPTY>
<!ATTLIST LQN.TASK
        task_id CDATA #REQUIRED
        proc_id CDATA #REQUIRED
        entry_id CDATA #REQUIRED
        comment CDATA #IMPLIED
>
<!ELEMENT LQN.ENTRY_DECL (LQN.A_DECL|LQN.PHASE|LQN.ACTIVITY)*>
<!ATTLIST LQN.ENTRY_DECL
        name CDATA #REQUIRED
        number_of_entry CDATA #REQUIRED
>
<!ELEMENT LQN.A_DECL EMPTY>
<!ATTLIST LQN.A_DECL
        task_id CDATA #REQUIRED
        activitystart_id CDATA #REQUIRED
>
<!ELEMENT LQN.CALL EMPTY>
<!ATTLIST LQN.CALL
        call_name CDATA #REQUIRED
        from_entry CDATA #REQUIRED
        to_entry CDATA #REQUIRED
        call_type       CDATA #REQUIRED
        comment CDATA #IMPLIED
>
<!ELEMENT LQN.PHASE (LQN.PHASESTEP | LQN.CALL)*>
<!ATTLIST LQN.PHASE
        phase_number CDATA #REQUIRED
        entry_id CDATA #REQUIRED
        think-time      CDATA #REQUIRED
>
```

```
<!ELEMENT LQN.PHASESTEP EMPTY>
<!ATTLIST LQN.PHASE
        entry_id CDATA #REQUIRED
        ph_ser_time CDATA #REQUIRED
>
<!ELEMENT LQN.ACTIVITY (LQN.ACTIVITYSTEP | LQN.CALL | LQN.JOIN | LQN.FORK)*>
<!ATTLIST LQN.ACTIVITY
        task_id CDATA#REQUIRED
        activitystart_id CDATA #REQUIRED
>
<!ELEMENT LQN.ACTIVITYSTEP (LQN.FROM|LQN.TO)*>
<!ATTLIST LQN.ACTIVITYSTEP
        activity_id CDATA #REQUIRED
        serv_time CDATA #REQUIRED
>
<!ELEMENT LQN.JOIN (LQN.FROM|LQN.TO)*>
<!ATTLIST LQN.JOIN
        name CDATA #REQUIRED
        join_type CDATA #REQUIRED
        source_activity CDATA #REQUIRED
        target_avtivity CDATA #REQUIRED
>
<!ELEMENT LQN.FORK (LQN.FROM|LQN.TO)*>
<!ATTLIST LQN.FORK
        name CDATA #REQUIRED
        fork_type CDATA #REQUIRED
        source_activity CDATA #REQUIRED
        target_avtivity CDATA #REQUIRED
>
<!ELEMENT LQN.FROM EMPTY>
<!ATTLIST LQN.FROM
        transname CDATA #REQUIRED
        fromStep CDATA #REQUIRED
>
<!ELEMENT LQN.TO EMPTY>
<!ATTLIST LQN.TO
        transname CDATA #REQUIRED
        toStep CDATA #REQUIRED
>
```

APPENDIX III: CSIM DTD

```
<!ELEMENT CSIM.Model (CSIM.HardwareFacilitySet | CSIM.SoftwareFacilitySet | CSIM.ProcessSet)*>
<!ATTLIST CSIM.Model
        modelname CDATA #REQUIRED
        comment CDATA #IMPLIED
>
<!ELEMENT CSIM.HardwareFacilitySet (CSIM.HardwareFacility)*>
<!ATTLIST CSIM.HardwareFacilitySet
        name CDATA #REQUIRED
>
<!ELEMENT CSIM.HardwareFacility EMPTY>
<!ATTLIST CSIM.HardwareFacility
        name CDATA #REQUIRED
        comment CDATA #IMPLIED
>
<!ELEMENT CSIM.SoftwareFacilitySet (CSIM.SoftwareFacility)*>
<!ATTLIST CSIM.SoftwareFacilitySet
        name CDATA #REQUIRED
>
<!ELEMENT CSIM.SoftwareFacility (CSIM.MailBox|CSIM.ChildFacility)*>
<!ATTLIST CSIM.SoftwareFacility
        name CDATA #REQUIRED
        hardware_ref CDATA #REQUIRED
        comment CDATA #IMPLIED
>
<!ELEMENT CSIM.ChildFacility EMPTY>
<!ATTLIST CSIM.ChildFacility
        name CDATA #REQUIRED
        hardware_ref    CDATA #REQUIRED
        comment CDATA #IMPLIED
>
<!ELEMENT CSIM.ProcessSet (CSIM.Process)*>
<!ATTLIST CSIM.ProcessSet
        name CDATA #REQUIRED
>
<!ELEMENT CSIM.Process (CSIM.Step | CSIM.Fork | CSIM.Join | CSIM.OrFork | CSIM.OrJoin|
CSIM.MessageIn | CSIM.MessageOUT | CSIM.Iteration-begin | CSIM.Iteration-
end|CSIM.ChildProcess)*>
<!ATTLIST CSIM.Process
        name CDATA #REQUIRED
        hardware_ref    CDATA #REQUIRED
        software_ref CDATA #REQUIRED
>
<!ELEMENT CSIM.MailBox EMPTY>
<!ATTLIST CSIM.MailBox
        name CDATA #REQUIRED
        software_ref CDATA #REQUIRED
>
<!ELEMENT CSIM.ChildProcess EMPTY>
<!ATTLIST CSIM.ChildProcess
        startStep CDATA #REQUIRED
        endStep CDATA #REQUIRED
>
```

```
<!ELEMENT CSIM.Step (CSIM.From | CSIM.To)*>
<!ATTLIST CSIM.Step
        name CDATA #REQUIRED
        servicetime CDATA #REQUIRED
>
<!ELEMENT CSIM.Join (CSIM.From | CSIM.To)*>
<!ATTLIST CSIM.Join
        name CDATA #REQUIRED
        condition CDATA #REQUIRED
>
<!ELEMENT CSIM.Fork (CSIM.From | CSIM.To)*>
<!ATTLIST CSIM.Fork
        name CDATA #REQUIRED
        condition CDATA #REQUIRED
>
<!ELEMENT CSIM.OrJoin (CSIM.From | CSIM.To)*>
<!ATTLIST CSIM.OrJoin
        name CDATA #REQUIRED
        condition CDATA #REQUIRED
>
<!ELEMENT CSIM.OrFork (CSIM.From | CSIM.To)*>
<!ATTLIST CSIM.OrFork
        name CDATA #REQUIRED
        condition CDATA #REQUIRED
>
<!ELEMENT CSIM.MessageOUT EMPTY>
<!ATTLIST CSIM.MessageOUT
        name CDATA #REQUIRED
        fromProcess CDATA #REQUIRED
        toProcess CDATA #REQUIRED
>
<!ELEMENT CSIM.MessageIn EMPTY>
<!ATTLIST CSIM.MessageIn
        name CDATA #REQUIRED
        fromProcess CDATA #REQUIRED
        toProcess CDATA #REQUIRED
>
<!ELEMENT CSIM.From EMPTY>
<!ATTLIST CSIM.From
        transname CDATA #REQUIRED
>
<!ELEMENT CSIM.To EMPTY>
<!ATTLIST CSIM.To
        transname CDATA #REQUIRED
>
<!ELEMENT CSIM.Iteration-begin EMPTY>
<!ATTLIST CSIM.Iteration-begin
        startStepCDATA #REQUIRED
        repeatition CDATA #REQUIRED
>
<!ELEMENT CSIM.Iteration-end EMPTY>
<!ATTLIST CSIM.Iteration-end
        endStep CDATA #REQUIRED
>
```

APPENDIX IV: TEST OUTPUTS

Test Case 1: Client-Server (IM)

```
<?xml version="1.0" encoding="utf-8" ?>
- <IM:Model xmlns:IM="http://www.w3.org/1999/XSL/Format" name="client-server-1">
  - <IM:HardwareEntity name="Hardware">
      <IM:Hardware name="ClientProc" taskname="Client" />
      <IM:Hardware name="ServerProc" taskname="Server" />
    </IM:HardwareEntity>
  - <IM:TaskEntity name="Task">
      <IM:Task name="Client" hardwareref="ClientProc" />
      <IM:Task name="Server" hardwareref="ServerProc" />
    </IM:TaskEntity>
  - <IM:ServiceEntity name="Service">
    - <IM:Service name="Client" hardwareref="ClientProc" think-time="60">
      - <IM:InitialStep name="init" taskname="Client" prestate="none"
            pname="Client">
          <IM:To transname="t1" toStep="send" />
        </IM:InitialStep>
      - <IM:Step name="send" taskname="Client" servicetime="3"
            prestate="none" pname="Client">
          <IM:To transname="send" toStep="j1" />
          <IM:From transname="t1" fromStep="init" />
        </IM:Step>
        <IM:TransitionArc name="send" sourcetask="Client" targettask="Server"
            source="send" target="j1" arctype="request" calltype="synchronous"
            />
      - <IM:Step name="receive" taskname="Client" servicetime="0.7"
            prestate="none" pname="Client">
          <IM:To transname="t5" toStep="end" />
          <IM:From transname="reply" fromStep="f1" />
        </IM:Step>
      - <IM:FinalStep name="end" taskname="Client" prestate="none"
            pname="Client">
          <IM:From transname="t5" fromStep="receive" />
        </IM:FinalStep>
      </IM:Service>
    - <IM:Service name="Server" hardwareref="ServerProc">
      - <IM:Step name="idle" taskname="Server" servicetime=""
            prestate="none" pname="Server">
          <IM:To transname="t2" toStep="j1" />
        </IM:Step>
      - <IM:Join name="j1" taskname="Server" prestate="j1" pname="Server">
          <IM:To transname="t3" toStep="process" />
          <IM:From transname="send" fromStep="send" />
          <IM:From transname="t2" fromStep="idle" />
        </IM:Join>
      - <IM:Step name="process" taskname="Server" servicetime="3"
            prestate="j1" pname="Server">
          <IM:To transname="t4" toStep="f1" />
          <IM:From transname="t3" fromStep="j1" />
        </IM:Step>
```

```
      - <IM:Fork name="f1" taskname="Server" prestate="f1" pname="Server">
          <IM:To transname="reply" toStep="receive" />
          <IM:To transname="t6" toStep="undef" />
          <IM:From transname="t4" fromStep="process" />
        </IM:Fork>
        <IM:TransitionArc name="reply" sourcetask="Server" targettask="Client"
            source="f1" target="receive" arctype="reply" calltype="synchronous"
            />
      - <IM:Step name="undef" taskname="Server" servicetime="" prestate="f1"
            pname="Server">
          <IM:From transname="t6" fromStep="f1" />
        </IM:Step>
      </IM:Service>
    </IM:ServiceEntity>
  - <IM:PatternInformation name="Pattern">
    - <IM:Pattern name="Client Server">
        <IM:PatternAssociate name="Client" />
        <IM:PatternAssociate name="Server" />
        <IM:PatternInteraction name="request" actiontype="synchronous"
            actionsource="Client" actiontarget="Server" />
        <IM:PatternInteraction name="reply" actiontype="synchronous"
            actionsource="Server" actiontarget="Client" />
      </IM:Pattern>
    </IM:PatternInformation>
  </IM:Model>
```

Client-Server (LQN)

```
  <?xml version="1.0" encoding="utf-8" ?>
- <LQN:Root xmlns:LQN="http://www.w3.org/1999/XSL/Format">
    - <LQN:MODEL modelname="client-server-1" comment="G 0.0000001 100 1 0.9 -
        1">
      - <LQN:HW_DECL name="Hardware" number_of_processor="2">
          <LQN:HARDWARE proc_id="ClientProc" comment="f i" />
          <LQN:HARDWARE proc_id="ServerProc" comment="f i" />
        </LQN:HW_DECL>
      - <LQN:TASK_DECL name="Task" number_of_task="2">
          <LQN:TASK task_id="Client" proc_id="ClientProc" entry_id="Client_e"
              comment="m 1" />
          <LQN:TASK task_id="Server" proc_id="ServerProc" entry_id="Server_e"
              comment="m 1" />
        </LQN:TASK_DECL>
      - <LQN:ENTRY_DECL name="Service" number_of_entry="2">
        - <LQN:PHASE name="Client" entry_id="Client_e" think-time="60">
            <LQN:PHASESTEP entry_id="Client_e" ph_ser_time="3.7" />
            <LQN:CALL call_name="send" from_entry="Client_e"
                to_entry="Server_e" call_type="synchronous" comment="call
                comment" />
          </LQN:PHASE>
        - <LQN:PHASE name="Server" entry_id="Server_e">
            <LQN:PHASESTEP entry_id="Server_e" ph_ser_time="3" />
          </LQN:PHASE>
        </LQN:ENTRY_DECL>
      </LQN:MODEL>
  </LQN:Root>
```

Client-Server (CSIM)

```
<?xml version="1.0" encoding="utf-8" ?>
- <CSIM:Root xmlns:CSIM="http://www.w3.org/1999/XSL/Format">
  - <CSIM:Model modelname="client-server-1" comment="closeload">
    - <CSIM:HardwareFacilitySet name="Hardware">
        <CSIM:HardwareFacility name="ClientProc" comment="n/a" />
        <CSIM:HardwareFacility name="ServerProc" comment="n/a" />
      </CSIM:HardwareFacilitySet>
    - <CSIM:SoftwareFacilitySet name="Task">
      - <CSIM:SoftwareFacility name="Client_s" hardwareref="ClientProc"
            comment="n/a">
          <CSIM:MailBox name="Client_mailbox" softwareref="Client_s" />
        </CSIM:SoftwareFacility>
      - <CSIM:SoftwareFacility name="Server_s" hardwareref="ServerProc"
            comment="n/a">
          <CSIM:MailBox name="Server_mailbox" softwareref="Server_s" />
        </CSIM:SoftwareFacility>
      </CSIM:SoftwareFacilitySet>
    - <CSIM:ProcessSet name="Service">
      - <CSIM:Process name="Client" hardwareref="ClientProc"
            softwareref="Client_s">
        - <CSIM:Step name="send" servicetime="3">
            <CSIM:To transname="send" target="j1" />
            <CSIM:From transname="t1" source="init" />
          </CSIM:Step>
          <CSIM:MessageOut name="send" fromProcess="Client"
              toProcess="Server" />
        - <CSIM:Step name="receive" servicetime="0.7">
            <CSIM:To transname="t5" target="end" />
            <CSIM:From transname="reply" source="f1" />
          </CSIM:Step>
        </CSIM:Process>
      - <CSIM:Process name="Server" hardwareref="ServerProc"
            softwareref="Server_s">
        - <CSIM:Step name="idle">
            <CSIM:To transname="t2" target="j1" />
          </CSIM:Step>
        - <CSIM:Join name="j1" condition="n/a">
            <CSIM:To transname="t3" target="process" />
            <CSIM:From transname="send" source="send" />
            <CSIM:From transname="t2" source="idle" />
          </CSIM:Join>
        - <CSIM:Step name="process" servicetime="3">
            <CSIM:To transname="t4" target="f1" />
            <CSIM:From transname="t3" source="j1" />
          </CSIM:Step>
        - <CSIM:Fork name="f1" condition="n/a">
            <CSIM:To transname="reply" target="receive" />
            <CSIM:To transname="t6" target="undef" />
            <CSIM:From transname="t4" source="process" />
          </CSIM:Fork>
```

```
            <CSIM:MessageIn name="reply" fromProcess="Server"
                toProcess="Client" />
          - <CSIM:Step name="undef">
                <CSIM:From transname="t6" source="f1" />
            </CSIM:Step>
        </CSIM:Process>
    </CSIM:ProcessSet>
</CSIM:Model>
</CSIM:Root>
```

Test Case 2: Pipeline (IM)

```
<?xml version="1.0" encoding="utf-8" ?>
- <IM:Model xmlns:IM="http://www.w3.org/1999/XSL/Format" name="pipeline">
  - <IM:HardwareEntity name="Hardware">
      <IM:Hardware name="FilterProc-1" taskname="Filter-1" />
      <IM:Hardware name="FilterProc-2" taskname="Filter-2" />
    </IM:HardwareEntity>
  - <IM:TaskEntity name="Task">
      <IM:Task name="Filter-1" hardwareref="FilterProc-1" />
      <IM:Task name="Filter-2" hardwareref="FilterProc-2" />
    </IM:TaskEntity>
  - <IM:ServiceEntity name="Service">
    - <IM:Service name="Filter-1" hardwareref="FilterProc-1">
      - <IM:Step name="f1-action-1" taskname="Filter-1" servicetime="0.5"
            prestate="none" pname="Filter-1">
          <IM:To transname="t1" toStep="f1-pass-info" />
        </IM:Step>
      - <IM:Step name="f1-pass-info" taskname="Filter-1" servicetime="0.6"
            prestate="none" pname="Filter-1">
          <IM:To transname="send" toStep="f2-receive" />
          <IM:To transname="t2" toStep="f1-action-2" />
          <IM:From transname="t1" fromStep="f1-action-1" />
        </IM:Step>
        <IM:TransitionArc name="send" sourcetask="Filter-1" targettask="Filter-
            2" source="f1-pass-info" target="f2-receive" arctype="request"
            calltype="Asynchronous" />
      - <IM:Step name="f1-action-2" taskname="Filter-1" servicetime="0.3"
            prestate="none" pname="Filter-1">
          <IM:From transname="t2" fromStep="f1-pass-info" />
        </IM:Step>
      </IM:Service>
    - <IM:Service name="Filter-2" hardwareref="FilterProc-2">
      - <IM:Step name="f2-action-1" taskname="Filter-2" servicetime="0.7"
            prestate="none" pname="Filter-2">
          <IM:To transname="t3" toStep="f2-receive" />
        </IM:Step>
      - <IM:Step name="f2-receive" taskname="Filter-2" servicetime="1.7"
            prestate="none" pname="Filter-2">
          <IM:To transname="t4" toStep="f2-pass-info" />
          <IM:From transname="send" fromStep="f1-pass-info" />
          <IM:From transname="t3" fromStep="f2-action-1" />
        </IM:Step>
      - <IM:Step name="f2-pass-info" taskname="Filter-2" servicetime="0.7"
            prestate="none" pname="Filter-2">
          <IM:To transname="t5" toStep="f2-action-2" />
          <IM:From transname="t4" fromStep="f2-receive" />
        </IM:Step>
      - <IM:Step name="f2-action-2" taskname="Filter-2" servicetime="0.5"
            prestate="none" pname="Filter-2">
          <IM:From transname="t5" fromStep="f2-pass-info" />
        </IM:Step>
      </IM:Service>
    </IM:ServiceEntity>
  - <IM:PatternInformation name="Pattern">
```

```
    - <IM:Pattern name="Pipeline">
        <IM:PatternAssociate name="Filter-1" />
        <IM:PatternAssociate name="Filter-2" />
        <IM:PatternInteraction name="pass" actiontype="synchronous"
            actionsource="Filter-1" actiontarget="Filter-2" />
      </IM:Pattern>
    </IM:PatternInformation>
  </IM:Model>
```

Pipeline (LQN)

```
  <?xml version="1.0" encoding="utf-8" ?>
- <LQN:Root xmlns:LQN="http://www.w3.org/1999/XSL/Format">
    - <LQN:MODEL modelname="pipeline" comment="G 0.0000001 100 1 0.9 -1">
      - <LQN:HW_DECL name="Hardware" number_of_processor="2">
          <LQN:HARDWARE proc_id="FilterProc-1" comment="f i" />
          <LQN:HARDWARE proc_id="FilterProc-2" comment="f i" />
        </LQN:HW_DECL>
      - <LQN:TASK_DECL name="Task" number_of_task="2">
          <LQN:TASK task_id="Filter-1" proc_id="FilterProc-1" entry_id="Filter-
            1_e" comment="m 1" />
          <LQN:TASK task_id="Filter-2" proc_id="FilterProc-2" entry_id="Filter-
            2_e" comment="m 1" />
        </LQN:TASK_DECL>
      - <LQN:ENTRY_DECL name="Service" number_of_entry="2">
        - <LQN:PHASE name="Filter-1" entry_id="Filter-1_e">
            <LQN:PHASESTEP entry_id="Filter-1_e" ph_ser_time="1.4" />
            <LQN:CALL call_name="send" from_entry="Filter-1_e"
              to_entry="Filter-2_e" call_type="Asynchronous" comment="call
              comment" />
          </LQN:PHASE>
        - <LQN:PHASE name="Filter-2" entry_id="Filter-2_e">
            <LQN:PHASESTEP entry_id="Filter-2_e" ph_ser_time="3.6" />
          </LQN:PHASE>
        </LQN:ENTRY_DECL>
    </LQN:MODEL>
  </LQN:Root>
```

Pipeline (CSIM)

```
  <?xml version="1.0" encoding="utf-8" ?>
- <CSIM:Root xmlns:CSIM="http://www.w3.org/1999/XSL/Format">
    - <CSIM:Model modelname="pipeline" comment="closeload">
      - <CSIM:HardwareFacilitySet name="Hardware">
          <CSIM:HardwareFacility name="FilterProc-1" comment="n/a" />
          <CSIM:HardwareFacility name="FilterProc-2" comment="n/a" />
        </CSIM:HardwareFacilitySet>
      - <CSIM:SoftwareFacilitySet name="Task">
        - <CSIM:SoftwareFacility name="Filter-1_s" hardwareref="FilterProc-1"
            comment="n/a">
            <CSIM:MailBox name="Filter-1_mailbox" softwareref="Filter-1_s" />
          </CSIM:SoftwareFacility>
        - <CSIM:SoftwareFacility name="Filter-2_s" hardwareref="FilterProc-2"
            comment="n/a">
            <CSIM:MailBox name="Filter-2_mailbox" softwareref="Filter-2_s" />
```

```
        </CSIM:SoftwareFacility>
      </CSIM:SoftwareFacilitySet>
    - <CSIM:ProcessSet name="Service">
      - <CSIM:Process name="Filter-1" hardwareref="FilterProc-1"
            softwareref="Filter-1_s">
        - <CSIM:Step name="f1-action-1" servicetime="0.5">
            <CSIM:To transname="t1" target="f1-pass-info" />
          </CSIM:Step>
        - <CSIM:Step name="f1-pass-info" servicetime="0.6">
            <CSIM:To transname="send" target="f2-receive" />
            <CSIM:To transname="t2" target="f1-action-2" />
            <CSIM:From transname="t1" source="f1-action-1" />
          </CSIM:Step>
          <CSIM:MessageOut name="send" fromProcess="Filter-1"
              toProcess="Filter-2" />
        - <CSIM:Step name="f1-action-2" servicetime="0.3">
            <CSIM:From transname="t2" source="f1-pass-info" />
          </CSIM:Step>
        </CSIM:Process>
      - <CSIM:Process name="Filter-2" hardwareref="FilterProc-2"
            softwareref="Filter-2_s">
        - <CSIM:Step name="f2-action-1" servicetime="0.7">
            <CSIM:To transname="t3" target="f2-receive" />
          </CSIM:Step>
        - <CSIM:Step name="f2-receive" servicetime="1.7">
            <CSIM:To transname="t4" target="f2-pass-info" />
            <CSIM:From transname="send" source="f1-pass-info" />
            <CSIM:From transname="t3" source="f2-action-1" />
          </CSIM:Step>
        - <CSIM:Step name="f2-pass-info" servicetime="0.7">
            <CSIM:To transname="t5" target="f2-action-2" />
            <CSIM:From transname="t4" source="f2-receive" />
          </CSIM:Step>
        - <CSIM:Step name="f2-action-2" servicetime="0.5">
            <CSIM:From transname="t5" source="f2-pass-info" />
          </CSIM:Step>
        </CSIM:Process>
      </CSIM:ProcessSet>
    </CSIM:Model>
  </CSIM:Root>
```

Test Case 3: Fork-Case (IM)

```
<?xml version="1.0" encoding="utf-8" ?>
- <IM:Model xmlns:IM="http://www.w3.org/1999/XSL/Format" name="fork-case">
  - <IM:HardwareEntity name="Hardware">
      <IM:Hardware name="Appproc" taskname="Application" />
    </IM:HardwareEntity>
  - <IM:TaskEntity name="Task">
      <IM:Task name="Application" hardwareref="Appproc" />
    </IM:TaskEntity>
  - <IM:ServiceEntity name="Service">
    - <IM:Service name="Application" hardwareref="Appproc">
      - <IM:Step name="a" taskname="Application" servicetime="1.5"
            prestate="none" pname="Application">
          <IM:To transname="t1" toStep="Fork" />
        </IM:Step>
      - <IM:Fork name="Fork" taskname="Application" prestate="Fork"
            pname="Application">
          <IM:To transname="t2" toStep="b1" />
          <IM:To transname="t3" toStep="c1" />
          <IM:From transname="t1" fromStep="a" />
        </IM:Fork>
      - <IM:Step name="b1" taskname="Application" servicetime="0.4"
            prestate="Fork" pname="Application">
          <IM:To transname="t4" toStep="b2" />
          <IM:From transname="t2" fromStep="Fork" />
        </IM:Step>
      - <IM:Step name="b2" taskname="Application" servicetime="0.2"
            prestate="Fork" pname="Application">
          <IM:To transname="t6" toStep="Join" />
          <IM:From transname="t4" fromStep="b1" />
        </IM:Step>
      - <IM:Step name="c1" taskname="Application" servicetime="0.1"
            prestate="Fork" pname="Application">
          <IM:To transname="t5" toStep="c2" />
          <IM:From transname="t3" fromStep="Fork" />
        </IM:Step>
      - <IM:Step name="c2" taskname="Application" servicetime="0.7"
            prestate="Fork" pname="Application">
          <IM:To transname="t7" toStep="Join" />
          <IM:From transname="t5" fromStep="c1" />
        </IM:Step>
      - <IM:Join name="Join" taskname="Application" prestate="Join"
            pname="Application">
          <IM:To transname="t8" toStep="d" />
          <IM:From transname="t6" fromStep="b2" />
          <IM:From transname="t7" fromStep="c2" />
        </IM:Join>
      - <IM:Step name="d" taskname="Application" servicetime="1.5"
            prestate="Join" pname="Application">
          <IM:From transname="t8" fromStep="Join" />
        </IM:Step>
      </IM:Service>
    </IM:ServiceEntity>
    <IM:PatternInformation name="Pattern" />
```

```
</IM:Model>
```

Fork-Case (LQN)

```
<?xml version="1.0" encoding="utf-8" ?>
- <LQN:Root xmlns:LQN="http://www.w3.org/1999/XSL/Format">
  - <LQN:MODEL modelname="fork-case" comment="G 0.0000001 100 1 0.9 -1">
    - <LQN:HW_DECL name="Hardware" number_of_processor="1">
        <LQN:HARDWARE proc_id="Appproc" comment="f i" />
      </LQN:HW_DECL>
    - <LQN:TASK_DECL name="Task" number_of_task="1">
        <LQN:TASK task_id="Application" proc_id="Appproc"
            entry_id="Application_e" comment="m 1" />
      </LQN:TASK_DECL>
    - <LQN:ENTRY_DECL name="Service" number_of_entry="1">
        <LQN:A_DECL task_id="Application" entry_id="Application_e"
            activity_id="a" />
      - <LQN:ACTIVITY task_id="Application" entry_id="Application_e"
            activity_id="a">
        - <LQN:ACTIVITYSTEP activity="a" serv_time="1.5">
            <LQN:To transname="t1" toStep="Fork" />
          </LQN:ACTIVITYSTEP>
        - <LQN:FORK name="Fork" fork_type="and">
            <LQN:To transname="t2" toStep="b1" />
            <LQN:To transname="t3" toStep="c1" />
            <LQN:From transname="t1" fromStep="a" />
          </LQN:FORK>
        - <LQN:ACTIVITYSTEP activity="b1" serv_time="0.4">
            <LQN:To transname="t4" toStep="b2" />
            <LQN:From transname="t2" fromStep="Fork" />
          </LQN:ACTIVITYSTEP>
        - <LQN:ACTIVITYSTEP activity="b2" serv_time="0.2">
            <LQN:To transname="t6" toStep="Join" />
            <LQN:From transname="t4" fromStep="b1" />
          </LQN:ACTIVITYSTEP>
        - <LQN:ACTIVITYSTEP activity="c1" serv_time="0.1">
            <LQN:To transname="t5" toStep="c2" />
            <LQN:From transname="t3" fromStep="Fork" />
          </LQN:ACTIVITYSTEP>
        - <LQN:ACTIVITYSTEP activity="c2" serv_time="0.7">
            <LQN:To transname="t7" toStep="Join" />
            <LQN:From transname="t5" fromStep="c1" />
          </LQN:ACTIVITYSTEP>
        - <LQN:JOIN name="Join" join_type="and">
            <LQN:To transname="t8" toStep="d" />
            <LQN:From transname="t6" fromStep="b2" />
            <LQN:From transname="t7" fromStep="c2" />
          </LQN:JOIN>
        - <LQN:ACTIVITYSTEP activity="d" serv_time="1.5">
            <LQN:From transname="t8" fromStep="Join" />
          </LQN:ACTIVITYSTEP>
        </LQN:ACTIVITY>
      </LQN:ENTRY_DECL>
    </LQN:MODEL>
  </LQN:Root>
```

Fork-Case (CSIM)

```
<?xml version="1.0" encoding="utf-8" ?>
- <CSIM:Root xmlns:CSIM="http://www.w3.org/1999/XSL/Format">
  - <CSIM:Model modelname="fork-case" comment="closeload">
    - <CSIM:HardwareFacilitySet name="Hardware">
        <CSIM:HardwareFacility name="Appproc" comment="n/a" />
      </CSIM:HardwareFacilitySet>
    - <CSIM:SoftwareFacilitySet name="Task">
      - <CSIM:SoftwareFacility name="Application_s" hardwareref="Appproc"
            comment="n/a">
          <CSIM:MailBox name="Application_mailbox"
              softwareref="Application_s" />
          <CSIM:ChildFacility name="Application_child"
              hardwareref="Appproc" comment="n/a" />
        </CSIM:SoftwareFacility>
      </CSIM:SoftwareFacilitySet>
    - <CSIM:ProcessSet name="Service">
      - <CSIM:Process name="Application" hardwareref="Appproc"
            softwareref="Application_s">
        - <CSIM:Step name="a" servicetime="1.5">
            <CSIM:To transname="t1" target="Fork" />
          </CSIM:Step>
        - <CSIM:Fork name="Fork" condition="n/a">
            <CSIM:ChildProcess startStep="c1" endStep="Join" />
            <CSIM:To transname="t2" target="b1" />
            <CSIM:To transname="t3" target="c1" />
            <CSIM:From transname="t1" source="a" />
          </CSIM:Fork>
        - <CSIM:Step name="b1" servicetime="0.4">
            <CSIM:To transname="t4" target="b2" />
            <CSIM:From transname="t2" source="Fork" />
          </CSIM:Step>
        - <CSIM:Step name="b2" servicetime="0.2">
            <CSIM:To transname="t6" target="Join" />
            <CSIM:From transname="t4" source="b1" />
          </CSIM:Step>
        - <CSIM:Step name="c1" servicetime="0.1">
            <CSIM:To transname="t5" target="c2" />
            <CSIM:From transname="t3" source="Fork" />
          </CSIM:Step>
        - <CSIM:Step name="c2" servicetime="0.7">
            <CSIM:To transname="t7" target="Join" />
            <CSIM:From transname="t5" source="c1" />
          </CSIM:Step>
        - <CSIM:Join name="Join" condition="n/a">
            <CSIM:To transname="t8" target="d" />
            <CSIM:From transname="t6" source="b2" />
            <CSIM:From transname="t7" source="c2" />
          </CSIM:Join>
        - <CSIM:Step name="d" servicetime="1.5">
            <CSIM:From transname="t8" source="Join" />
          </CSIM:Step>
```

```
        </CSIM:Process>
      </CSIM:ProcessSet>
    </CSIM:Model>
  </CSIM:Root>
```

Test Case 4/5: Merge-Case (IM)

```
<?xml version="1.0" encoding="utf-8" ?>
- <IM:Model xmlns:IM="http://www.w3.org/1999/XSL/Format" name="merge-case">
  - <IM:HardwareEntity name="Hardware">
      <IM:Hardware name="Appproc" taskname="Application" />
    </IM:HardwareEntity>
  - <IM:TaskEntity name="Task">
      <IM:Task name="Application" hardwareref="Appproc" />
    </IM:TaskEntity>
  - <IM:ServiceEntity name="Service">
    - <IM:Service name="Application" hardwareref="Appproc">
      - <IM:Step name="a1" taskname="Application" servicetime="0.5"
            prestate="none" pname="Application">
          <IM:To transname="t1" toStep="decision" />
        </IM:Step>
      - <IM:Branch name="decision" taskname="Application"
            prestate="decision" pname="Application">
          <IM:To transname="t2" toStep="a2" />
          <IM:To transname="t3" toStep="a3" />
          <IM:From transname="t1" fromStep="a1" />
        </IM:Branch>
      - <IM:Step name="a2" taskname="Application" servicetime="1.5"
            prestate="none" pname="Application">
          <IM:To transname="t4" toStep="a4" />
          <IM:From transname="t2" fromStep="decision" />
        </IM:Step>
      - <IM:Step name="a4" taskname="Application" servicetime="1.5"
            prestate="none" pname="Application">
          <IM:To transname="t6" toStep="merge" />
          <IM:From transname="t4" fromStep="a2" />
        </IM:Step>
      - <IM:Step name="a3" taskname="Application" servicetime="0.3"
            prestate="none" pname="Application">
          <IM:To transname="t5" toStep="a5" />
          <IM:From transname="t3" fromStep="decision" />
        </IM:Step>
      - <IM:Step name="a5" taskname="Application" servicetime="2.5"
            prestate="none" pname="Application">
          <IM:To transname="t7" toStep="merge" />
          <IM:From transname="t5" fromStep="a3" />
        </IM:Step>
      - <IM:Merge name="merge" taskname="Application" prestate="merge"
            pname="Application">
          <IM:To transname="t8" toStep="a7" />
          <IM:From transname="t6" fromStep="a4" />
          <IM:From transname="t7" fromStep="a5" />
        </IM:Merge>
      - <IM:CompositeStep name="a7" serviceref="a7-CompositeStep"
            prestate="none" pname="Application">
          <IM:To transname="t9" toStep="a8" />
          <IM:To transname="t10:Loop=10" toStep="a7" />
          <IM:From transname="t8" fromStep="merge" />
          <IM:From transname="t10:Loop=10" fromStep="a7" />
        - <IM:Service name="a7-CompositeStep" hardwareref="Appproc">
```

```
            - <IM:Step name="b1" taskname="a7-CompositeStep"
                  servicetime="0.5" prestate="none" pname="Application">
                <IM:To transname="t7a" toStep="b2" />
                <IM:From transname="t7in" fromStep="" />
              </IM:Step>
            - <IM:Step name="b2" taskname="a7-CompositeStep"
                  servicetime="0.7" prestate="none" pname="Application">
                <IM:To transname="t7b" toStep="b3" />
                <IM:From transname="t7a" fromStep="b1" />
              </IM:Step>
            - <IM:Step name="b3" taskname="a7-CompositeStep"
                  servicetime="0.4" prestate="none" pname="Application">
                <IM:To transname="t7out" toStep="" />
                <IM:From transname="t7b" fromStep="b2" />
              </IM:Step>
          </IM:Service>
        </IM:CompositeStep>
      - <IM:Step name="a8" taskname="Application" servicetime="0.5"
            prestate="none" pname="Application">
          <IM:From transname="t9" fromStep="a7" />
        </IM:Step>
        <IM:Iteration name="t10" taskname="Application" begin="a7" end="a7"
            repeatition="10" prestate="none" pname="Application" />
    </IM:Service>
  </IM:ServiceEntity>
  <IM:PatternInformation name="Pattern" />
</IM:Model>
```

Merge-Case (LQN)

```
<?xml version="1.0" encoding="utf-8" ?>
- <LQN:Root xmlns:LQN="http://www.w3.org/1999/XSL/Format">
  - <LQN:MODEL modelname="merge-case" comment="G 0.0000001 100 1 0.9 -1">
    - <LQN:HW_DECL name="Hardware" number_of_processor="1">
        <LQN:HARDWARE proc_id="Appproc" comment="f i" />
      </LQN:HW_DECL>
    - <LQN:TASK_DECL name="Task" number_of_task="1">
        <LQN:TASK task_id="Application" proc_id="Appproc"
            entry_id="Application_e" comment="m 1" />
      </LQN:TASK_DECL>
    - <LQN:ENTRY_DECL name="Service" number_of_entry="1">
        <LQN:A_DECL task_id="Application" entry_id="Application_e"
            activity_id="a1" />
      - <LQN:ACTIVITY task_id="Application" entry_id="Application_e"
            activity_id="a1">
        - <LQN:ACTIVITYSTEP activity="a1" serv_time="0.5">
            <LQN:To transname="t1" toStep="decision" />
          </LQN:ACTIVITYSTEP>
        - <LQN:FORK name="decision" fork_type="or">
            <LQN:To transname="t2" toStep="a2" />
            <LQN:To transname="t3" toStep="a3" />
            <LQN:From transname="t1" fromStep="a1" />
          </LQN:FORK>
        - <LQN:ACTIVITYSTEP activity="a2" serv_time="1.5">
            <LQN:To transname="t4" toStep="a4" />
```

```
            <LQN:From transname="t2" fromStep="decision" />
          </LQN:ACTIVITYSTEP>
        - <LQN:ACTIVITYSTEP activity="a4" serv_time="1.5">
            <LQN:To transname="t6" toStep="merge" />
            <LQN:From transname="t4" fromStep="a2" />
          </LQN:ACTIVITYSTEP>
        - <LQN:ACTIVITYSTEP activity="a3" serv_time="0.3">
            <LQN:To transname="t5" toStep="a5" />
            <LQN:From transname="t3" fromStep="decision" />
          </LQN:ACTIVITYSTEP>
        - <LQN:ACTIVITYSTEP activity="a5" serv_time="2.5">
            <LQN:To transname="t7" toStep="merge" />
            <LQN:From transname="t5" fromStep="a3" />
          </LQN:ACTIVITYSTEP>
        - <LQN:JOIN name="merge" join_type="or">
            <LQN:To transname="t8" toStep="a7" />
            <LQN:From transname="t6" fromStep="a4" />
            <LQN:From transname="t7" fromStep="a5" />
          </LQN:JOIN>
        - <LQN:ACTIVITYSTEP activity="a7" serv_time="16">
            <LQN:To transname="t9" toStep="a8" />
            <LQN:From transname="t8" fromStep="merge" />
          </LQN:ACTIVITYSTEP>
        - <LQN:ACTIVITYSTEP activity="a8" serv_time="0.5">
            <LQN:From transname="t9" fromStep="a7" />
          </LQN:ACTIVITYSTEP>
        </LQN:ACTIVITY>
      </LQN:ENTRY_DECL>
    </LQN:MODEL>
  </LQN:Root>
```

Merge-Case (CSIM)

```
  <?xml version="1.0" encoding="utf-8" ?>
- <CSIM:Root xmlns:CSIM="http://www.w3.org/1999/XSL/Format">
    - <CSIM:Model modelname="merge-case" comment="closeload">
        - <CSIM:HardwareFacilitySet name="Hardware">
            <CSIM:HardwareFacility name="Appproc" comment="n/a" />
          </CSIM:HardwareFacilitySet>
        - <CSIM:SoftwareFacilitySet name="Task">
            - <CSIM:SoftwareFacility name="Application_s" hardwareref="Appproc"
                comment="n/a">
                <CSIM:MailBox name="Application_mailbox"
                  softwareref="Application_s" />
              </CSIM:SoftwareFacility>
          </CSIM:SoftwareFacilitySet>
        - <CSIM:ProcessSet name="Service">
            - <CSIM:Process name="Application" hardwareref="Appproc"
                softwareref="Application_s">
              - <CSIM:Step name="a1" servicetime="0.5">
                  <CSIM:To transname="t1" target="decision" />
                </CSIM:Step>
              - <CSIM:OrFork name="decision" condition="n/a">
                  <CSIM:To transname="t2" target="a2" />
                  <CSIM:To transname="t3" target="a3" />
```

```
            <CSIM:From transname="t1" source="a1" />
          </CSIM:OrFork>
        - <CSIM:Step name="a2" servicetime="1.5">
            <CSIM:To transname="t4" target="a4" />
            <CSIM:From transname="t2" source="decision" />
          </CSIM:Step>
        - <CSIM:Step name="a4" servicetime="1.5">
            <CSIM:To transname="t6" target="merge" />
            <CSIM:From transname="t4" source="a2" />
          </CSIM:Step>
        - <CSIM:Step name="a3" servicetime="0.3">
            <CSIM:To transname="t5" target="a5" />
            <CSIM:From transname="t3" source="decision" />
          </CSIM:Step>
        - <CSIM:Step name="a5" servicetime="2.5">
            <CSIM:To transname="t7" target="merge" />
            <CSIM:From transname="t5" source="a3" />
          </CSIM:Step>
        - <CSIM:OrJoin name="merge" condition="n/a">
            <CSIM:To transname="t8" target="a7" />
            <CSIM:From transname="t6" source="a4" />
            <CSIM:From transname="t7" source="a5" />
          </CSIM:OrJoin>
          <CSIM:Iteration-begin startStep="a7" repeatition="10" />
        - <CSIM:Step name="a7" servicetime="0.5">
            <CSIM:To transname="t9" target="a8" />
            <CSIM:From transname="t8" source="merge" />
          </CSIM:Step>
          <CSIM:Iteration-end endStep="a7" />
        - <CSIM:Step name="a8" servicetime="0.5">
            <CSIM:From transname="t9" source="a7" />
          </CSIM:Step>
        </CSIM:Process>
      </CSIM:ProcessSet>
    </CSIM:Model>
</CSIM:Root>
```

Test Case 6: Buffer (IM)

```
<?xml version="1.0" encoding="utf-8" ?>
- <IM:Model xmlns:IM="http://www.w3.org/1999/XSL/Format" name="buffer">
  - <IM:HardwareEntity name="Hardware">
      <IM:Hardware name="AppProc" taskname="Application" />
    </IM:HardwareEntity>
  - <IM:TaskEntity name="Task">
      <IM:Task name="Application" hardwareref="AppProc" />
      <IM:Task name="Buffer" hardwareref="AppProc" />
    </IM:TaskEntity>
  - <IM:ServiceEntity name="Service">
    - <IM:Service name="Application" hardwareref="AppProc">
      - <IM:Step name="acquire" taskname="Application" servicetime="0.5"
          prestate="none" pname="Application">
        <IM:To transname="acquire" toStep="join-1" />
      </IM:Step>
      <IM:TransitionArc name="acquire" sourcetask="Application"
          targettask="Buffer" source="acquire" target="join-1"
          arctype="request" calltype="synchronous" />
      - <IM:Step name="use" taskname="Application" servicetime="3.5"
          prestate="none" pname="Application">
        <IM:To transname="t3" toStep="fork-2" />
        <IM:From transname="alloc_buff" fromStep="return_buffer" />
      </IM:Step>
      - <IM:Fork name="fork-2" taskname="Application" prestate="fork-2"
          pname="Application">
        <IM:To transname="t4" toStep="other job" />
        <IM:To transname="release" toStep="buffer_avail" />
        <IM:From transname="t3" fromStep="use" />
      </IM:Fork>
      - <IM:Step name="other job" taskname="Application" servicetime="3.0"
          prestate="fork-2" pname="Application">
        <IM:From transname="t4" fromStep="fork-2" />
      </IM:Step>
      <IM:TransitionArc name="release" sourcetask="Application"
          targettask="Buffer" source="fork-2" target="buffer_avail"
          arctype="request" calltype="asynchronous" />
    </IM:Service>
    - <IM:Service name="Buffer" hardwareref="AppProc">
      - <IM:Join name="join-1" taskname="Buffer" prestate="join-1"
          pname="Buffer">
        <IM:To transname="t2" toStep="return_buffer" />
        <IM:From transname="acquire" fromStep="acquire" />
        <IM:From transname="t1" fromStep="Buffer_idle" />
      </IM:Join>
      - <IM:Step name="return_buffer" taskname="Buffer" servicetime="0.5"
          prestate="join-1" pname="Buffer">
        <IM:To transname="alloc_buff" toStep="use" />
        <IM:From transname="t2" fromStep="join-1" />
      </IM:Step>
      <IM:TransitionArc name="alloc_buff" sourcetask="Buffer"
          targettask="Application" source="return_buffer" target="use"
          arctype="request" calltype="synchronous" />
```

```
        - <IM:Step name="buffer_avail" taskname="Buffer" servicetime="0.25"
              prestate="join-1" pname="Buffer">
            <IM:From transname="release" fromStep="fork-2" />
          </IM:Step>
        </IM:Service>
      </IM:ServiceEntity>
    - <IM:PatternInformation name="Pattern">
      - <IM:Pattern name="Buffer">
          <IM:PatternAssociate name="Application" />
          <IM:PatternAssociate name="Buffer" />
          <IM:PatternInteraction name="acquire" actiontype="synchronous"
              actionsource="Application" actiontarget="Buffer" />
          <IM:PatternInteraction name="return-buffer" actiontype="synchronous"
              actionsource="Buffer" actiontarget="Application" />
        </IM:Pattern>
      </IM:PatternInformation>
  </IM:Model>
```

Buffer (LQN)

```
 <?xml version="1.0" encoding="utf-8" ?>
- <LQN:Root xmlns:LQN="http://www.w3.org/1999/XSL/Format">
    - <LQN:MODEL modelname="buffer" comment="G 0.0000001 100 1 0.9 -1">
      - <LQN:HW_DECL name="Hardware" number_of_processor="1">
          <LQN:HARDWARE proc_id="AppProc" comment="f i" />
        </LQN:HW_DECL>
      - <LQN:TASK_DECL name="Task" number_of_task="2">
          <LQN:TASK task_id="Application" proc_id="AppProc"
              entry_id="Application_e" comment="m 1" />
          <LQN:TASK task_id="Buffer" proc_id="AppProc" entry_id="Buffer_e"
              comment="m 1" />
        </LQN:TASK_DECL>
      - <LQN:ENTRY_DECL name="Service" number_of_entry="2">
        - <LQN:PHASE name="Application" entry_id="Application_e">
            <LQN:PHASESTEP entry_id="Application_e" ph_ser_time="1" />
            <LQN:CALL call_name="acquire" from_entry="Application_e"
                to_entry="Buffer_e" call_type="synchronous" comment="call
                comment" />
          </LQN:PHASE>
        - <LQN:PHASE name="Buffer" entry_id="Buffer_e">
            <LQN:PHASESTEP entry_id="Buffer_e" ph_ser_time="0" />
            <LQN:CALL call_name="alloc_buff" from_entry="Buffer_e"
                to_entry="Application_dummy_e" call_type="synchronous"
                comment="call comment" />
          </LQN:PHASE>
        - <LQN:PHASE name="Application_dummy"
              entry_id="Application_dummy_e">
            <LQN:PHASESTEP entry_id="Application_e" ph_ser_time="6.75"
                />
          </LQN:PHASE>
        </LQN:ENTRY_DECL>
      </LQN:MODEL>
  </LQN:Root>
```

Buffer (CSIM)

```
<?xml version="1.0" encoding="utf-8" ?>
- <CSIM:Root xmlns:CSIM="http://www.w3.org/1999/XSL/Format">
  - <CSIM:Model modelname="buffer" comment="closeload">
    - <CSIM:HardwareFacilitySet name="Hardware">
        <CSIM:HardwareFacility name="AppProc" comment="n/a" />
      </CSIM:HardwareFacilitySet>
    - <CSIM:SoftwareFacilitySet name="Task">
      - <CSIM:SoftwareFacility name="Application_s" hardwareref="AppProc"
          comment="n/a">
          <CSIM:MailBox name="Application_mailbox"
            softwareref="Application_s" />
        </CSIM:SoftwareFacility>
      - <CSIM:SoftwareFacility name="Buffer_s" hardwareref="AppProc"
          comment="n/a">
          <CSIM:MailBox name="Buffer_mailbox" softwareref="Buffer_s" />
        </CSIM:SoftwareFacility>
      </CSIM:SoftwareFacilitySet>
    - <CSIM:ProcessSet name="Service">
      - <CSIM:Process name="Application" hardwareref="AppProc"
          softwareref="Application_s">
        - <CSIM:Step name="acquire" servicetime="0.5">
            <CSIM:To transname="acquire" target="join-1" />
          </CSIM:Step>
          <CSIM:MessageOut name="acquire" fromProcess="Application"
            toProcess="Buffer" />
        - <CSIM:Step name="use" servicetime="3.5">
            <CSIM:To transname="t3" target="fork-2" />
            <CSIM:From transname="alloc_buff" source="return_buffer"
              />
          </CSIM:Step>
        - <CSIM:Fork name="fork-2" condition="n/a">
            <CSIM:To transname="t4" target="other job" />
            <CSIM:To transname="release" target="buffer_avail" />
            <CSIM:From transname="t3" source="use" />
          </CSIM:Fork>
        - <CSIM:Step name="other job" servicetime="3.0">
            <CSIM:From transname="t4" source="fork-2" />
          </CSIM:Step>
          <CSIM:MessageOut name="release" fromProcess="Application"
            toProcess="Buffer" />
        </CSIM:Process>
      - <CSIM:Process name="Buffer" hardwareref="AppProc"
          softwareref="Buffer_s">
        - <CSIM:Join name="join-1" condition="n/a">
            <CSIM:To transname="t2" target="return_buffer" />
            <CSIM:From transname="acquire" source="acquire" />
            <CSIM:From transname="t1" source="Buffer_idle" />
          </CSIM:Join>
        - <CSIM:Step name="return_buffer" servicetime="0.5">
            <CSIM:To transname="alloc_buff" target="use" />
            <CSIM:From transname="t2" source="join-1" />
          </CSIM:Step>
```

```
            <CSIM:MessageOut name="alloc_buff" fromProcess="Buffer"
                toProcess="Application" />
          - <CSIM:Step name="buffer_avail" servicetime="0.25">
              <CSIM:From transname="release" source="fork-2" />
            </CSIM:Step>
          </CSIM:Process>
        </CSIM:ProcessSet>
    </CSIM:Model>
</CSIM:Root>
```

Test Case 7: 3-Tier Client Server (IM)

```
<?xml version="1.0" encoding="utf-8" ?>
- <IM:Model xmlns:IM="http://www.w3.org/1999/XSL/Format" name="3tierclientserver">
  - <IM:HardwareEntity name="Hardware">
      <IM:Hardware name="User Proc" taskname="User" />
      <IM:Hardware name="App Proc" taskname="Application" />
      <IM:Hardware name="DB Proc" taskname="Database" />
    </IM:HardwareEntity>
  - <IM:TaskEntity name="Task">
      <IM:Task name="User" hardwareref="User Proc" />
      <IM:Task name="Application" hardwareref="App Proc" />
      <IM:Task name="Database" hardwareref="DB Proc" />
    </IM:TaskEntity>
  - <IM:ServiceEntity name="Service">
    - <IM:Service name="User" hardwareref="User Proc" think-time="30">
      - <IM:InitialStep name="init" taskname="User" prestate="none"
            pname="User">
          <IM:To transname="t1" toStep="Send" />
        </IM:InitialStep>
      - <IM:Step name="Send" taskname="User" servicetime="3"
            prestate="none" pname="User">
          <IM:To transname="request" toStep="j1" />
          <IM:From transname="t1" fromStep="init" />
        </IM:Step>
        <IM:TransitionArc name="request" sourcetask="User"
            targettask="Application" source="Send" target="j1" arctype="request"
            calltype="synchronous" />
      - <IM:Step name="receive" taskname="User" servicetime="0.7"
            prestate="none" pname="User">
          <IM:To transname="t2" toStep="end" />
          <IM:From transname="reply" fromStep="f4" />
        </IM:Step>
      </IM:Service>
    - <IM:Service name="Application" hardwareref="App Proc">
      - <IM:Join name="j1" taskname="Application" prestate="j1"
            pname="Application">
          <IM:To transname="t4" toStep="y1" />
          <IM:From transname="t3" fromStep="idle_y" />
          <IM:From transname="request" fromStep="Send" />
        </IM:Join>
      - <IM:Step name="y1" taskname="Application" servicetime="1.5"
            prestate="j1" pname="Application">
          <IM:To transname="t5" toStep="f2" />
          <IM:From transname="t4" fromStep="j1" />
        </IM:Step>
      - <IM:Fork name="f2" taskname="Application" prestate="f2"
            pname="Application">
          <IM:To transname="t6" toStep="y2" />
          <IM:To transname="read" toStep="j5" />
          <IM:From transname="t5" fromStep="y1" />
        </IM:Fork>
      - <IM:Step name="y2" taskname="Application" servicetime="1.2"
            prestate="f2" pname="Application">
          <IM:To transname="t7" toStep="y3" />
```

```
        <IM:From transname="t6" fromStep="f2" />
      </IM:Step>
    - <IM:Step name="y3" taskname="Application" servicetime="1.1"
          prestate="f2" pname="Application">
        <IM:To transname="t8" toStep="j3" />
        <IM:From transname="t7" fromStep="y2" />
      </IM:Step>
    - <IM:Join name="j3" taskname="Application" prestate="j3"
          pname="Application">
        <IM:To transname="t9" toStep="y4" />
        <IM:From transname="t8" fromStep="y3" />
        <IM:From transname="data" fromStep="f6" />
      </IM:Join>
    - <IM:Step name="y4" taskname="Application" servicetime="0.5"
          prestate="j3" pname="Application">
        <IM:To transname="t10" toStep="f4" />
        <IM:From transname="t9" fromStep="j3" />
      </IM:Step>
    - <IM:Fork name="f4" taskname="Application" prestate="f4"
          pname="Application">
        <IM:To transname="t11" toStep="y5" />
        <IM:To transname="reply" toStep="receive" />
        <IM:From transname="t10" fromStep="y4" />
      </IM:Fork>
    - <IM:Step name="y5" taskname="Application" servicetime="1.0"
          prestate="f4" pname="Application">
        <IM:To transname="t12" toStep="undef_y" />
        <IM:From transname="t11" fromStep="f4" />
      </IM:Step>
      <IM:TransitionArc name="reply" sourcetask="Application"
          targettask="User" source="f4" target="receive" arctype="reply"
          calltype="synchronous" />
      <IM:TransitionArc name="read" sourcetask="Application"
          targettask="Database" source="f2" target="j5" arctype="request"
          calltype="synchronous" />
  </IM:Service>
- <IM:Service name="Database" hardwareref="DB Proc">
    - <IM:Join name="j5" taskname="Database" prestate="j5"
          pname="Database">
        <IM:To transname="t14" toStep="z1" />
        <IM:From transname="t13" fromStep="idle_z" />
        <IM:From transname="read" fromStep="f2" />
      </IM:Join>
    - <IM:Step name="z1" taskname="Database" servicetime="5.0"
          prestate="j5" pname="Database">
        <IM:To transname="t15" toStep="z2" />
        <IM:From transname="t14" fromStep="j5" />
      </IM:Step>
    - <IM:Step name="z2" taskname="Database" servicetime="5.0"
          prestate="j5" pname="Database">
        <IM:To transname="t16" toStep="f6" />
        <IM:From transname="t15" fromStep="z1" />
      </IM:Step>
    - <IM:Fork name="f6" taskname="Database" prestate="f6"
          pname="Database">
        <IM:To transname="t17" toStep="z3" />
```

```
            <IM:To transname="data" toStep="j3" />
            <IM:From transname="t16" fromStep="z2" />
          </IM:Fork>
        - <IM:Step name="z3" taskname="Database" servicetime="2.0"
              prestate="f6" pname="Database">
            <IM:To transname="t18" toStep="undef_z" />
            <IM:From transname="t17" fromStep="f6" />
          </IM:Step>
          <IM:TransitionArc name="data" sourcetask="Database"
              targettask="Application" source="f6" target="j3" arctype="reply"
              calltype="synchronous" />
      </IM:Service>
    </IM:ServiceEntity>
  - <IM:PatternInformation name="Pattern">
    - <IM:Pattern name="CLIENT SERVER">
        <IM:PatternAssociate name="User" />
        <IM:PatternAssociate name="Application" />
        <IM:PatternInteraction name="request" actiontype="synchronous"
            actionsource="User" actiontarget="Application" />
        <IM:PatternInteraction name="read" actiontype="synchronous"
            actionsource="Application" actiontarget="Database" />
        <IM:PatternInteraction name="data" actiontype="synchronous"
            actionsource="Database" actiontarget="Application" />
        <IM:PatternInteraction name="reply" actiontype="synchronous"
            actionsource="Application" actiontarget="User" />
      </IM:Pattern>
    - <IM:Pattern name="CLIENT SERVER">
        <IM:PatternAssociate name="Application" />
        <IM:PatternAssociate name="Database" />
        <IM:PatternInteraction name="request" actiontype="synchronous"
            actionsource="User" actiontarget="Application" />
        <IM:PatternInteraction name="read" actiontype="synchronous"
            actionsource="Application" actiontarget="Database" />
        <IM:PatternInteraction name="data" actiontype="synchronous"
            actionsource="Database" actiontarget="Application" />
        <IM:PatternInteraction name="reply" actiontype="synchronous"
            actionsource="Application" actiontarget="User" />
      </IM:Pattern>
    </IM:PatternInformation>
</IM:Model>
```

3-Tier Client Server (LQN)

```
<?xml version="1.0" encoding="utf-8" ?>
- <LQN:Root xmlns:LQN="http://www.w3.org/1999/XSL/Format">
  - <LQN:MODEL modelname="3tierclientserver" comment="G 0.0000001 100 1 0.9 -
        1">
    - <LQN:HW_DECL name="Hardware" number_of_processor="3">
        <LQN:HARDWARE proc_id="User Proc" comment="f i" />
        <LQN:HARDWARE proc_id="App Proc" comment="f i" />
        <LQN:HARDWARE proc_id="DB Proc" comment="f i" />
      </LQN:HW_DECL>
    - <LQN:TASK_DECL name="Task" number_of_task="3">
        <LQN:TASK task_id="User" proc_id="User Proc" entry_id="User_e"
            comment="m 1" />
```

```
        <LQN:TASK task_id="Application" proc_id="App Proc"
            entry_id="Application_e" comment="m 1" />
        <LQN:TASK task_id="Database" proc_id="DB Proc"
            entry_id="Database_e" comment="m 1" />
      </LQN:TASK_DECL>
    - <LQN:ENTRY_DECL name="Service" number_of_entry="3">
      - <LQN:PHASE name="User" entry_id="User_e" think-time="30">
          <LQN:PHASESTEP entry_id="User_e" ph_ser_time="3.7" />
          <LQN:CALL call_name="request" from_entry="User_e"
              to_entry="Application_e" call_type="synchronous"
              comment="call comment" />
        </LQN:PHASE>
        <LQN:A_DECL task_id="Application" entry_id="Application_e"
            activity_id="init" />
      - <LQN:ACTIVITY task_id="Application" entry_id="Application_e"
            activity_id="init">
          <LQN:JOIN name="j1" join_type="and" />
          <LQN:ACTIVITYSTEP activity="y1" serv_time="1.5" />
          <LQN:FORK name="f2" fork_type="and" />
          <LQN:ACTIVITYSTEP activity="y2" serv_time="1.2" />
          <LQN:ACTIVITYSTEP activity="y3" serv_time="1.1" />
          <LQN:JOIN name="j3" join_type="and" />
          <LQN:ACTIVITYSTEP activity="y4" serv_time="0.5" />
          <LQN:FORK name="f4" fork_type="and" />
          <LQN:ACTIVITYSTEP activity="y5" serv_time="1.0" />
          <LQN:CALL call_name="read" from_entry="Application_e"
              to_entry="Database_e" call_type="synchronous"
              comment="call comment" />
        </LQN:ACTIVITY>
      - <LQN:PHASE name="Database" entry_id="Database_e">
        - <LQN:PHASESTEP entry_id="Database_e" ph_ser_time="10">
            <LQN:PHASESTEP entry_id="Database_e" ph_ser_time="2"
                />
          </LQN:PHASESTEP>
        </LQN:PHASE>
      </LQN:ENTRY_DECL>
    </LQN:MODEL>
  </LQN:Root>
```

3-Tier Client Server (CSIM)

```
  <?xml version="1.0" encoding="utf-8" ?>
- <CSIM:Root xmlns:CSIM="http://www.w3.org/1999/XSL/Format">
    - <CSIM:Model modelname="3tierclientserver" comment="closeload">
      - <CSIM:HardwareFacilitySet name="Hardware">
          <CSIM:HardwareFacility name="User Proc" comment="n/a" />
          <CSIM:HardwareFacility name="App Proc" comment="n/a" />
          <CSIM:HardwareFacility name="DB Proc" comment="n/a" />
        </CSIM:HardwareFacilitySet>
      - <CSIM:SoftwareFacilitySet name="Task">
        - <CSIM:SoftwareFacility name="User_s" hardwareref="User Proc"
            comment="n/a">
            <CSIM:MailBox name="User_mailbox" softwareref="User_s" />
          </CSIM:SoftwareFacility>
```

```
    - <CSIM:SoftwareFacility name="Application_s" hardwareref="App Proc"
          comment="n/a">
        <CSIM:MailBox name="Application_mailbox"
            softwareref="Application_s" />
      </CSIM:SoftwareFacility>
    - <CSIM:SoftwareFacility name="Database_s" hardwareref="DB Proc"
          comment="n/a">
        <CSIM:MailBox name="Database_mailbox"
            softwareref="Database_s" />
      </CSIM:SoftwareFacility>
  </CSIM:SoftwareFacilitySet>
- <CSIM:ProcessSet name="Service">
    - <CSIM:Process name="User" hardwareref="User Proc"
          softwareref="User_s">
      - <CSIM:Step name="Send" servicetime="3">
          <CSIM:To transname="request" target="j1" />
          <CSIM:From transname="t1" source="init" />
        </CSIM:Step>
        <CSIM:MessageOut name="request" fromProcess="User"
            toProcess="Application" />
      - <CSIM:Step name="receive" servicetime="0.7">
          <CSIM:To transname="t2" target="end" />
          <CSIM:From transname="reply" source="f4" />
        </CSIM:Step>
      </CSIM:Process>
    - <CSIM:Process name="Application" hardwareref="App Proc"
          softwareref="Application_s">
      - <CSIM:Join name="j1" condition="n/a">
          <CSIM:To transname="t4" target="y1" />
          <CSIM:From transname="t3" source="idle_y" />
          <CSIM:From transname="request" source="Send" />
        </CSIM:Join>
      - <CSIM:Step name="y1" servicetime="1.5">
          <CSIM:To transname="t5" target="f2" />
          <CSIM:From transname="t4" source="j1" />
        </CSIM:Step>
      - <CSIM:Fork name="f2" condition="n/a">
          <CSIM:To transname="t6" target="y2" />
          <CSIM:To transname="read" target="j5" />
          <CSIM:From transname="t5" source="y1" />
        </CSIM:Fork>
      - <CSIM:Step name="y2" servicetime="1.2">
          <CSIM:To transname="t7" target="y3" />
          <CSIM:From transname="t6" source="f2" />
        </CSIM:Step>
      - <CSIM:Step name="y3" servicetime="1.1">
          <CSIM:To transname="t8" target="j3" />
          <CSIM:From transname="t7" source="y2" />
        </CSIM:Step>
      - <CSIM:Join name="j3" condition="n/a">
          <CSIM:To transname="t9" target="y4" />
          <CSIM:From transname="t8" source="y3" />
          <CSIM:From transname="data" source="f6" />
        </CSIM:Join>
      - <CSIM:Step name="y4" servicetime="0.5">
          <CSIM:To transname="t10" target="f4" />
```

```
            <CSIM:From transname="t9" source="j3" />
          </CSIM:Step>
        - <CSIM:Fork name="f4" condition="n/a">
            <CSIM:To transname="t11" target="y5" />
            <CSIM:To transname="reply" target="receive" />
            <CSIM:From transname="t10" source="y4" />
          </CSIM:Fork>
        - <CSIM:Step name="y5" servicetime="1.0">
            <CSIM:To transname="t12" target="undef_y" />
            <CSIM:From transname="t11" source="f4" />
          </CSIM:Step>
          <CSIM:MessageIn name="reply" fromProcess="Application"
              toProcess="User" />
          <CSIM:MessageOut name="read" fromProcess="Application"
              toProcess="Database" />
        </CSIM:Process>
      - <CSIM:Process name="Database" hardwareref="DB Proc"
            softwareref="Database_s">
        - <CSIM:Join name="j5" condition="n/a">
            <CSIM:To transname="t14" target="z1" />
            <CSIM:From transname="t13" source="idle_z" />
            <CSIM:From transname="read" source="f2" />
          </CSIM:Join>
        - <CSIM:Step name="z1" servicetime="5.0">
            <CSIM:To transname="t15" target="z2" />
            <CSIM:From transname="t14" source="j5" />
          </CSIM:Step>
        - <CSIM:Step name="z2" servicetime="5.0">
            <CSIM:To transname="t16" target="f6" />
            <CSIM:From transname="t15" source="z1" />
          </CSIM:Step>
        - <CSIM:Fork name="f6" condition="n/a">
            <CSIM:To transname="t17" target="z3" />
            <CSIM:To transname="data" target="j3" />
            <CSIM:From transname="t16" source="z2" />
          </CSIM:Fork>
        - <CSIM:Step name="z3" servicetime="2.0">
            <CSIM:To transname="t18" target="undef_z" />
            <CSIM:From transname="t17" source="f6" />
          </CSIM:Step>
          <CSIM:MessageIn name="data" fromProcess="Database"
              toProcess="Application" />
        </CSIM:Process>
      </CSIM:ProcessSet>
    </CSIM:Model>
  </CSIM:Root>
```