AUTOMATIC TRANSFORMATION OF UML SOFTWARE SPECIFICATION INTO LQN PERFORMANCE MODELS USING GRAPH GRAMMAR TECHNIQUES

By

Hoda Amer

A thesis submitted to the faculty of Graduate Studies
in partial fulfillment of the requirements for the degree of
Masters of Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
Ottawa, Canada
May, 2001

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

AUTOMATIC TRANSFORMATION OF UML SOFTWARE SPECIFICATION INTO LQN PERFORMANCE MODELS USING GRAPH GRAMMAR TECHNIQUES

Submitted by

Hoda Amer

in partial fulfillment of the requirements for the degree of
Masters of Engineering

**J. W. Chinneck, Chair Department of
Systems and Computer Engineering**

D. C. Petriu, Thesis Supervisor

Carleton University
May 2001

To My Mother and in Memory of

My Father

ABSTRACT

The interest in relating Software Architecture specification to Software Performance Analysis has been growing rapidly in the past few years. However, there is still a gap between the software development and the performance analysis domains. The main reasons for having this gap has been the lack of a simple, inexpensive, and scalable techniques and tools for building models of complex, distributed and real-time software systems, and for solving these models to obtain the systems performance properties. Automated techniques are therefore needed to ease the process of building and solving performance models.

In this thesis, a systematic methodology is developed to automatically translate Unified Modeling Language (UML) software specification into Layered Queueing networks (LQN) performance models using Graph Grammar techniques. UML was chosen as the input language because it is a standard, widely accepted notation for software systems specification. LQN was chosen as the output language since it is very appropriate for modeling and analyzing the performance of layered systems, both at the software and hardware levels. The Graph Grammar techniques were used since both the input and output models have well defined graphical representations. Finally, a case study is developed to verify the transformation process.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOLEDGEMENT

I am deeply indebted to my supervisor, Professor Dorina Petriu, for her encouragement and friendship throughout this research. Her continuous support, patience, and guidance have been enormous and have been a major factor in the success of this work. I'd like also to thank Professor Murray Woodside, who taught me a lot about Performance Analysis and raised my interest in the field. His nice remarks have always been encouraging to me.

My family has been very supportive to me throughout the years. I am very grateful to my husband Walid, for his moral support and constant encouragement. He and my son, Hazem were very patient throughout many days and weekends where they missed having lots of fun because Mom had some work to do. I am so grateful to my mother and all my sisters and brothers. Their prayers were also a major factor in the success of my work. I would like to dedicate this thesis to the memory of my father, Abdallah Amer. His dedication to raising us in the best way he could is the reason my brothers, sisters, and myself are what we are today.

I would like to thank Xin Wang for giving me a quick boost into PROGRES. Also, thanks to Istabrak Abdul Fatah for all his help.

Financial assistance provided by Nortel Networks and Rational software was greatly appreciated.

GLOSSARY

CORBA	Common Object Request Broker Architecture
F-ORB	Forwarding ORB
H-ORB	Handle-Driven ORB
LQN	Layered Queueing Networks
ORB	Object Request Broker
P-ORB	Process Planner ORB
PROGRES	Programmed Graph Rewriting System
QN	Queueing Networks
SPE	Software Performance Engineering
UML.	Unified Modeling Language

1 INTRODUCTION

Software Performance Engineering (SPE) has evolved over the past years and has been demonstrated to be effective during the development of many large systems [Smith-90]. Although the need for SPE is generally recognized by the industry, there is still a gap between the software development and the performance analysis domains. The main reasons for this gap has been the lack of a simple, inexpensive, and scalable techniques and tools for building performance models of complex, distributed and real-time software systems, and for solving these models to obtain the systems performance properties.

In the current practice, constructing performance models of complex systems is expensive to develop and validate. To construct performance models, analysts inspect, analyze and translate "by hand" software specifications into models, then solve these models under different workload factors in order to diagnose performance problems and recommend design alternatives for performance improvement. This performance analysis cycle, when done properly starting at the early stages of design and continuing through all software development stages is 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 undeniable. Automation is an important factor in accelerating the development of the 20th and the 21st civilization so quickly. Many processes when automated

become cost-effective by consuming less time and effort, and consequently, less money. If Software Performance models were to be generated automatically, software designers will not find it hard to employ in their software development cycles, thus bridging the gap between software development and software performance analysis domains.

1.1 Objectives

This work was done with the objective of developing a systematic methodology to automatically translate software specifications written in the Unified Modeling Language (UML) into Layered Queueing networks (LQN) performance models using Graph Grammar techniques.

UML [OMG-99] was chosen as the language used for the input software specification since it is now a standard, widely accepted notation for software systems specification. LQN models [Woodside-89, 95] were chosen as target output models. LQN was developed especially for modeling complex concurrent and/or distributed software systems, and was proven useful for providing insights into performance characteristics at software and hardware levels. In addition, an efficient LQN toolset, which includes both simulation and analytical solvers, is available for solving the generated models.

This transformation was done using Graph Grammar techniques. A programming tool named PROGRES (Programmed Graph Rewriting System) was used for this purpose [Schürr-94, 99]. The PROGRES Language was found suitable to use since both the UML and LQN models have well defined graphical representations.

1.2 Contributions of the thesis

The thesis proposes a method to convert a UML specification of a software design, augmented with quantitative performance annotations, into a LQN performance model. The structure of the LQN model (i.e., its tasks and devices) is obtained from the high-level software architecture and deployment information showing the allocation of software components to physical devices. The high-level software architecture is described by UML Collaboration diagrams that show the relationship between concurrent components (i.e., processes or threads represented in UML as active objects). The allocation of software to hardware devices is described by UML Deployment diagrams.

The entries, phases, and activities of LQN tasks and their quantitative parameters are obtained from UML behavioral descriptions of a few, well chosen representative scenarios. UML Activity diagrams with swimlanes are used to represent these scenarios. The Activity diagrams are specially built to show the scenario steps executed by different concurrent components, and to emphasize the flow of control and object flow between concurrent components in the system. In general, software designers prefer to use interaction diagrams (Sequence or Collaboration) to describe the behaviour of the system and the interaction between objects, rather than Activity diagrams, which show the flow control of activities but not the responsibilities of every individual object. Therefore, the proposed method constructs automatically the Activity diagrams used for LQN derivation from Sequence diagrams. Quantitative performance annotations were added to the UML diagrams to specify information such as processors and link speed, processes multiplicity, and expected CPU demands for every scenario step. The following figure summarizes the proposed method.

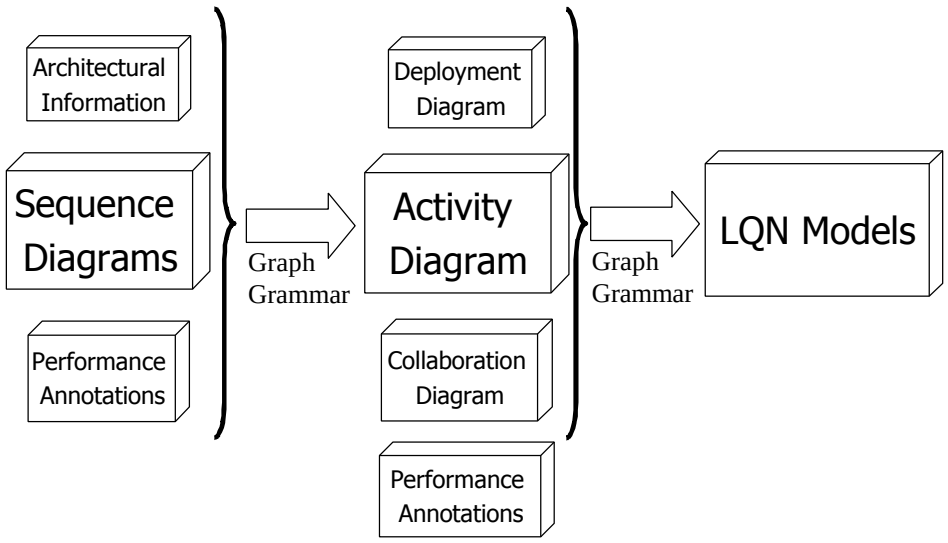


Figure 1: Summary of the proposed method of automatic transformation.

The proposed approach was implemented as a stand-alone PROGRES program, which represents a “proof of concept” for the transformation from UML to LQN. The PROGRES program runs in two phases. In the first phase of the graph transformation process, the Activity diagrams that show the flow of control for the chosen scenarios from Sequence diagrams and architectural information are obtained. In the second phase of the graph transformation, the LQN model is derived from Activity diagrams, deployment information and performance annotation. A PROGRES program requires a schema that describes the types of nodes and edges in the graphs manipulated by the transformation program. A PROGRES schema was proposed in the thesis that represents both the UML and LQN model elements. The UML part of the schema is similar to the UML metamodel.

Finally, a case study was developed to validate and verify the transformation process. In a previous work by [Abd-97, 98a + b], three different software architectures of a distributed software systems were developed and measurements of performance parameters under various workload factors were recorded. In this thesis, the corresponding LQN models for these architectures are developed based on the UML specification describing them and the models are solved using the LQN solver presented in [Franks-95]. The “Activities” feature of the LQN models enabled us to model parallelism in one of the studied architectures (namely the P-ORB case). The measurements of the total response time of the three architectures are then compared to the results obtained by solving their corresponding models.

1.3 Thesis Contents

This thesis is organized as follows:

Chapter 2 provides an overview of the background information on this thesis, such as Software Performance Engineering (SPE), Layered Queueing Networks (LQN), the Unified Modeling Language (UML), and Programmed Graph Rewriting System (PROGRES). It also reviews some of the works that were recently done in transforming software specifications written in UML into various kinds of performance models.

Chapter 3 describes the first phase of the graph transformation process, producing Activity Diagrams with the objective of identifying the flows of control for the chosen scenarios. The PROGRES schema and transformation rules specific to this phase are also explained.

Chapter 4 describes the second phase of the graph transformation process, where the LQN models are built. The PROGRES schema and transformation rules specific to this phase are also explained.

Chapter 5 presents a case study that is developed to validate and verify the transformation process, where the measurements of the total response time of three different architectures of a distributed software systems are compared to the results obtained by solving their generated models.

Chapter 6 concludes the thesis research, summarizes contributions of the thesis, and identifies directions for future research.

2 LITERATURE REVIEW

2.1 Software Performance Engineering (SPE)

Software Performance Engineering (SPE) is a technique introduced in [Smith-90]. It proposes the use of quantitative methods and performance models in order to assess the performance effects of different design and implementation alternatives. It is applied starting from the earliest stages of software development throughout the whole lifecycle. Most current practices in software design and implementation are based on a "design now and fix performance later" approach. That is, the functional design and implementation of the system are done first and the performance techniques are retrofitted at a later point in time. In many situations the prototype fails to meet the performance requirements resulting in an expensive redesign of the system. Analytic performance models are often used in software performance engineering (SPE) because of its lower cost in comparison to simulation and measurement-based approaches. Analytic models are also used in system selection studies and in capacity planning [Menasce-94].

Performance modeling, if done in the early design stages, can reduce the risk of performance related failures by giving an early warning of problems. They also provide performance predictions under varying environmental conditions or design alternatives. Solving the performance model yields several values; for instance, the total response time and the utilization for each server in the system. The performance model solution is then compared

with the requirements and the 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.

In the early development stages, the input parameter values to an analytic model are estimates based on previous experience with similar systems, on measurement of reusable components, on known platform overheads (such as system call execution times), and on time budgets allocated to different components. As the development progresses and more components are implemented and measured, the model parameters become more accurate and so do the results. In [Smith-90], 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.

As an application of SPE, a case study on how to apply the techniques of Software Performance Engineering to Web applications is introduced in [Smith-00]. When developing a web application, both responsiveness and scalability are very important design goals. This paper focuses on using SPE techniques to construct and evaluate performance models of various architectural alternatives early in development, with the goal of selecting a combination that will meet performance objectives. Since the Web execution environment is typically complex, simple models of software processing that are easily constructed and solved were deliberately used to provide feedback on whether the proposed software is likely to meet performance goals. This was achieved by employing an approximation technique that provides the most important information while keeping the models themselves simple.

This solution provided an estimate of the end-to-end response time and the scalability of the Web application by studying how it performs under different workload conditions.

2.2 Layered Queueing Network (LQN)

Queueing network modeling is a popular and widely used technique for predicting the performance of computing systems. Although queueing network models have been successfully used in the context of traditional time-sharing computers, they often fail to capture complex interactions among various software and hardware components in client-server distributed processing systems. The Layered Queueing Networks (LQN) [Woodside-89], [Woodside-95], [Franks-95] and the Method of Layers [Rolia-95] are examples of new modeling techniques that were developed for handling such complex interactions.

LQN is a new adaptation of queueing models for systems with software and hardware servers and resources. A model in LQN is closely linked to software specifications, which makes it easy to develop and understand. It is well suited for systems with parallel processes running on a multiprocessor or on a network, such as client-server systems.

An LQN model is represented as an acyclic graph whose nodes (named also *tasks*) are software entities and hardware devices, and whose arcs denote service requests (See Figure 2). Tasks represent hardware or software objects that may execute concurrently. In LQN models, tasks are classified into three categories; namely *pure clients* (also named *reference tasks,* as they drive the system), *pure servers,* and *active servers*. While *pure clients* can only send messages (requests) and *pure servers* can only receive requests, *active server*s can both send and receive requests. This marks the main difference between LQN and QN, where *active servers,* to which requests are arriving and queueing for service, may become clients to

other servers as well. This gives rise to nested services. 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 [Petriu-00b].

The LQN toolset presented in [Franks-99], which includes both simulation and analytical solvers, is used in solving the generated LQN performance models of this work.

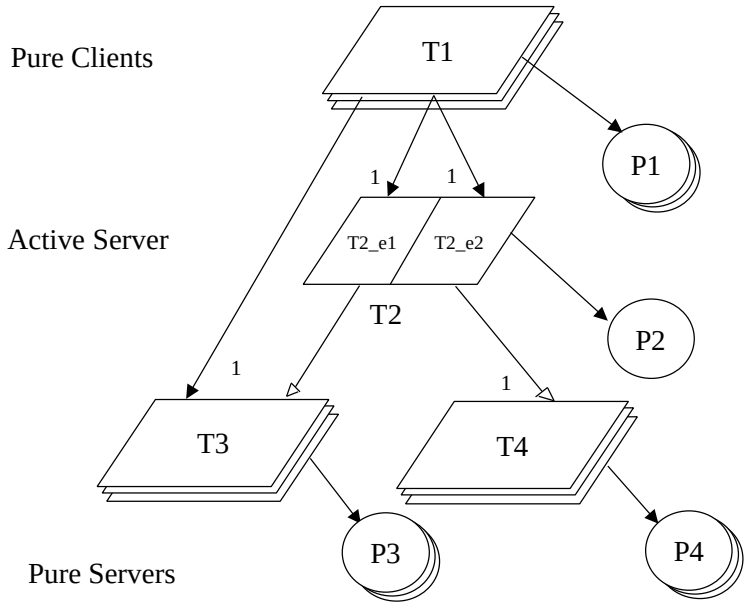


Figure 2: Example of an LQN Model

2.2.1 LQN Components

2.2.1.1 Tasks and Processors (Server Nodes)

As mentioned before, server nodes, which can be either tasks or processors, are classified into three categories; namely *pure clients*, *pure servers*, and *active servers*. They are typically used as follows:

- *pure clients,* which only send messages (requests), are used to model actual users and other input sources.

- *pure servers,* which only receive requests, are generally used to model hardware devices such as processors or disks.

- *active servers,* which can receive requests as well as make their own, 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 their turn to be served. The default scheduling policy of the request queue is FIFO, 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 clones 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. The tasks are drawn as parallelograms, and the processors as circles (see Figure 3 for illustration).

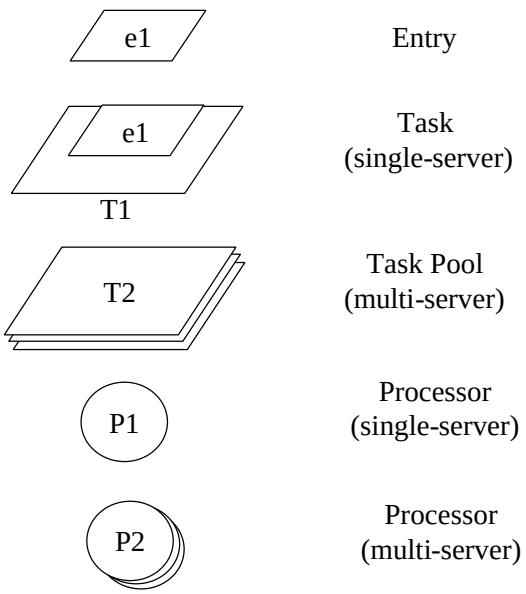


Figure 3: Task and Entry Graphical Notation

2.2.1.2 Entries

An LQN task may offer more than one kind of service, each modeled by a so-called *entry* drawn as a parallelogram "slice" (see Figure 3 for illustration). An entry is like a port or an address of a particular service offered by a task. An entry 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 activities if more details are required to describe its execution scenario. This is typically required when entries have fork and join interactions. Activities are discussed next.

2.2.1.3 Activities

Activities are components that represent the lowest level of detail in the 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 its task runs or have zero service time. Just like entries, activities can make requests to other tasks by way of *synchronous* or *asynchronous* messages (see Figure 4 for illustration).

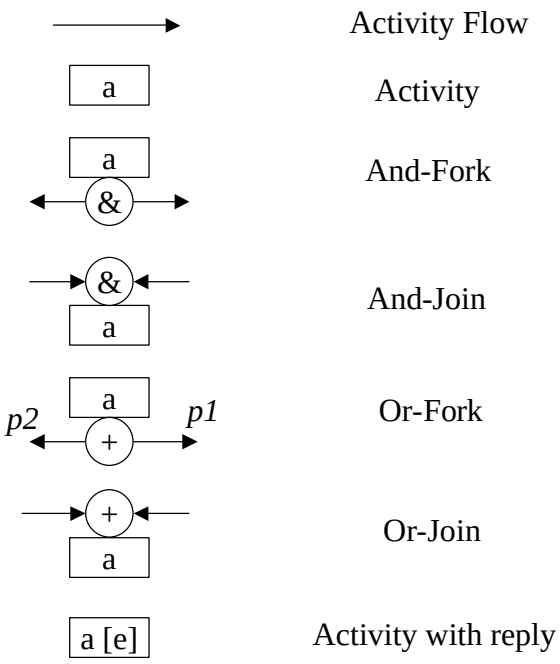
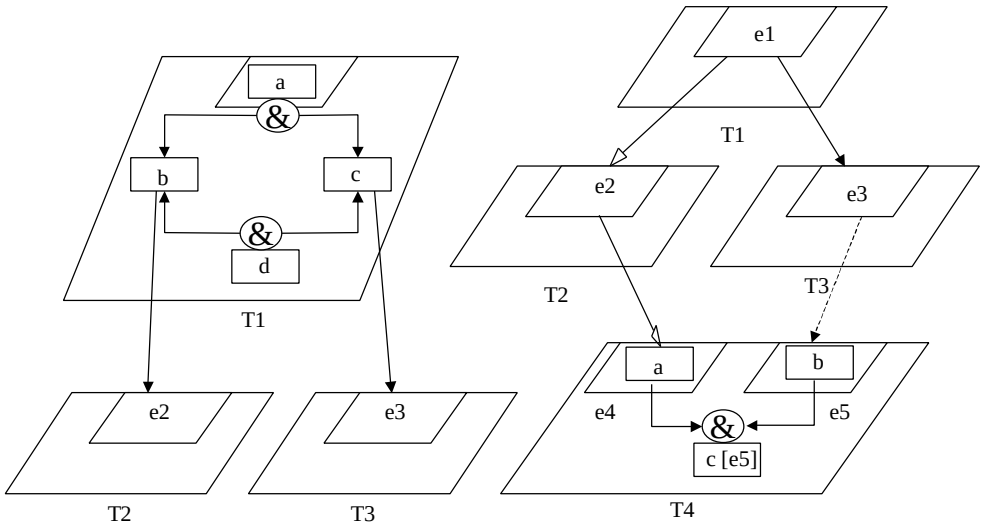


Figure 4: Activity Graphical Notation

After an AND-fork, all successor activities can execute in parallel, while after an OR-fork, only one of the successor activities is executed, with probability *Pi*. Sequential execution is a special case of an OR-fork with only one branch. Joins happens when multiple threads of control are connected together. A special case of joins is the AND-joins since they introduce synchronization delays [Franks-99].

Forking happens when a thread of control splits into two or more concurrent sub-threads, while joining happen when two or more tasks synchronize with one another. There are two forms of fork-join behaviour that are based on whether the fork and join take place within the same task, or in two separate tasks; namely *intra-task fork-join* and *inter-task fork-join* (see Figure 5 for illustration).

Intra-task fork-join behaviour occurs when the fork and join take place within the same task. This pattern is particularly useful for improving performance if parallelism in an application can be exploited. With inter-task fork-join, messages originate from a common client task, follow independent routes, and then join at another server task [Franks-99].



(a) Intra-task Fork-Join (b) Inter-task Fork-Join

Figure 5: Intra VS Inter task Fork-Join

2.2.1.4 Arcs (Requests)

Arcs in an LQN model denote requests from one entry to another. The labels on the arcs denote the average number of requests made each time the corresponding phase in the source entry 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* [Petriu-00b] (see Figure 6 for illustration).

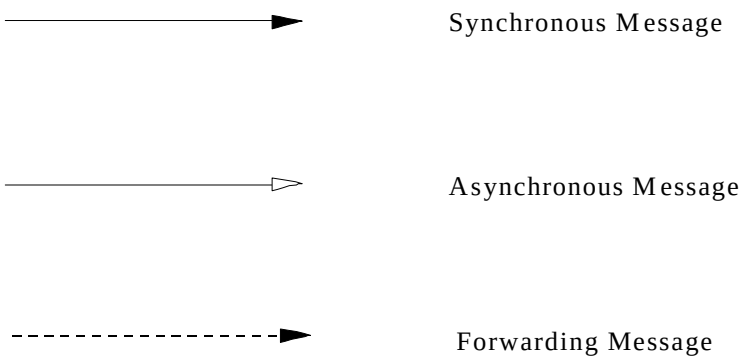


Figure 6: Messages (requests) Graphical Notation

- 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 entry. At the end of *phase 1*, the server replies to the client, which is unblocked and continues its work. The server continues with the following phases, if any, working in parallel with the client, until the completion of the last phase. After finishing the last phase, the server begins to serve a new request from the queue, or becomes idle if the queue is empty. During any phase, the server may act as a client to other servers.

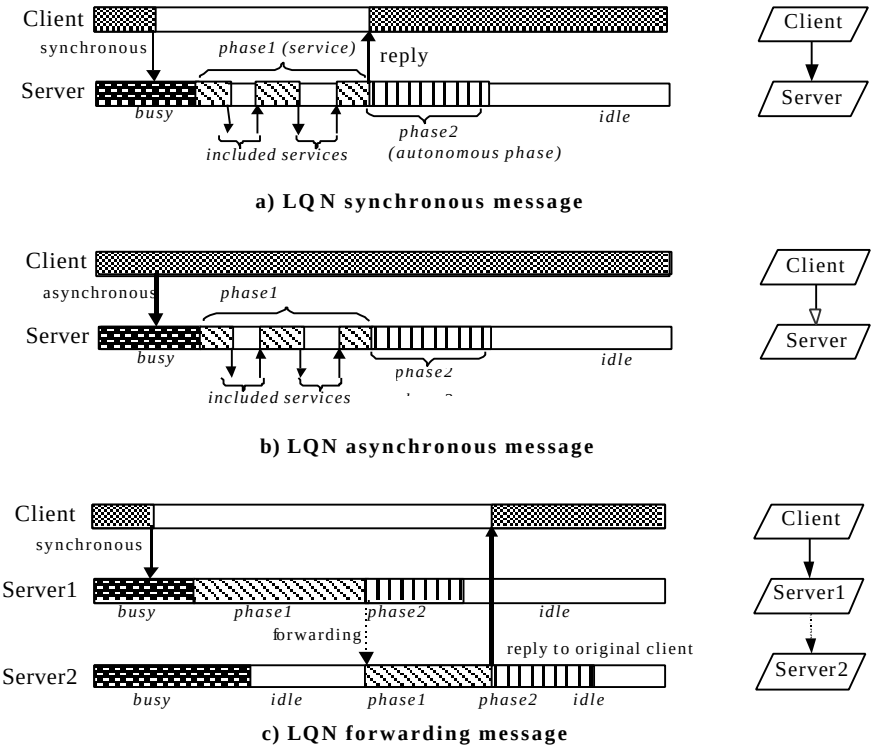


Figure 7: Execution of synchronous, asynchronous, and forwarding LQN requests

- In the case of *asynchronous* message, the client does not block after sending the message and the server does not reply back; instead only executing its phases as shown in Figure 7.

- The *forwarding* message (represented by a dotted request arc) is associated 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, then at the end of its last phase starts another cycle. 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.

A phase may be *deterministic* or *stochastic*, and is subject to the following assumptions [Petriu-00a]:

- The total CPU demand of a phase (whose mean is given as a parameter) 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 (given as a parameter) in stochastic phases, and occur for a fixed number of times in a deterministic phase.

2.2.2 LQN parameters

The parameters of an LQN model are as follows:

- Customer (client) classes and their associated populations or arrival rates;

- The number of processor nodes and the task allocation on them;

- The multiplicity of each task or processor node in the system;

- Scheduling discipline for each software and hardware server;

- The mean service time demand per visit for each activity and/or entry per phase;

- The mean number of synchronous, asynchronous, or forwarded messages sent from entry/or activity to another per phase.

2.2.3 Solving the LQN Model

LQN models can be solved using the solving tools provided by the toolset in [Franks-95]. Typical results of an 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 [Neilson-95].

LQN was developed especially for modeling complex concurrent and/or distributed software systems. LQN was applied to a number of concrete industrial systems and was proven useful for providing insights into performance limitations at software and hardware levels.

2.3 The Unified Modeling Language (UML)

2.3.1 A Visual Modeling Language

As mentioned in [Booch-99] and [Quatr-00], a language provides the vocabulary and the rules used to combine words in that vocabulary for the purpose of communication. A modeling language is a language whose vocabulary and rules are used to represent both the conceptual and the physical aspects of a system. The vocabulary and rules of a modeling language tell you how to create and read models. The Unified Modeling Language (UML) is a graphical modeling language that is used for visualizing, specifying constructing and documenting software systems. It is a language for expressing system properties that are best modeled graphically so that software developers can visualize their work products in standardized blueprints or diagrams.

Models are abstractions of real systems. They help us visualize and understand complex systems by highlighting the essentials of complex systems and filtering out nonessential details, thus making the system easier to understand. Notation plays an important part in any model. It serves as the language (vocabulary and rules) for communication between model designers. The unified Modeling Language (UML) provides a very robust notation, which grows from analysis into design.

2.3.2 Designing The System Architecture

Software architecture is very difficult to define, as mentioned in [Booch-99] and [Quatr-00]. Software architecture is not a one-dimensional thing. It is made up of concurrent multiple views. It can be viewed as a set of strategic decisions about the structure and behavior of the

system and the collaboration among the system elements; however, it is not merely that. It is stated in [Booch-99] that:

> "Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and tradeoffs and aesthetic concerns."

According to [Booch-99], the architecture of a software system can be best described by five interactive views, each focusing on a particular aspect of the system.

The *use case view* of a system includes the use cases that describe the behavior of the system as seen by its end users, analysts and testers. It specifies the forces that shape the system's architecture.

The *design view* of a system addresses the functional requirements of the system, which are the services that the system should provide to its end users. It includes classes, interfaces, and collaborations that form the software system.

The *process view* of a system focuses on the runtime implementation of the system. It includes the threads and processes that form the system's concurrency and synchronization mechanisms. It takes into account requirements such as performance, reliability, scalability, integrity, system management, and synchronization.

The *implementation view* of a system is concerned with the software module organization within the development environment. It includes the components and files that can be assembled in various ways to produce a running system.

The *deployment view* of a system involves mapping software components to processing nodes that form the system's hardware topology on which the system is expected to run. The deployment view takes into account the physical system parts requirements such as their distribution, delivery, and installation.

2.3.3 The UML Building Blocks

The building blocks (vocabulary) of UML include three categories as described in [Jacob-98] and [MOD-00]: elements, relationships and diagrams. Here is a short description of each category:

- *Elements* are the basic object-oriented building blocks used to construct models. The four kinds of elements are Structural, Behavioral, Grouping and Annotational.
    - *Structural elements*, such as Classes, Active Classes, Interfaces, Usecases, Collaborations, Components and Nodes, identify the entities that are to be modeled.
    - *Behavioral Elements*, such as State machines and Interactions, represent the dynamic parts of the model and describe the changes in a system's state over time.
    - *Grouping elements*, such as Packages, Models, Subsystems, and Frameworks, are used for organizing parts of a model.
    - *Annotational elements*, such as Notes, are used to describe or annotate elements in a diagram.
- *Relationships* are used to join elements together in a model.

Within this category we find four kinds of relationships:

- *Generalization* is the relationship between a general element, called the super class, and a more specific element, called the subclass.
- *Association* is the relationship used to connect one element to another. Aggregation is a special form of association that specifies the whole-part relationship. Composition is another form of association, with stronger relationship between the whole and the part.

- *Dependency* specifies the relationship between one element and the elements that will be affected if its specification is changed.

- *Realization* defines the relationship between an interface or a use case and the classifier that will implement their behavior. Realization is a form of generalization, in which only behavior is inherited.

- *Diagrams* assemble related collections of elements together representing all or part of a model. The static parts of a system use one of four structural diagrams, namely Class, Object, Component, and Deployment diagrams, while the dynamic parts of a system use one of five behavioral diagrams, namely Use case, Sequence, Collaboration, State chart, and Activity diagrams.

    - *Class diagrams* are the most commonly used diagrams in modeling Object-Oriented systems. A Class diagram shows a set of classes, interfaces, and collaborations and their relationships to illustrate the static design view of a system.

    - *Object diagrams* show sets of objects and their relationships. They are used to illustrate static snapshots of instances of classes in different situations.

    - *Component diagram*s show sets of components and their relationships. They are used to illustrate the static implementation view of a system.

    - *Deployment diagrams* show sets of nodes and their relationships, and are used to illustrate the deployment view of a system.

    - *Use Case diagrams* show sets of use cases and actors and their relationships. They present an outside view of the system. The flow-of-events capture the functionality of the use case, while scenarios are used to describe how use cases are realized by identifying the objects, the classes, and the object interactions needed to carry out a piece of the functionality specified by the use case.

    - *Sequence diagrams* are interaction diagrams that show sets of objects and the messages sent and received by those objects, with emphasis on the time ordering of messages. Sequence diagrams are semantically equivalent to collaboration diagrams.

    - *Collaboration diagrams* are interaction diagrams that show sets of objects, links, and messages sent and received by these objects. They emphasize the structural organization of their objects. Collaboration diagrams are semantically equivalent to Sequence diagrams.

- *State diagrams* show sets of states, transitions, events, and activities that together constitute state machines. State diagrams are used to model the behavior of an instance, class, or collaboration. State diagrams are semantically equivalent to Activity diagrams; however, they emphasize the event-ordered behavior of an object.

- *Activity diagrams* show sets of activities, the sequential or branching flow from one activity to another, and objects that act and are acted upon. Activity diagrams are semantically equivalent to State diagrams; however, they emphasize the flow of control among objects.

2.3.4 The Metamodel

The architecture of the UML, as stated in [OMG-99], is based on a four-layer metamodeling architecture that forms the infrastructure for defining the precise semantics required by complex models. This architecture consists of the following layers: meta-metamodel, metamodel, model, and user objects.

The meta-metamodeling layer forms the foundation for the metamodeling architecture. This layer's basic responsibility is to define the language for specifying the metamodel. Examples of meta-meta-objects in the meta-metamodeling layer are: MetaClass, MetaAttribute, and MetaOperation.

A metamodel is an instance of a meta-metamodel. This layer's basic responsibility is to define a language for specifying models. Examples of meta-objects in the metamodeling layer are: Class, Attribute, and Operation.

A model is an instance of a metamodel. The Model layer's basic responsibility is to define a language that describes an information domain. Examples of objects in the modeling layer are: Client, Broker, SendRequest, and ReceiveReply.

User objects (user data) are an instance of a model. The basic responsibility of the user objects layer is to describe a specific realization of the model. Examples of objects in the user objects layer would be the instances created of object models or the current value of an attribute, such as <Client-354> and 654.56.

In this work, we are particularly interested in the metamodel layer that describes the language formally used to specify both sequence diagrams and activity diagrams. These metamodel specifications will be used in setting the rules of the two diagrams transformation into LQN models.

2.4 Transformations from UML

The interest in relating Software Architecture specification to Software Performance Analysis has been growing rapidly in the past few years. Combining Software Architecture specification with a software performance model enables software designers to compare design alternatives, to test whether or not their software meats its intended performance restrictions, and to avoid potential problems. Various approaches have been proposed to derive a performance model from the Software Architecture specification. Many of these approaches consider the Unified Modeling Language (UNL) as a specification language since UML is now a widely accepted notation for specification of software systems. In this section, some of these approaches will be presented.

In [Petriu-98, Petriu-99] and [Wang-99], a methodology that identifies frequently used architectural patterns is presented. The methodology can be described as a formal approach for generating LQN performance models from software architectural patterns using Graph Grammar. The study describes patterns by their structure and behaviour. It then shows their

corresponding performance models. This study considers a significant set of architectural patterns, such as pipelines and filters, client-server, broker, layers, critical sections and master-slave patterns. Since there is a direct correspondence between the single pattern and the performance model, the target model is immediately given. This approach specifies the patterns using UML–Collaborations, which represents a society of classes, interfaces, and other elements that work together to provide some cooperative behaviour that is bigger than the sum of all of parts. It then derives their performance models using a systematic approach. This approach follows the SPE methodology and generates the performance models by applying graph transformation techniques using the PROGRES tool.

The input to the tool is each architectural pattern's elements represented in the Graph Grammar language (the graph schema). Transformation rules were executed for each architectural pattern found in the input architectural description graph, and the input graph transforms into an output graph that represents an LQN model in graph schema. Performance attributes, such as the allocation of processes to processors, the average execution time for each software component; the average demands for resources and the network communication delays were added to the system in the form of annotations.

In [Andolfi-00], Software specifications are specified using Message Sequence Charts, which are Sequence diagrams in the UML terminology. The proposed transformation methodology is based on a systematic analysis of the Sequence diagrams, which allows singling out the real degree of parallelism among the Software Architecture components and their dynamic dependencies. This information is then used to automatically build a QN Model corresponding to the Software Architecture description. This approach defines the QN

Model by the analysis of a Labeled Transition System associated with the Sequence diagram, and is only interested in analyzing the software architecture, without specifying or using information regarding the underlying hardware platform.

The method proposed in [Corte-00] is based on the Software Performance Engineering (SPE) methodology, introduced in [Smith-90]. It is based on two models: *the software execution model*, which is based on execution graphs and represents the software architecture, and the *system execution model*, which is based on EQN models and represents the hardware platform. The analysis of the software model gives information concerning the resource requirements of the software system. The obtained results are combined together with information about the hardware devices. The resulting model represents the whole software/hardware system.

In this approach, the software architecture is specified using three different UML diagrams: Deployment diagrams, Sequence diagrams, and Use Case diagrams. From the Use Case and Sequence diagram, the proposed methodology derives the software execution model, and from the Deployment diagrams, it derives the system execution model. Both Use Case and Deployment diagrams are enriched with performance annotations concerning workload distribution and parameters of devices.

In [Balsamo-01], a comparison between eleven of the recently proposed approaches on the transformation of the UML models of software Architectures into Performance evaluation models is presented, with the aim of pointing out how the model transformation techniques make use of the UML diagrams. The comparison is based on different aspects of these

approaches. For example, it considers the types of UML diagrams and the type of derived performance models (such as queueing networks, stochastic Petri nets, stochastic process algebra, or simulation models). The comparison also considers whether or not the approaches introduce specific constraints on the Software Architecture specified by the UML diagrams. It also considers the generality of the proposed technique, which can be a systematic methodology, or an informal presentation or just a case study. Most importantly, it considers the additional information associated with the UML diagrams that are needed for a complete definition and analysis of the performance model, and how this information is specified (i.e. whether by using UML extensions or by using simple annotations on the UML diagrams).

This study came up with very interesting results. For instance, it concluded that Interaction diagrams, such as Sequence diagrams and Collaboration diagrams, are the most important diagrams used, since they model the software behaviour and present information concerning the flow of control of the software components. Use case diagrams are used to derive information to specify workloads, while Deployment and Component diagrams give information concerning the structure of hardware devices and software components.

The study also observed that there is a common feature of the transformation approaches that concerns the information to be added to the UML model to complete the definitions and parameterization of the performance model. It found out that the approaches generally observe that UML models do not provide all the necessary information for a complete definition of the performance model; hence, UML extensions or simple diagram annotation are proposed in almost all approaches.

Various approaches, as was concluded, follow the Software Performance Engineering (SPE) methodology, firstly presented in [Smith-99]. Finally, the study observed that most of the considered approaches are based on Queueing Networks (QN) performance models, including their extensions like Extended Queueing Networks (EQN) models and Layered Queueing Networks (LQN) models, since QN models have been traditionally used for performance evaluation and prediction, and have proved to be a powerful and versatile tool.

2.4.1 Performance profile

Performance profile is a new concept in the works of OMG [OMG-00]. The 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. [Woodside-01]. OMG performance profile draft appeared after most of the work in this thesis was done. However, it would have been so helpful to have this notion available, since UML models do not provide all the necessary information for a complete definition of the performance model, as was mentioned before.

2.5 Programmed Graph Rewriting System (PROGRES)

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. Graph rewriting systems are often used as visual and executable specifications of abstract data types or graph manipulating tools. PROGRES is a visual programming language that supports **PRO**gramming with **G**raph **R**ewriting **S**ystems. It has a graph-oriented data model and a graphical syntax for its most important language constructs. It offers additional means for defining derived data and for non-deterministic programming [Schürr-94, 97, 97b, 99].

2.5.1 Components of a PROGRES Graph

A PROGRES graph is a directed attributed graph that consists of labeled nodes and directed labeled edges, where attributes may be attached to nodes only. Nodes represent different types of objects, while edges represent the relationships between these objects.

2.5.1.1 PROGRES Syntax

PROGRES offers the following syntactic constructs for defining the components of a particular class of graphs and their legal combinations. These are

- **Node types**: which determine the static properties of their nodes instances. These nodes are called REALIZATION nodes.

- **Node Classes**: which define common node type properties in order to inherit them to node types as needed. Node classes play about the same role as abstract classes, whereas node types are the counterparts to derived objects. They are called SPECIFICATION nodes.

- **Intrinsic relationships**: also called **edge types**, which are explicitly manipulated, and possess restrictions concerning the types of their sources and targets.

- **Derived relationships**: defined by means of **path** or **restriction** expressions, which model complex relationships between nodes or node classes, that are often needed or used in a given graph. A *path* is represented by double arrows between two nodes, whereas a r*estriction* is represented by double arrows pointing to a node.

- **Intrinsic attributes**: which are defined for a particular set of node types and which are explicitly manipulated

- **Derived attributes**: which are defined by means of directed equations and which may have different definitions for different node types.

2.5.1.2 Negative Nodes and Relationships

A crossed-out node and a crossed-out edge (or path) represent the declaration of a negative node and edge respectively. The condition of a negative node or edge succeeds if matches for all positive node and edge patterns may be found such that a match for the negative node or edge pattern does not exist. In other words, a negative node or a negative edge represent the condition that such a node or edge cannot exist if the condition is to succeed.

2.5.2 Definitions of Graph Schema

Using the previously defined syntax, the language features provided by PROGRES enable us to define static properties of graphs in the form of a graph schema. The term graph schema is similar to the term database schema in the database design process. The graph schema

definition part of a PROGRES specification is usually kept separate from the graph rewriting rules definitions. Figure 8 illustrates the notation for defining PROGRES schema:

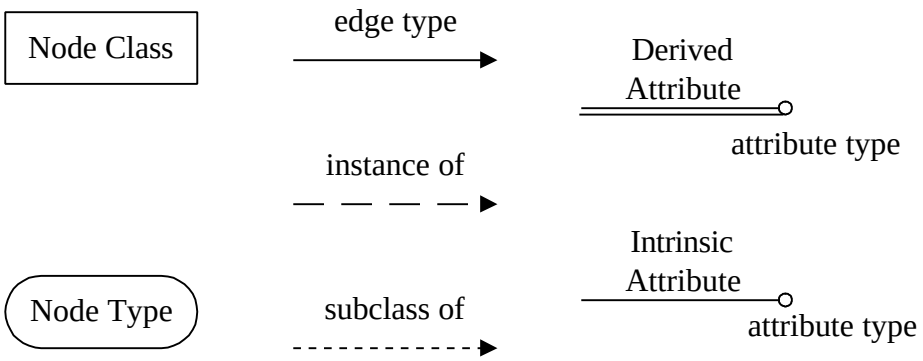


Figure 8 Notation of Graph Schema

- Normal boxes represent node classes, which are connected to their super classes by means of dotted edges representing “is-a” relationships.
- Boxes with round corners represent node types, which are connected to their uniquely defined classes by means of dashed edges representing “type is instance of class” relationships
- Solid edges between node classes represent edge type definitions
- Circles attached to node classes represent attributes with their names above or below the connection line segment and their type definition nearby the circle. A double line segment connects a derived attribute to its class while a normal line segment connects an intrinsic attribute to its class.

All nodes inherit the attributes of their ancestors, but different node types have different rules on how to compute the value of their derived attributes. PROGRES enables us to build hierarchies of node classes by exploiting multiple inheritance as the relation between node

classes using the is_a relationship to express the inheritance relationship. Multiple inheritance may be used to cut down the size of graph schema definitions considerably.

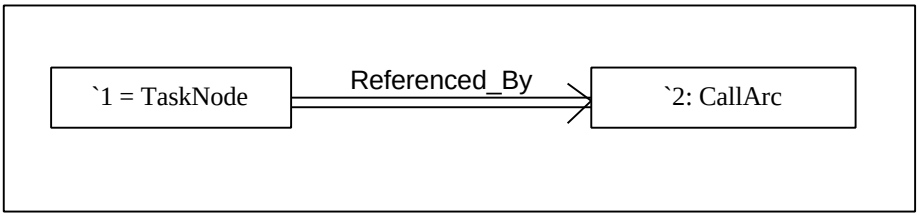
2.5.3 Definition Of Graph Transformations

Whereas the graph schema part of a PROGRES specification enables us to specify the static properties of a graph, the graph transformation part enables us to define the rewriting rules by which we can query and change the manipulated graph. This can be done using tests, queries, productions, transactions, and functions. PROGRES allows the importing of external functions and data types as well.

Tests and queries

Sub-graph tests and queries are the main constructs for inspecting already existing graphs using path declarations and restrictions. Sub-graph tests search for the existence or nonexistence of a certain sub-graph pattern in a host graph.

```
test IsTaskReference( TaskNode : TASK_NODE) =
```




```
end;
```

Figure 9 Example of a Test

A query is the complex form of tests, where you can have multiple tests in a conditional branch or a loop, for example. All sub-graph tests use restrictions and paths as complex application conditions.

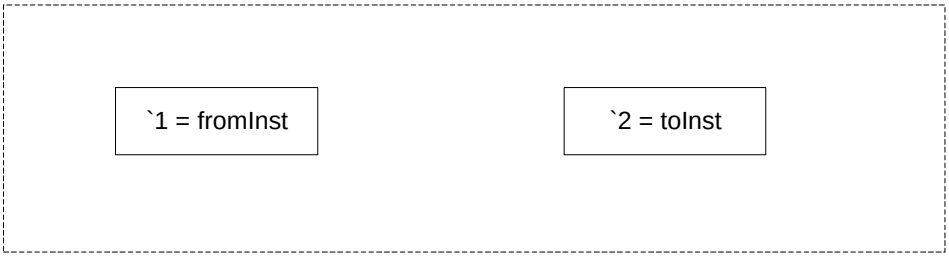
```
query IsThisEntryEmpty( LocalEntry : Entry) =
         begin
              IsEntryEmpty ( LocalEntry )
            & IsEntryInactive ( LocalEntry )
         end
      end;
```

Figure 10 Example of a Query

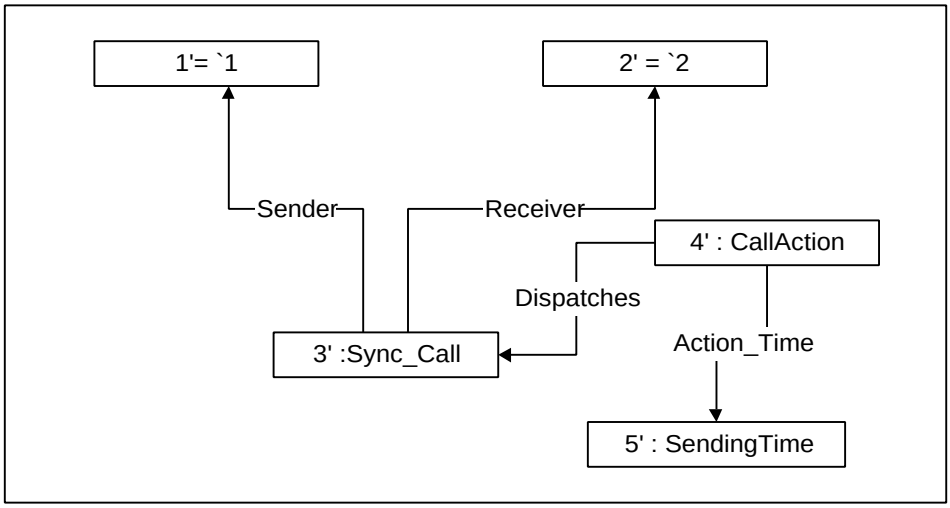
Productions and Transactions

Productions and transactions are used for creating and modifying schema consistent graphs. Productions have a left and a right hand side graph pattern as their main components. The left hand side (LHS) pattern of a production describes a sub-graph that must exist if the production was to be executed. It is similar to executing a test to search for the existence or non-existence of the LHS sub-graph. The right hand side (RHS) pattern of a production defines the node transformations that take place. It may include adding or deleting nodes or edges, as well as changing attribute values.

```
production CreateBasicSyncCall(fromInst, toInst : INSTANCE ;
msgName : string ; TimeVal : integer ; out ActionId : ACTION) =
```



::=




```
transfer 3’.Name := msgName;
         4'.Type := "SyncCallActionType";
         5'.Value := TimeVal;
         return ActionId := 4';
    end;
```

Figure 11 Example of a Production

The general rules for executing a Production are the following:

- The LHS matching succeeds if all positive nodes and edge patterns are found and the negative nodes and edges are not found.

- All positive nodes and edges in the LHS that have no counterparts in the RHS are deleted from the graph.

- All nodes and edges in the RHS that have no counterparts in the LHS are added to the graph.

- New attribute values are computed, as specified by the transfer section of the production.

Transactions use the same control structures to construct complex graph transformation processes from basic production applications in the same way as queries use them to construct complex graph analysis processes from basic sub-graph tests. Both Productions and Transactions can take a list of parameters and return a list of output values. Output parameters must have the keyword “out” before their declaration.

```
transaction GenerateLQNFile =
   use ret : integer
   do
        ret := openFile ( "test.lqn" )
      & WriteHeaderSection
      & WriteProcessorSection
      & WriteTaskSection
      & WriteEntrySection
      & ret := closeFile ( 1 )
   end
end;
```

Figure 12 Example of a Transaction

The Main Transaction

Each PROGRES program must have exactly one transaction named “MAIN”. This transaction serves as the entry point to the PROGRES program. Here is an example of a MAIN transaction:

```
transaction MAIN =
   begin
       CreateProblem
      & SolveProblem
   end
end;
```

Figure 13 Example of the Main Transaction

Non-deterministic Programming and Backtracking:

The PROGRES language was developed having certain design goals in mind. One design goal is to use a graphical syntax where appropriate but not to exclude textual syntax when it is more natural and concise. Also, keeping track of rewriting conflicts and backtracking out of dead end derivations was another design goal in mind. PROGRES specifications do not rely on the rule-oriented programming paradigm for all purposes but support also imperative programming of rule application strategies. These design goals were laid out because graph rewrite rules with complex application conditions and control structures with backtracking are rather useful for specifying software development tasks.

Backtracking in PROGRES tries to find another match within the host graph in case of failure. It works as follows. PROGRES tries to find a match for a certain configuration in a

certain test or query. If more than one match exists, one is chosen non-deterministically (which means that if the execution of a program repeated, it is not guaranteed that the same matches will be chosen again). PROGRES then tries to find a match for the next configuration in the next test or query. If it succeeds in finding a match, then it proceeds to finding a match for the next configuration and so on. If it fails to find any match in the existing graph, then backtracking starts, reentering the first test call and reactivating the pattern match process. As a result, the first test either determines another configuration or fails. In the first case, execution of the test or query is continued with checking the consistency of a new configuration; in the second case, the whole query fails and triggers backtracking or abortion of its calling test or query in turn.

Functions

A function in PROGRES does some work for computing a certain value. It may have may input parameters, but it must have only one output value.

```
function Increment : ( Number : integer) -> integer =
         (Number + 1)
      end;
```

Figure 14 Example of a Function

Imported types and functions

Imported or built-in types and functions may be defined in PROGRES for an application-oriented style of programming. These types and functions are defined in the “imported”

section of a PROGRES specification, and the libraries where their specifications exist are copied to PROGRES imported libraries directory.

```
from RealNumbers import

      types
         Real;

      functions
         R_IntToReal    : ( integer) -> Real,
         Real_StringToValue
                        : ( string) -> Real,
         Real_ValueToString
                        : ( Real) -> string,
         R_0            : ( integer) -> Real,
         BQPlusQuote    : ( Real, Real) -> Real,
         BQStarQuote    : ( Real, Real) -> Real,
         BQSlashQuote   : ( Real, Real) -> Real;

   end;
```

Figure 15 Example of the Import Section

Using the PROGRES language may not be suitable for solving all kinds of problems. Choosing a suitable programming language depends on the nature of the problem you are trying to solve. In this work, we are trying to transform software specification from UML to LQN, both of which having well defined graphical representation. The PROGRES Language was found suitable to use for solving this kind of problem.

3 UML TRANSFORMATION – PHASE 1

3.1 Conceptual Description

3.1.1 Sequence Diagrams

Sequence diagrams is a popular way of describing an interaction, which is a set of partially ordered messages between several objects in a collaboration to achieve a certain function or result. An Interaction specifies several communications between a sender role and a receiver role. Collections of Objects that conform to these Classifier roles communicate by dispatching Stimuli, which in turn conform to the Messages in the Interaction. Sequence diagrams emphasize the time ordering of messages. A sending time can be specified for each dispatched message.

Each Message was dispatched as a result of an Action. Several types of actions could be specified that may or may not dispatch a Message. For example, the Create Action results in the creation of a new Object Instance. The Destroy Action results in a message that destroys the receiver Object. On the other hand, a Local Action could be specified for an Object to perform some calculations locally or to make a decision. Also the Terminate Action results in an Object that terminates itself, without any Messages being dispatched.

UML differentiates between two types of messages, namely synchronous and asynchronous messages. In the synchronous message or call case, the sender object role blocks waiting for the receiver object role to finish processing before it returns back to its flow of control. In the

asynchronous message case, the sender object role drops a message in the receiver object role mailbox and resumes immediately its own flow of control. There is also a notation for a reply call if indicating a reply is necessary to show in a sequence diagram.

One interaction diagram is often not enough to represent a use case. Several sequence diagrams might be needed to represent all possible flows of events of a use case. Action sequences can be used to represent nested sequence diagrams as an action sequence can be further decomposed into several actions and action sequences, and may possibly be represented via another sequence diagram.

3.1.2 Activity Diagrams

Activity diagrams focus on the activities that take place among objects. Whereas interaction diagrams show flow of control from object to object, activity diagrams show flow of control from activity to activity. An interaction diagram looks at the objects that pass messages, while an activity diagram looks at the flow of actions executed by the objects. An activity is an ongoing non-atomic execution within a state machine. Activities ultimately result in some kind of an action. Within an activity diagram, you can also model the flow of an object as it moves from state to state at different points in the flow of control.

Activity diagrams commonly contain action states, activity states, transitions, and objects. Action states are atomic and can't be decomposed. They represent the execution on an action. Activity states, on the other hand, are not atomic and can be further decomposed. Each activity state might be composed of other action or activity states and can have its own activity diagram.

Transitions show the path from one action or activity state to the next action or activity state. Each transition is triggered upon the completion of its previous state and does not have a special trigger of its own. Beside the simple, sequential transitions, activity diagrams can have a branch, which specifies alternate paths taken based on some Boolean expression called a Guard. When branching is non-conditional, it represents concurrent flows of control. A fork represents the splitting of a single flow of control into two or more flows of control, while a join represents the synchronization of two or more concurrent flows of control.

If objects are involved in the flow of control associated with an activity diagram, they are represented by what is called “Object flows”. Activity diagrams can illustrate the object flows by connecting objects with dependency relationships to the activities acting upon them. It can also show the objects’ changing role, state and attribute values within a certain object flow.

An interesting way or organizing activity states on an activity diagram is to partition them into groups, called swimlanes. In an activity diagram, each swimlane represents a high-level responsibility for a group of activities, and can be implemented by one or more classes in general. There is a loose connection between swimlanes and concurrent flows of control. Independent and concurrent flows of control can, but not necessarily do, map to different swimlanes. For example, an Activity diagram may represent the workflow in an enterprise, where different swimlanes represent different departments. Even though a department may have internal concurrency, this is not shown in the Activity diagram.

In our case, however, we choose to build the activity diagrams at a granularity level where each swimlane corresponds to a single execution flow. In other words, a swimlane will

contain the activities carried out by one active object and any number of associated passive objects. In the second graph transformation phase, a swimlane will correspond to an LQN task.

3.1.3 The Aim Of Phase1

The first phase of the transformation converts from Sequence diagrams to Activity diagrams. The main purpose of this phase is to identify the active threads of control and the interactions between them. Several Sequence diagrams describing different interactions between the same set of objects, passive and active, would typically be converted into one Activity diagram with multiple levels of abstractions. This transformation is done using PROGRES.

3.2 PROGRES Graph Transformation

3.2.1 The Schema

In UML, all diagrams could be described in terms of their constituting elements and the relationship existing between these elements. This part of the schema (given in Figure 16) represents the UML metamodel description of a Sequence diagrams in PROGRES notation. In this notation, nodes represent the UML elements and edges represent the relationships among them. Figure 17 describes in turn Activity diagrams in exactly the same way. The shaded nodes in this figure are ones that were initially introduced in the first figure that had to be repeated.

3.2.1.1 UML Defined Elements For Sequence Diagrams

The elements constituting any Sequence diagram, as described in the UML metamodel specification [OMG-99], are described in this section. The relationships defined between them are represented via the edges connecting these elements as shown in Figure 16 below.

Some attributes, such as “Size” (for ARGUMENT), “Cycles” (for ACTION), and “Probability” (for GUARD and STATE_VERTEX) do not come from the UML metamodel. They were added to the schema to represent performance annotations.

Model Element:

A model element is an element that is an abstraction drawn from the system being modeled. In the metamodel, a Model Element is a named entity in a Model. It is the base for all modeling meta-classes in the UML. All other modeling meta-classes are either direct or indirect subclasses of Model Element.

Classifier:

A classifier is an element that describes behavioral and structural features. It comes in several forms, including class, data type, interface, and component. In the metamodel, a Classifier declares a collection of Attributes, Methods, and Operations.

Classifier Role:

A classifier role is a specific role played by a participant in a collaboration. It specifies a restricted view of a classifier, defined by what is required in the collaboration.

Instance:

The instance construct defines an entity to which a set of operations can be applied and which has a state that stores the effects of the operations. In the metamodel, Instance is connected to

at least one Classifier which declares its structure and behavior. Instance is an abstract metaclass.

Object:

An object is an instance that originates from a class. In the metamodel, an object is a subclass of Instance and it originates from at least one Class. In this work, we represent two types of objects: **Active Objects**, which are instances of Active Classes, and **Passive Objects**, which are instances of Passive Classes.

Message:

In the metamodel, a Message defines one specific kind of communication between instances in an Interaction such as raising a Signal, invoking an Operation, creating or destroying an Instance.

Stimulus

In the metamodel, a stimulus conforms to a Message. It is a communication, such as a Signal sent to an Instance, or an invocation of an Operation. It has a sender, a receiver, and may have a set of actual arguments, all being Instances.

Action:

An action is a specification of an executable statement that forms an abstraction of a computational procedure that results in a change in the state of the model. It can be realized by sending a message to an object or modifying a link or a value of an attribute. In the

metamodel, an Action may be part of an Action Sequence and may contain a specification of a target as well as a specification of the actual arguments.

Action Sequence:

An action sequence is a collection of actions. In the metamodel, an Action Sequence is an Action, which is an aggregation of other Actions.

Call Action:

A call action is an action resulting in an invocation of an operation on an instance. In the metamodel, the Call Action is an Action. The designated Instance or set of Instances is specified via the target expression, and the actual arguments are designated via the argument association inherited from Action.

Send Action

A send action is an action that results in the sending of a signal. Although calling an operation is different than sending a signal, in this work both Send Actions Call actions are considered to be the same type of action.

Create Action:

A Create Action is an action resulting in the creation of an instance of some classifier.

Destroy Action:

A Destroy Action is an action results in the destruction of an object specified by the target association of the Action.

Terminate Action:

A Terminate Action results in self-destruction of an object. The target of a Terminate Action is implicitly the Instance executing the action.

Signal:

A signal is a specification of an asynchronous stimulus communicated between instances. In this work, both signals and asynchronous stimuli are considered to be the same thing.

Argument:

An argument is an expression describing how to determine the actual values passed in a dispatched request. It is aggregated within an action.

Time

In the metamodel, a Time defines a value representing an absolute or relative moment in time and space. A Sending Time is associated with each action or sent message in this work for determining the order of actions with respect to time.

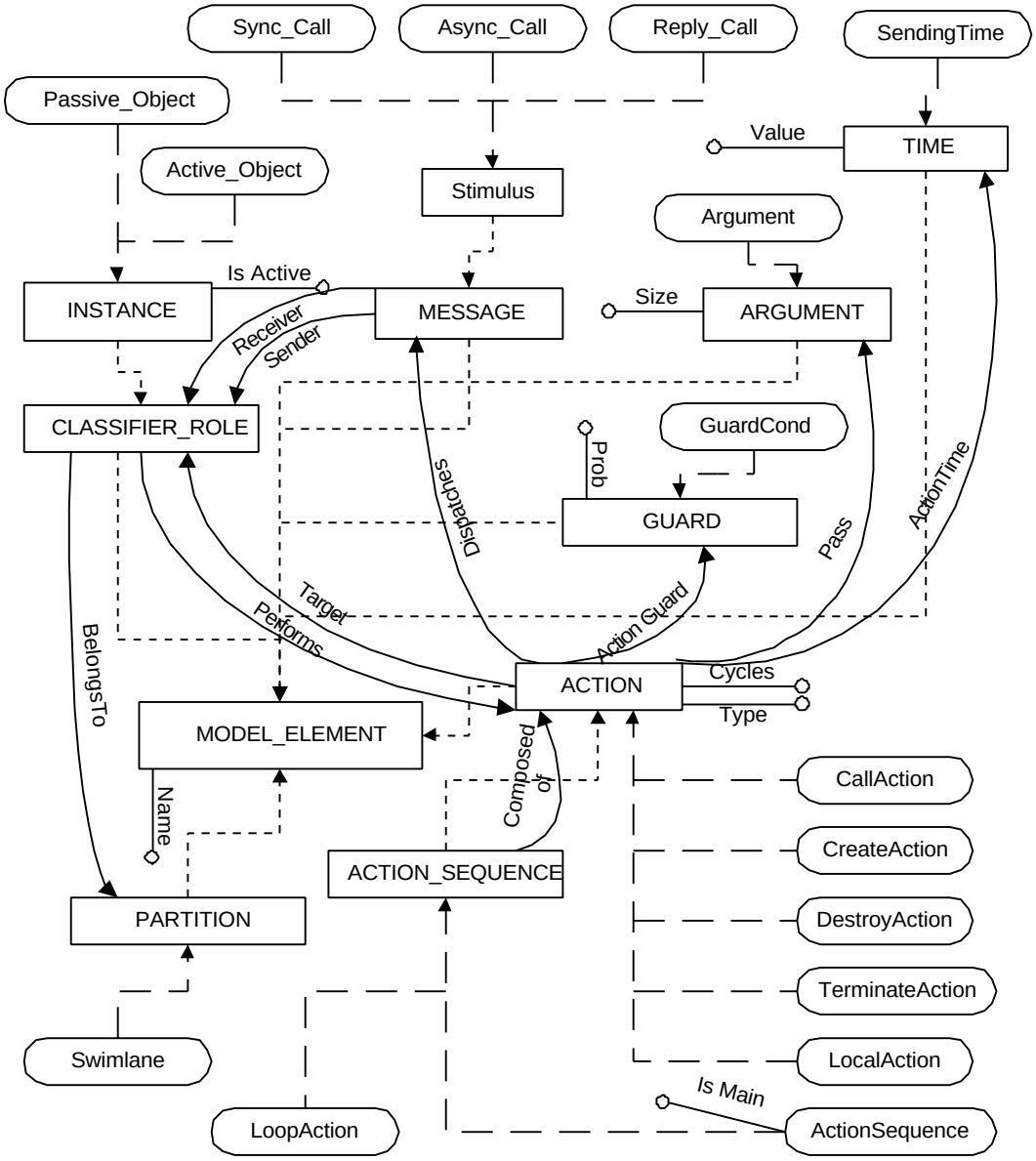


Figure 16: PROGRES Schema for Sequence Diagrams

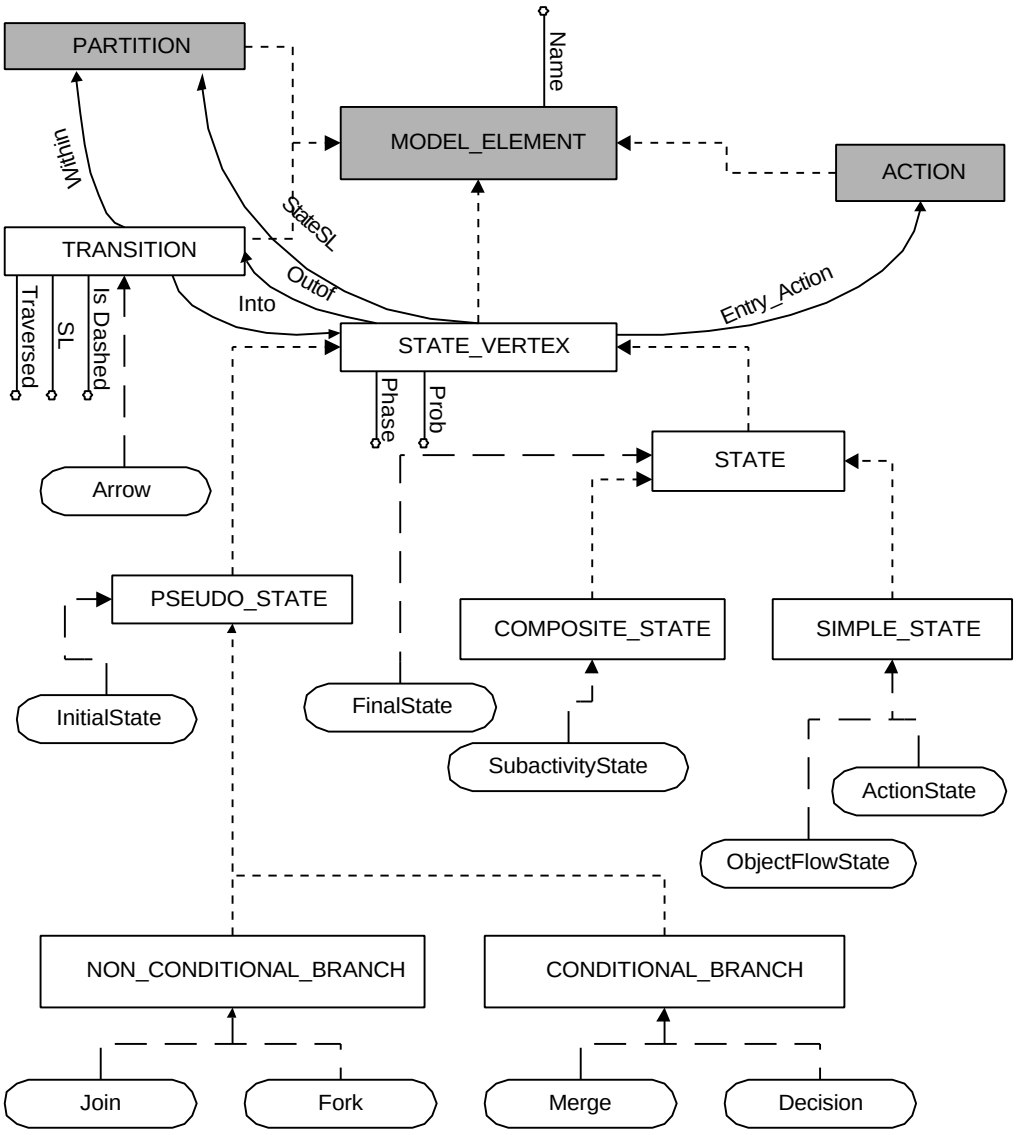


Figure 17: PROGRES Schema for Activity Diagrams

3.2.1.2 UML Defined Elements For Activity Diagrams

The following are the definitions of elements in the UML metamodel that appear in an Activity diagram as described in [OMG-99].

Transition

A transition is a directed relationship between a source state vertex and a target state vertex. Transition is a child of Model Element.

State Vertex

A State Vertex is an abstraction of a node in a state chart graph. In general, it can be the source or destination of any number of transitions. State Vertex is a child of Model Element.

State

A state is an abstract meta-class that models a static situation, such as an object waiting for some external event to occur, or a dynamic situation, such as the process of performing some activity. The model element under consideration enters the state when the activity starts and leaves it as soon as the activity is completed. State is a child of State Vertex.

Pseudo State

A pseudo state is an abstraction that includes different types of transient vertices that are used to connect multiple transitions into more complex state transitions paths. Pseudo State is a child of State Vertex. Here are some of the pseudo states used in this work:

•An **initial** Pseudo state represents a default vertex that is the source for a single transition to the *default* state of a composite state. There can be at most one initial vertex in a composite state.

•The **join** pseudo state serves to merge several transitions coming from different source state vertices. The transitions entering a join vertex cannot have guards.

•The **fork** pseudo state serves to split an incoming transition into two or more transitions. The segments outgoing from a fork vertex must not have guards.

•The **decision** pseudo state is term defined in this work to represent both static and dynamic conditional branch. It is a junction that can be used to split an incoming transition into multiple outgoing transition segments with different guard conditions.

•The **merge** pseudo state is a junction that can be used to converge multiple incoming transitions into a single outgoing transition representing a shared transition path

Composite State:

A composite state is a state that contains other state vertices (states, pseudo states, etc.). A state vertex can be a part of at most one composite state. Composite State is a child of State.

Sub-activity State:

A sub-activity state is a submachine state that executes a nested activity graph. The semantics of a sub-activity state are equivalent to the model obtained by statically substituting the contents of the nested graph as a composite state replacing the sub-activity state.

Action State

An action state represents the execution of an atomic action, typically the invocation of an operation. An action state is a simple state with an entry action. The state therefore corresponds to the execution of the entry action itself and the outgoing transition is activated as soon as the action has completed its execution.

Object Flow State

An object flow state defines an object flow between actions in an activity graph. Operating on an object by an action in an action state may be modeled by an object flow state that is triggered by the completion of the action state. Generally each action places the object in a different state that is modeled as a distinct object flow state.

Final State

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

Guard

A guard is a boolean expression that is attached to a transition as a control over its firing. If the guard is true at its evaluation time, the transition is enabled; otherwise, it is disabled. Guard is a child of Model Element.

Partition

A partition is a mechanism for dividing the states of an activity graph into groups. Partitions often correspond to organizational units in a business model.

Swimlane

A swimlane maps into a Partition of the States in the Activity Graph. A state symbol in a swimlane causes the corresponding State to belong to the corresponding Partition.

3.2.1.3 Loops and Complex Branching

The notation provided in the UML for specifying iteration and complex branching is very limited. An iteration indicates that the message, as well as any nested messages, will be repeated in accordance with the given expression. A condition represents a message whose execution depends on the evaluation of a Boolean guarding condition. To model an iteration in the UML, the sequence number of a message is prefixed with an iteration expression such as *[i = 1..n], or just * without specifying any details. To model a condition, the sequence number of a message is prefixed with a guarding condition expression, such as (x >0). The alternate path of a branch will have the same sequence number, but each path must be

uniquely distinguishable by a non-overlapping condition. This notation is not sufficient to clearly represent loops and branching in a Sequence diagram.

In [ITUT-99], there is a formal, more elaborate, definition for specifying loops and complex branching in Message Sequence Charts (MSC). MSC is a structured, formal graphical language that was the basis from which Sequence diagrams were derived. In [ITUT-99], it is stated: “Simple scenarios (described by Basic Message Sequence Charts) can be combined to form more complete specifications by means of High-level Message Sequence Charts”. It has introduced the concept of Inline expressions, a high-level structural concept, that help structure notions of alternatives, parallel composition and loops. In this work, we have used these definitions to represent loops and complex branching in a Sequence diagram. The node “Loop Action” was added to the Sequence diagram Schema as explained below. This is not from the present UML metamodel, but there is a hope that new versions will include something similar. The reason for hope is that the present UML 1.3 version already states in words that “a connected set of messages may be endorsed and marked as an iteration”. This will have to be supported at the metamodel level in future UML versions.

Loop Action:

Which is of type ACTION_SEQUENCE. It indicates that the loop or a complex branch is a composite action, which can be further expanded in a separate Sequence diagram.

Both Complex Branching and Parallel Compositions can be represented via an Action Sequence node. In this work, we assume that if more than one Action or Action Sequence has the same Sending Time value, this indicates a case of either branching or parallel

composition. For simplicity, if the Actions or Action Sequences with the same sending time value have Guard Conditions attached to them, they are considered alternatives of conditional branching; otherwise, they are considered to be happening in parallel.

According to [Booch-99], in activity diagrams, the effect of iteration can be achieved by using one action state that sets the value of an iterator, another action state that increments the iterator, and a branch that evaluates if the iteration is finished. This technique is used in this work to transform loops in Sequence diagrams into loops in Activity diagrams.

3.2.2 Transactions And Productions

As described before, all elements of Sequence diagrams and Activity diagrams are represented in PROGRES in the form of Nodes and Edges. There are two kinds of nodes: Node Types and Node Classes. Node Types are the ones that determine the static properties of their node instances, while Node Classes are definitions of common Node Type properties. The relationship between node types and node classes is similar to the relationship between derived classes and their abstract base classes. There are two types of edges as well: edges that define intrinsic relationships, which are explicitly defined between sources and targets, and edges that define derived relationships, which define a path expression between a source and a target that are not directly connected.

3.2.2.1 Production Rules For Creating Sequence Diagrams

As mentioned before, the PROGRES program developed in the thesis is a standalone program, not connected to a UML tool. The “input graph” which will be transformed into an “output graph” has to be entered first. We have defined a set of APIs under the form of

PROGRES transactions that can be used to generate the input graph for this phase of the transformation. The list of APIs is given below.

- CreateNewInstance(InstanceName, TaskName : string ; IsActive : boolean ; out NewInst : INSTANCE)

```
(*Creates a new Instance. IsActive determines if it is an
Active_Object or Passive_Object. Returns a handle to the new
instance*)
```

- CreateMainActionSequence(ActionSequenceName : string ; out MainActionSequence : ACTION_SEQUENCE)

```
(*Creates the main action sequence, the one enclosing all other
actions and action sequences. Returns a handle to the main action
sequence*)
```

- CreateDynamicNewInstance(fromInst : INSTANCE ; InstanceName : string ; IsActive : boolean ; TimeVal : integer; LocalAS : ACTION_SEQUENCE ; ExpCycles : integer ; out NewInst : INSTANCE)

```
(*Creates a new instance dynamically. Returns a handle to the
new instance*)
```

- DestroyDynamicInstance(fromInst, toInst : INSTANCE ; TimeVal : integer ; LocalAS : COMPOSITE_ACTION ; ExpCycles : integer)

```
(*Destroys an instance dynamically*)
```

- TerminateInstance(theInst : INSTANCE ; TimeVal : integer ; LocalAS : COMPOSITE_ACTION ; ExpCycles : integer)

```
(*Terminates the calling instance theInst*)
```

- CreateLocalAction(InstId : INSTANCE ; actionName, OptGuard : string ; TimeVal : integer ; LocalAS : COMPOSITE_ACTION; ExpCycles : integer)

```
(*Creates a local action. No message is dispatched*)
```

- CreateActionSequence(InstId : INSTANCE ; actionName, OptGuard : string ; TimeVal : integer ; LocalAS : COMPOSITE_ACTION ; ExpCycles : integer ; out ResultASAction : COMPOSITE_ACTION)

```
(*Creates an action sequence. Returns a handle to the new action
sequence*)
```

- CreateLoopAction(InstId : INSTANCE ; LoopId : string ; LoopCount, TimeVal : integer ; LocalAS : COMPOSITE_ACTION; out ResultLoop : COMPOSITE_ACTION)

```
(*Creates an loop action. Returns a handle to the new loop
action*)
```

- ```
  CreateSyncCall( FromInstId, ToInstId : INSTANCE ; MsgName,
  OptArg : string ; ArgSize : integer; OptGuard : string ;
  TimeVal : integer ; LocalAS : COMPOSITE_ACTION ; ExpCycles :
  integer)
  (*Createa a synchronous call between two instances*)
  ```

- ```
  CreateAsyncCall( FromInstId, ToInstId : INSTANCE ; MsgName,
  OptArg : string ; ArgSize : integer; OptGuard : string ;
  TimeVal : integer ; LocalAS : COMPOSITE_ACTION ; ExpCycles :
  integer)
  (*Creates an asynchronous call between two instances*)
  ```

- ```
  CreateReplyCall( FromInstId, ToInstId : INSTANCE ; MsgName,
  OptArg : string ; ArgSize : integer; OptGuard : string ;
  TimeVal : integer ; LocalAS : COMPOSITE_ACTION ; ExpCycles :
  integer)
  (*To indicate a reply to a synchronous call*)
  ```

- ```
  Sleep( FromInstId : INSTANCE ; TimeVal : integer ; LocalAS :
  COMPOSITE_ACTION ; ExpCycles : integer)
  (*To indicate a delay without doing any work*)
  ```

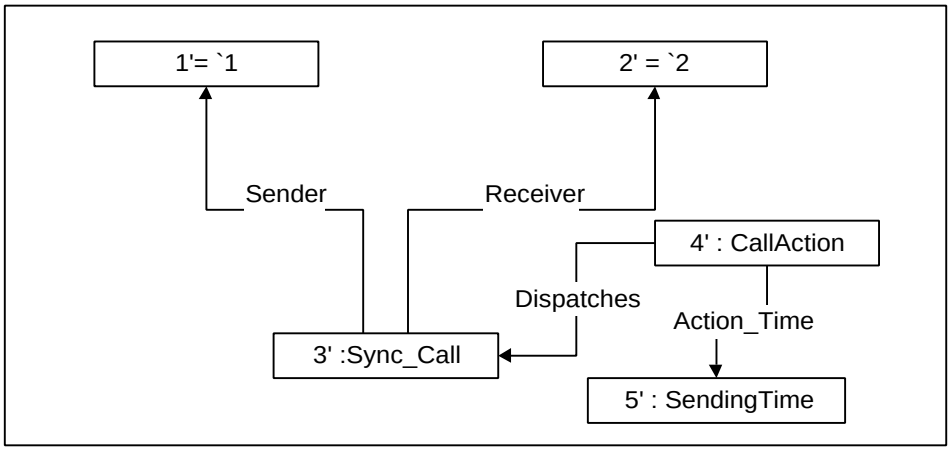
Here is an example of how PROGRES generate a Synchronous Call in a Sequence diagrams using its production rules (illustrated in Figure 18). This production rule is called when there is a call action between two active objects, where the message exchanged is of type “Synchronous Call”. The left had side (LHS) of the production has two nodes: *fromInstance* and *toInstance.* This means that the pre-conditions for performing this production rule is to find the match for the given two instances in the current PROGRES generated set of nodes. The right hand side (RHS) has five nodes, two of which match those in the LHS. This means that the effect of the production rules is the generation of the remaining three nodes and all the added edges while keeping the LHS nodes. The added nodes represent the call action, the time of the call action, and the message dispatched as a result of the call action, namely a synchronous call. The added edges establish the relationships between these nodes. For

example, the *fromInstance* becomes the *Sender* of the synchronous call, while the *toInstance* becomes the *Receiver* of the synchronous call.

The *Sync_Call* node is assigned *msgName* as its name, the *Sending Time* node is assigned the input parameter *TimeVal* as its value, and the *CallAction* is given the type *SyncCallActionType* as illustrated by the transfer section. The return value of this production is the *CallAction* node itself.

```
production CreateBasicSyncCall(fromInst, toInst : INSTANCE ;
msgName : string ; TimeVal : integer ; out ActionId : ACTION) =
```

::=




```
transfer 3'.Name := msgName;
         4'.Type := "SyncCallActionType";
         5'.Value := TimeVal;
         return ActionId := 4';
    end;
```

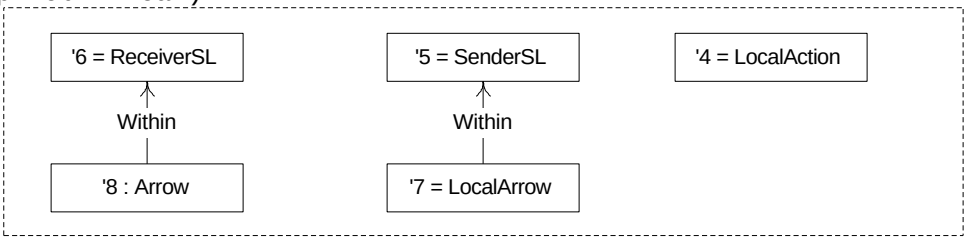
Figure 18: CreateBasicSyncCall Production Rule.

3.2.2.2 Production Rules For Transforming SDs To ADs

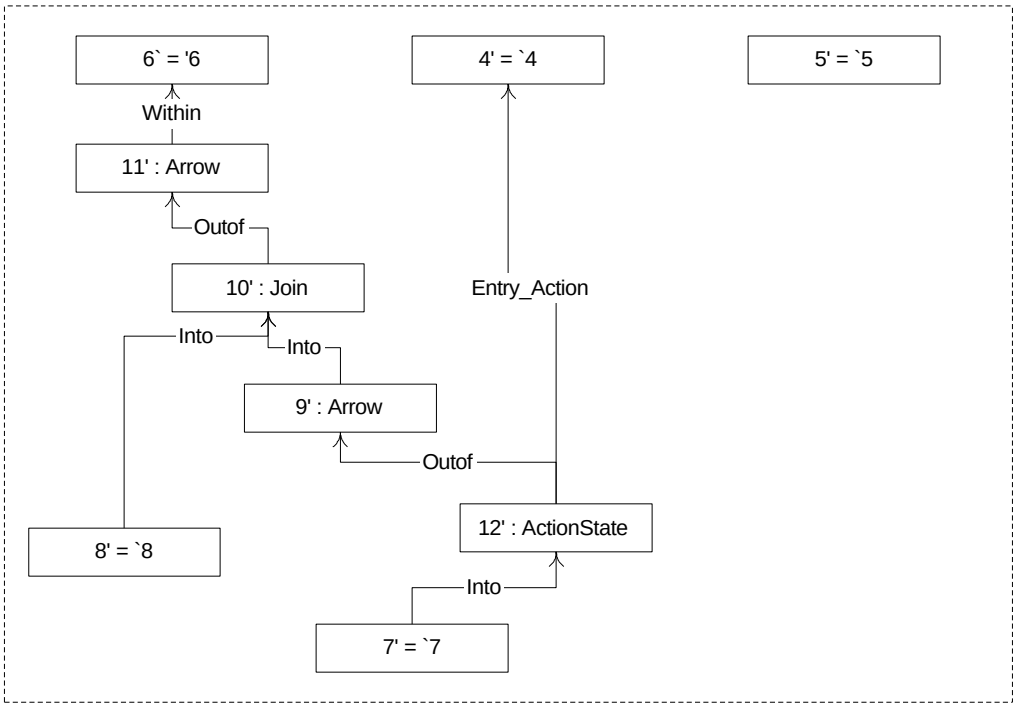
After creating the input graph that represents a set of Sequence diagrams, the PROGRES will transform the actual conversion of the graph into the output graph. The transformation from Sequence diagrams into Activity diagrams is done using production rules. Figure 19 illustrates one of the production rules used in this transformation, namely "Transform-SyncCallNoArg".

In this production rule, PROGRES looks for the pattern indicated in the LHS of the production. It looks for the sender thread and the receiver thread, along with the local action that caused the dispatch of the synchronous message. An arrow "*within*" the sender thread and the receiver thread indicate that both threads are running in their own flow of control and are not joined with another thread, blocked or terminated for any reason. Unfortunately, if PROGRES finds several matching pre-condition patterns in its running instance, it chooses one randomly. It is up to the programmer to take advantage of this fact, for instance in depth-first searches, or to make sure that there is only one pattern in a running instance for PROGRES to chose from.

```
production TransformSyncCallNoArgument( LocalAction : ACTION
; SenderSL, ReceiverSL : Swimlane ; LocalArrow : Arrow ;
OpProb : Real) =
```




```
::=
```



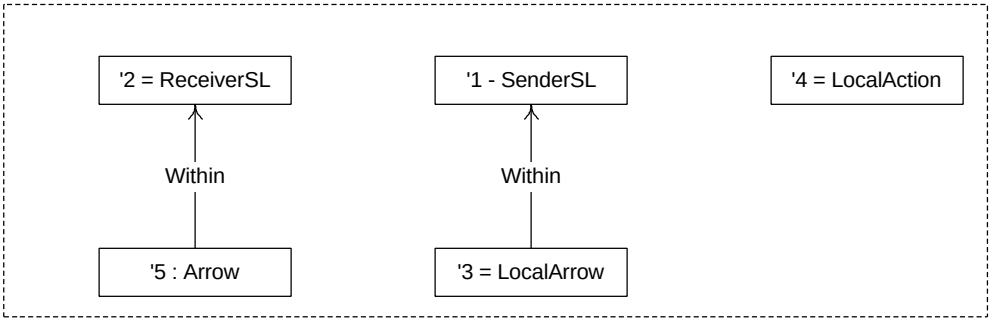

```
transfer 12’.Name := ‘6.Name & "." & ‘4.Name;
                10’.Name := "JOIN " & ‘5.Name & " and " & ‘6.Name;
                9’.Name := ‘4.Name;
                9’.SL := ‘5.Name;
                11’.SL := ‘6.Name;
                12’.Prob := OpProb;
                10’.Prob := OpProb;
    end;
```

Figure 19: TransformSyncCallNoArgument Production Rule

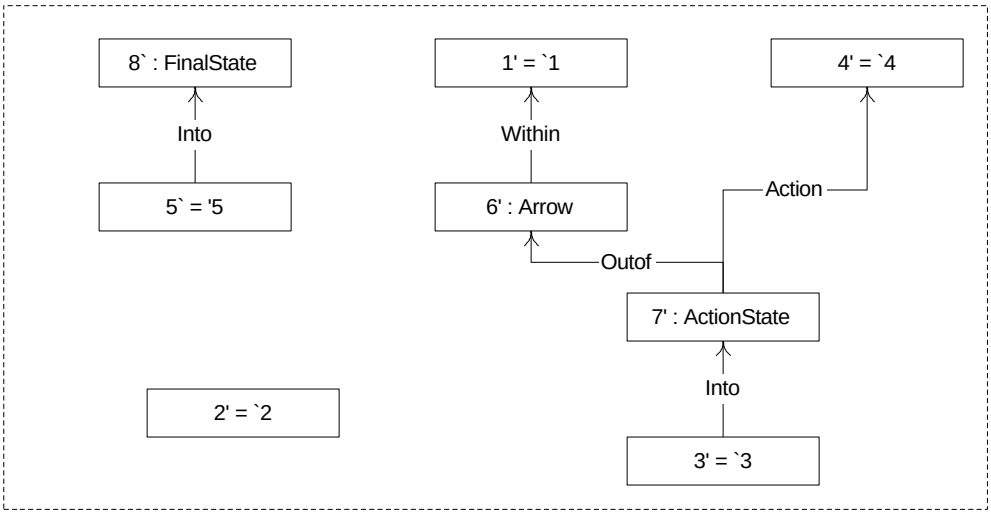
When PROGRES finds the pattern matching the pre-condition, it generated the pattern in its production’s RHS. In this example, an *Action State* node is created, with the action causing the message set as its entry action. Also a *join state* is created to join the sender thread and the receiver thread, and the sender thread is blocked waiting for a reply. Having no Arrow “within” the sender swimlane indicates this blocking.

Another example is illustrated in Figure 20, which shows the “TransformDestroyAction” production rule. The sender thread in this case attempts to destroy the receiver thread by sending a “Destroy Message”. When executing such a production rule and after finding the pre-condition pattern, an Action State is created in the sender swimlane, with the Destroy Action set as its entry action. A final state is created in the receiver swimlane, and the arrow “within” the receiver swimlane is directed to it. No arrows can be going “out of” the final state, which means that the thread has terminated.

```
production TransformDestroyAction( LocalAction : ACTION ; SenderSL,
ReceiverSL : Swimlane ; LocalArrow : Arrow ; OpProb : Real)
    =
```




```
::=
```




```
  transfer 7’.Name := ‘4.Name & " " & ‘2.Name;
           8’.Name := "END " & ‘2.Name;
           6’.SL := ‘1.Name;
           7’.Prob := OpProb;
  end;
```

Figure 20: TransformDestryAction Production Rule.

3.2.2.3 Transformation Algorithm

After creating a Sequence diagram (or a set of nested Sequence diagrams), the following algorithm is followed to transform all indicated messages and actions into an Activity diagram (or a set of nested Activity diagrams). The indicated Sending Time for each action is used to order these actions. Here is a simplified pseudo code of this algorithm:

- Get Main Action Sequence
- Transform All Instances Into Initial States
- Get All Times in Action Times Set
- When not empty (Action Times Set)
    - Loop
    - Get Min Time (Action Times Set)
    - If one action in an Action Time
        - Then Transform One Actions
    - If mode than one action in an Action Time
        - Then if these actions have Guard Conditions
            - Then transform as Alternative Actions
            - Else transform As Parallel Actions
    - End Loop
- Repeat algorithm for all nested Action Sequences

Note that the time of actions indicate the partial ordering of actions. The Action Times Sets holds the time values (i.e. order) of actions that occur in an action sequence.

The result of executing this algorithm is the generation of an equivalent set of Activity diagrams to the input set of Sequence diagrams that will be used in the next phase of transformation into LQN Models.

3.2.3 A Simple Example

This section explains the first phase of transformation via a simple example. In this example, we have two active objects, a client and a server. The client communicates with the server by

sending a synchronous message. When the server receives this message, it does some work, then sends a reply back to the client. The client is blocked until the server sends the reply. Three actions are identified in this example. The first action, a *Call Action*, is performed by the client and has a time value = 1. This call action dispatches a synchronous call, whose sender is the client and whose receiver is the server. The second action, a *Local Action*, is performed by the server, and has a time value = 2. This local action does not dispatch any message. If some expected number of cycles is associated with this amount of work, it will be assigned to the action and used later in calculating the service time. The third action, a Call Action, is performed by the server, and has a time value = 3. It dispatches a reply message whose sender is the server and whose receiver is the client. The following figure illustrates the sequence diagram that describes this example, along with its graph representation.

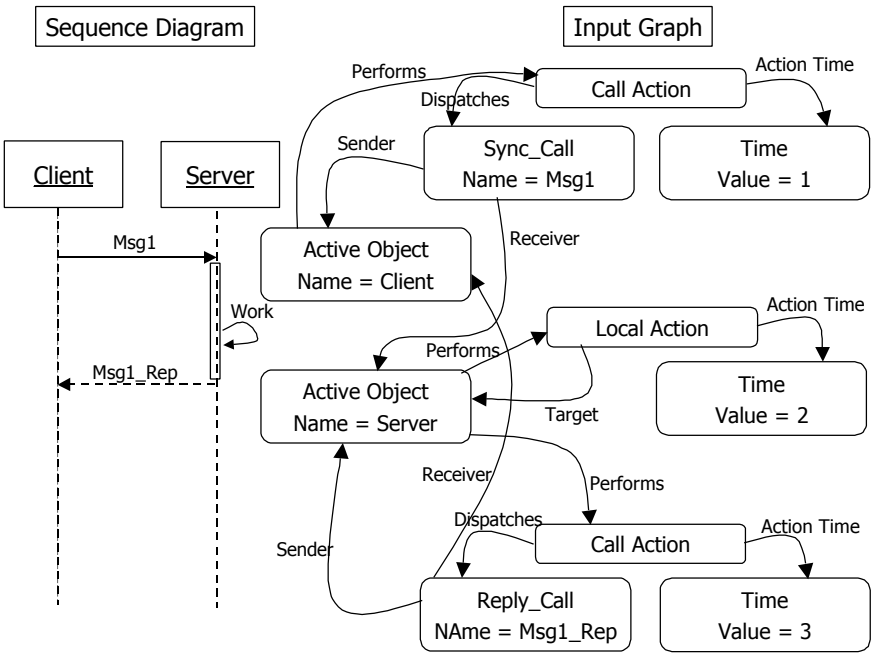


Figure 21: Simple example, Phase1 input graph

After applying the transformation rules, the input graph is transformed to an output graph that represents an activity diagram. The two active objects are transformed into two initial states, each belonging to a different swimlane, namely the client thread and the server thread. An arrow is going out of each initial state. The first action, the call action, is transferred into a join state followed by an arrow. The two arrows from the two swimlanes join by going into the join state. The second action, the local action, is transformed into an action state, whose entry action is the local action. An arrow is going out of this action state. The third action, the reply action, is transformed into a fork state, where two arrows are going out of it, each belonging to a different swimlane. The following figure illustrates the output graph of phase 1, along with the Activity graph that it represents.

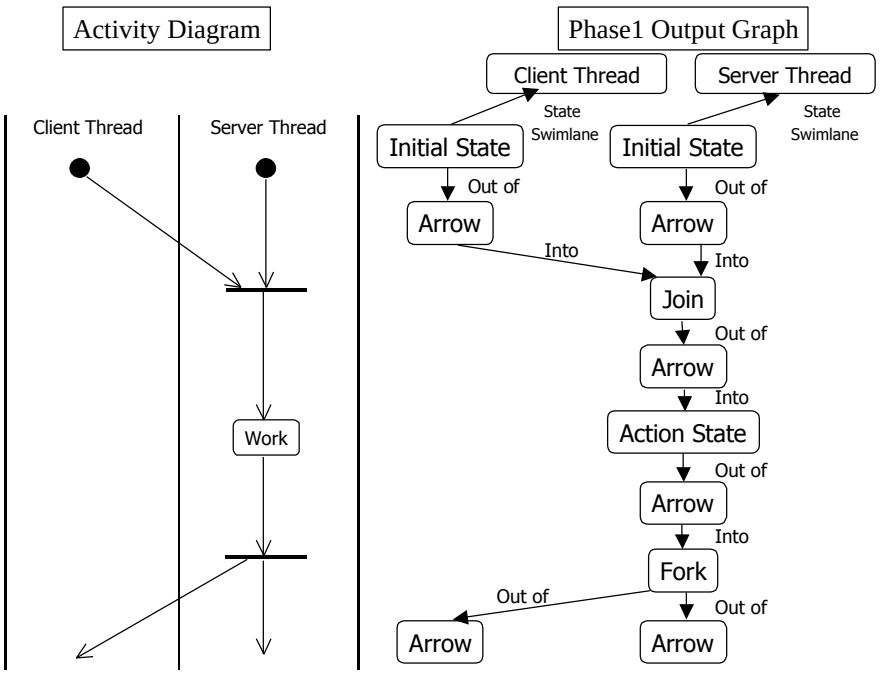


Figure 22: Simple Example, Phase1 output graph

More examples of the first phase of transformation can be found in Chapter 5 “Case Studies” (see for example Figure 39) as will be explained later.

4 UML TRANSFORMATION – PHASE 2

4.1 Conceptual Description

4.1.1 LQN models

The layered Queuing Networks (LQN) is a model of a network of tasks running on processors and communicating via a send-receive-reply pattern [Woodside-89][Woodside-95][Franks-95]. If the sender of a message is blocked waiting for a reply, we call this pattern of communication a rendezvous, an RPC, or a synchronous message. If the sender of a message does not wait for a reply, we call this asynchronous messaging.

In any LQN model, the two basic building blocks are tasks and requests. A task is an entity that models a software process execution demand and executes some work if its processor is available. Each task may have different classes of workloads on the processor, which can be represented by having several entries. Each entry provides a different service pattern and a different workload; it has its own service time and visit ratio to other server tasks. All entries of one task share a common task queue, which uses a first-come first-served queuing discipline.

The execution of an entry after receiving a message may be broken into more than one phase. The first phase ends when the reply is sent back and the caller is unblocked. The second phase ends when the entry is done processing the request. There might be a following third phase in which the entry forwards the request to another entry in another task.

An entry can be further decomposed into activities if more details are required to describe its execution scenario. This is typically required when entries have fork and join interactions. (See section 2.2.1.3 for more details about activities). Execution of a sequence of activities may branch into parallel concurrent threads of control, or choose randomly between different paths. In this work, Intra-task fork-join, where the fork and join take place within the same task, is used in one of the architectures of the case study, namely the P-ORB (see section 5.4 for more details).

A communication request from one task (called a client) to another (called a server) can be synchronous, asynchronous, or forwarding. The client is blocked when sending a synchronous request until the server sends back the reply. An asynchronous request is a request where the client does not block waiting for a reply. A forwarding request is similar to a synchronous request from the client's point of view. However, the server does not send back the reply to the client. Instead, it forwards the request to another server, and then, it is free to do other work. After the second server finishes the request, it sends back the reply to the original client instead of sending it to the first server. The client is blocked until it receives the reply. There can be more than two servers in the forwarding chain.

A server may be a single server, a multi server or an infinite server. A single server is modeled as a single task, which handles only one request at a time. A multi server is modeled as a number of tasks sharing one processor, and a common queue for incoming requests. A replicated server, however, is similar to a multi-server, except that each task has its own processor and request queue. An infinite server is modeled as an infinite number of tasks on an infinite number of processors that can handle an infinite number of requests.

4.1.2 The Logical Mapping

Flows of control and Messages exchanged among them in the UML notation map to Tasks and communication Requests in LQN. In UML, each independent flow of control is modeled as an active object, which can independently initiate and receive control activities (messages or signals). An Active object is an object that owns a process or a thread. A process is a heavyweight flow that can execute concurrently with other processes, whereas a thread is a lightweight flow that can execute concurrently with other threads within the same process [Booch-99].

Objects interact by passing messages from one to the other. If a message is passed from one passive object to another, such an interaction is nothing more than simple invocation of an operation, assuming that there is only one flow of control passing through these objects at a time. If a message is passed from one active object to another, we have inter-process communication, which might be either a synchronous call of an operation or an asynchronous send of a signal or call of an operation. If a message is passed from an active object to a passive object, it can be viewed as a thread using a resource. In this case, mechanisms for safe resource sharing among several active objects must be considered. If a message is passed from a passive object to an active object, it has the same semantics as an active object passing a message to an active object, since every flow of control is eventually rooted to some active object [Booch-99].

4.1.3 Aim of Phase 2

In phase 1, we obtain Activity diagrams that identify flows of control and inter-process communications. The aim of phase 2 is to transform software representation from Activity

diagrams into an LQN model. In phase 2, flows of control are transformed into tasks and inter-process communications are transformed into requests among these tasks. In this work, it is assumed that a task is a component in UML that contains only one active object representing its flow of control, and may contain zero or more passive objects as well, all belonging to one and the same swimlane. In a more general case, other criteria can be used to decide how many objects, active and passive, belong to a swimlane. Information of components and their contained objects are depicted in UML in Component diagrams and Deployment diagrams.

However, there are other performance parameters that must be specified as the input of an LQN models. For example, the number of processors in the system, how they are linked physically, and how tasks are allocated on them must be specified. The speed of processors and physical communication links and the multiplicity of tasks and processors are also input parameters. For each service provided, an average service time must be provided in an LQN model as well.

Moreover, it is very useful to know in advance the relationships between tasks in collaboration. For example, when interpreting an asynchronous message between two tasks, it would be very helpful to know who is the server and who is the client in that collaboration. With minimal effort, we can know if this message is a request or a reply. If the two tasks were in a different collaboration, a “pipeline” collaboration for instance, the interpretation of the message might be totally different.

Some of this information is extracted from UML specifications, but from diagrams other than Sequence and Activity diagrams. For example, Deployment diagrams show processing nodes and how they are liked together and Component diagrams show objects to components allocation. Moreover, one has to add quantitative performance annotations to these diagrams, as for example processors and links speed as well as the multiplicity of both processors and tasks. From Collaboration diagrams, the relationship between objects could be extracted and used in interpreting communication requests among tasks. This type of work was done in previous masters thesis [Qang-99]. Extracting information from Deployment, Component and Collaboration diagrams is not the main focus of this work; therefore, very simple methods were added for extracting the needed information without deep analysis done on the metamodel for these diagrams. The following sections in this chapter explain how Phase 2 transformation is done.

4.2 PROGRES Graph Transformation

The schema section describes how LQN models elements are represented in PROGRES schema. The following sections explain the rules and the algorithms used to transform PROGRES representation of Activity diagrams into an LQN model input file.

4.2.1 The Schema

This part of the schema includes the elements that constitute an LQN model. Some of the nodes of the schema represent input parameters needed for any LQN model. Other nodes represent the relationship that might exist between objects in a collaboration. This information could be extracted from notes attached to UML diagrams, such as Component,

Deployment, and Collaboration diagrams. Figure 23 illustrates these elements and the relationship between them.

Processor

A processor is the physical node on which one or more tasks may run.

Task

Tasks represent software components, which may execute concurrently.

Entry

Entries differentiate service demands at the tasks. A task may have several entries, one for each service it provides.

Activity

Activities are components that represent the lowest level of detail that are used to model concurrent processing or alternate processing within a task.

Phase

Phases denote different intervals of services within entries.

Arc

Arcs represent requests for service made from entry to entry through send-receive-reply message interactions.

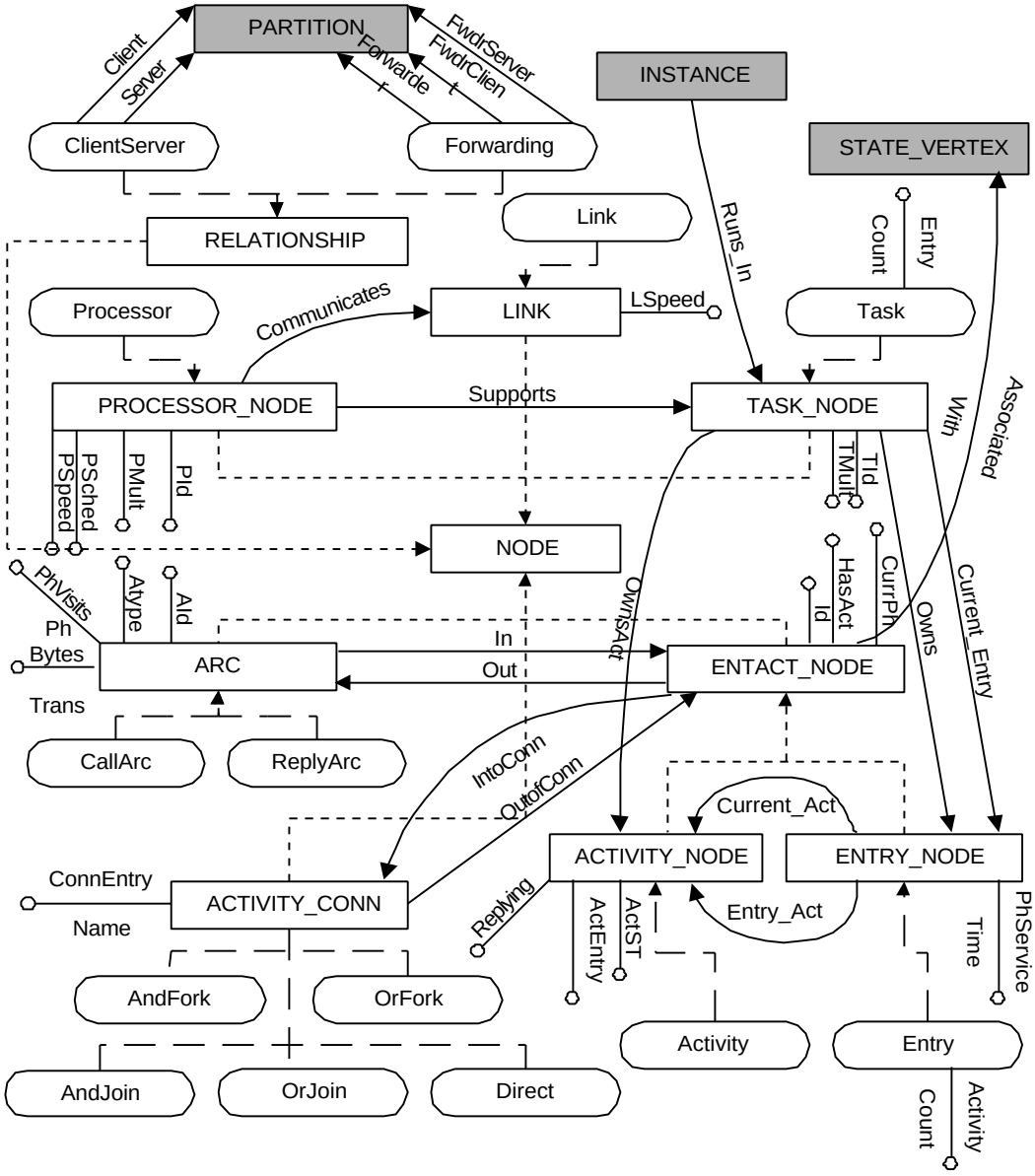


Figure 23: “PROGRES Schema for LQN elements”

Multiplicity:

A Multiplicity defines a non-empty set of non-negative integers. It indicates the number of copies that exist of a task (component) or a processor (node).

Service Time

The mean service time expected for each service provided by a server. In this work, it is given in terms of the number of expected processing cycles. It can then be multiplied by the processor's speed on which the server task is allocated (how many cycles per msec) to obtain the exact time.

Bytes Transferred

The size of the message transferred in bytes. This is used to calculate the communication delay, given a link speed.

Processor and Link Speed

The speed of the physical processor given by the number of cycles per msec, and the link speed given by the number of bytes transferred per second.

Reply Arc

This arc is not required to be explicitly shown in an LQN model. However, it was added to account for the communication delay when sending a long reply message.

Client-Server

Setting this relationship between two active objects indicates that the messages exchanged in between them can be interpreted in the context of a client-server relationship.

Forwarding

Setting this relationship between three active objects indicates that the messages exchanged in between them can be interpreted in the context of a client-broker-server relationship.

4.2.2 Transactions and Productions

In section 3.2.2.1 the APIs for generating the input graph were introduced. In section 4.2.2.1, the transactions used to set up the physical configuration are described. The productions used in the transformation from Activity diagrams to LQN models are described in section 4.2.2.2. Section 4.2.2.3 describes how an LQN input file is generated as an end result of this transformation.

4.2.2.1 APIs for creating configurations

Using the following APIs, we can describe the physical configuration of a software system. These APIs can be also used to indicate task-to-processor allocation (information extracted from deployment diagrams), as well as object-to-object relationship in any collaboration (information extracted from activity diagrams). Only two relationships are used in this work: Client-Server and Forwarding relationships.

- `transaction NewProcessorNode( ProcessorId : string ; Multiplicity : integer ; SchedulingFlag, ProcessorSpeed : string)`

```
(*Create a new processor node. Indicate processor multiplicity,
scheduling flag, and speed*)
```

- `transaction NewTaskComponent( TaskId, ProcessorId : string ; Multiplicity : integer)`

```
(*Create a task and allocated on this processor. Indicate Task
multiplicity*)
```

- `transaction LinkProcessorNodes( Proc1Id, Proc2Id, LinkSpeed : string)`

```
(*Create a physical link between the two processors. Indicate the link
speed*)
```

- production AddClientServerRelationship(ClientName, ServerName : string)

```
(*Set the relationship between two objects to be a client-Server
relationship*)
```

- production AddForwardingRelationship(ClientName, ForwarderName, ServerName : string)

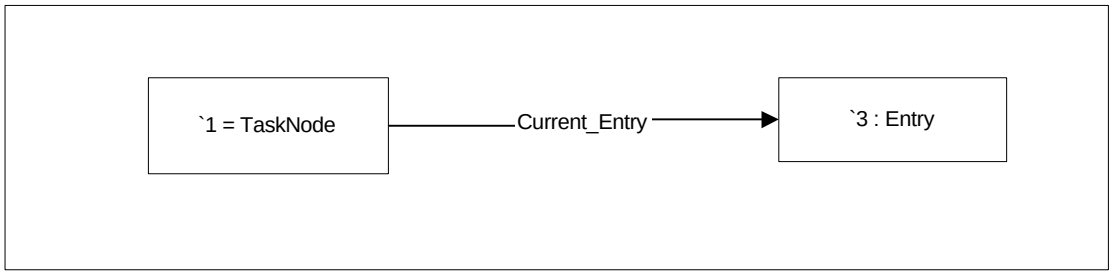
```
(*Set the relationship between three objects to be a client-broker-
server relationship*)
```

4.2.2.2 Production Rules

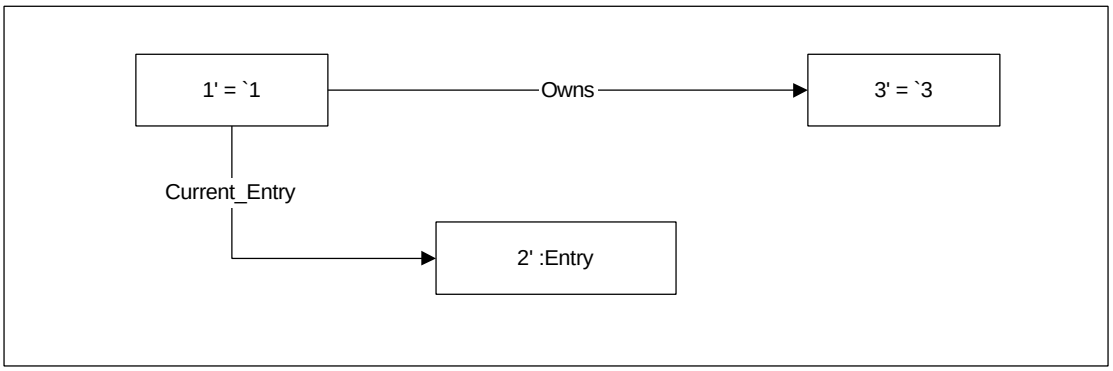
There are many production rules used in this phase of transformation to transform Activity diagrams into LQN models. We have selected only some of them for the purpose of illustration.

The first rule, “AddNewEntryToTask”, is called whenever a new entry is needs to be added to a task. This is detected when a server receives a new communication request that it didn’t receive before. Each task has one entry designated as the current entry. When a new entry is added, it becomes the task’s current entry.

```
production AddNewEntryToTask( TaskNode : Task ; EntryId : string) =
```




```
: : =
```



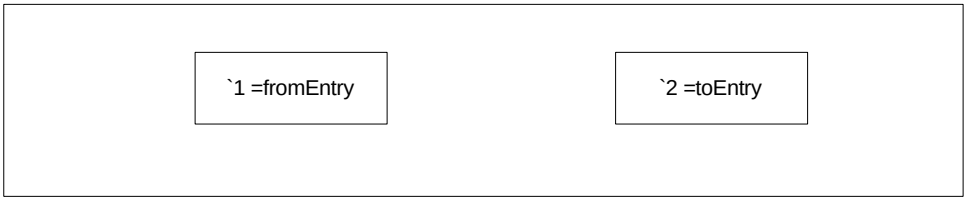

```
transfer 2'.Id := EntryId;
                  1'.EntryCount := (1 + `1.EntryCount);
end;
```

Figure 24:"AddNewEntryToTask" production rule

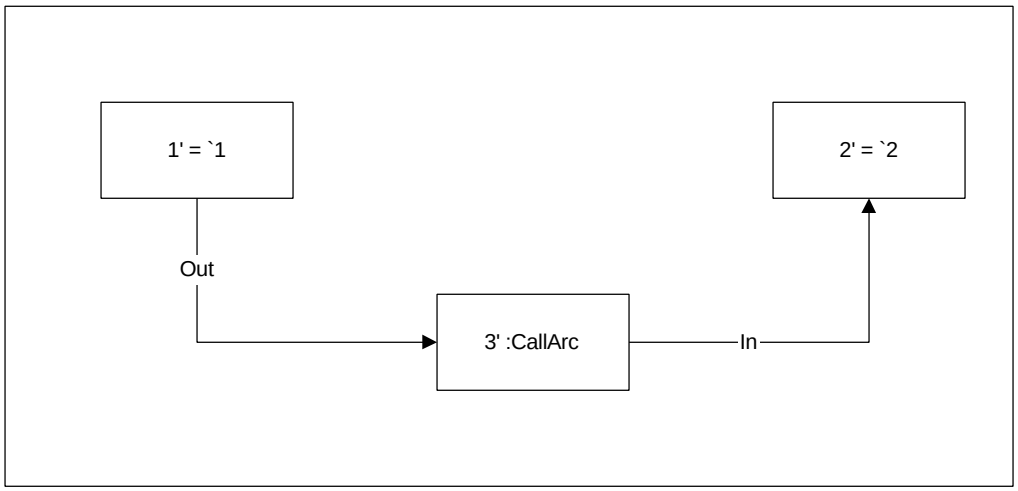
The new entry gets the passed entry ID and the task's entry count is incremented by one.

The "CreateArc" production rule adds an arc between two entries (or activities) whenever a request is detected. The node "CallArc" is created and passed back as a return value.

```
production CreateArc( Id, Type : string ; fromEntry, toEntry :
ENTACT_NODE ; ph1Visits, ph2Visits, ph3Visits : Real
                              ; out LocalArc : CallArc)
      =
```




```
::=
```




```
      transfer 3'.ArcId := Id;
               3'.ArcType := Type;
               3'.Phase1Visits := ph1Visits;
               3'.Phase2Visits := ph2Visits;
               3'.Phase3Visits := ph3Visits;
      return LocalArc := 3';
   end;
```

Figure 25: "CreateArc" production rule

A unique ID is given to each arc. The “Type” indicates whether it is a synchronous, asynchronous or a forwarding arc. The arc is created with a visit ratio calculated for each phase.

The following section presents the global picture of how this transformation is done.

4.2.2.3 Algorithm for generating LQN models

Generating an LQN performance model is done in three main stages. In the first stage, the processors, the links and the tasks in the system are identified and created. This information is directly given to the PROGRES program using the API “ConfigurePlatform” as will be illustrated in Chapter 5”Case Studies” later (see as an example Figure 36). In the second stage, as will be explained in this section, the transformation algorithm is applied to the

generated Activity diagram that resulted as output from phase 1 of the transformation. During this stage, entries and activities within a task are identified and generated, as well as arcs between these entries and activities. The final stage of creating the LQN performance model is writing the LQN output file and figuring out the service time of each entry and activity, as well as the arcs and their visit ratios. This stage is explained in the next section.

The transformation algorithm traverses the resulting activity graph that consists of one or more action sequences. It starts with the initial states in each action sequence. It calls "TraverseState" for every initial state. The transformation algorithm runs in two iterations. In the first iteration, it determines the number of entries or activities each task has. In the second iteration, it determines the arcs that go from one entry (or activity) to another, representing communication requests. The reason for this design is the non-deterministic nature of PROGRES. We cannot guarantee that the thread of control of the initiator of a request is going to be traversed before the receiver. If we are traversing the receiver before the initiator, and we detect that a request has been received, we cannot determine from which entry of the initiator this request has been sent. Here is the algorithm for traversing an Activity Graph.

```
• Traverse Activity Graph
• Get All Action Sequences
• Loop For all Action Sequences
    o  Get All Initial States
    o  Loop for all Initial States
        ▪  TraverseState ( InitialState, false )
    o  end
    o  Reinitialize Transitions
    o  Loop for all Initial States
        ▪  TraverseState ( InitialState, true )
    o  end
• end
```

Any activity graph consists of a set of states followed by a set of transitions followed by a set of states until final states are reached. That's why "TraverseState" ends up by calling "TraverseTransition" for all its outgoing transitions, which in turn calls "TraverseState" for the transition's following state. This recursion ends when the final state is reached. Here is the pseudo code for "TraverseTransition":

```
TraverseTransition( LocalTrans : TRANSITION)
    choose
     when (LocalTrans.Traversed = true)
       then skip
       else
           LocalTrans.Traversed := true
           GetStateOfTransition ( LocalTrans,out LocalState)
           TraverseState (LocalState)
    end
```

Figure 26: "Pseudo-code for TraverseTransition"

When traversing a state, one of four cases is detected. In case 1, the final state is reached. Upon reaching this case, traversing stops and continues where it has left before. If case 2 is reached, we have no transitions going out of the state, while it is not a final state. This means that the thread of control has blocked. In case 3, one or more transitions are outgoing from the current state. If there is only one transition going out, then the normal thread of control is followed. If there is more than one transition going out of the state, then we have a forking state (whether conditional or non conditional state) and outgoing transitions are traversed one by one. Here is the pseudo code for "TraverseState":

```
TraverseState( LocalState : STATE_VERTEX)
    GetTransitionsOfState ( out TransSet )
    Choose
       // Case 1: We have reached the final state
       when (LocalState is instance_of FinalState)
       then skip
       else
       // case 2: Thread is blocked
       when empty ( TransSet )
        then
            TranslateBlockingState
        else
        // case 3: thread is not blocked
        when not empty ( TransSet )
        then
          choose
             when (OnlyOneTransitionInTransSet)
             then
               // case 3.1: One thread of control after state
               TranslateNormalState
               TraverseTransition ( LocalTrans )
             else
                // case 3.2: Thread has forked to more than one thread
               of control
               TranslateForkingState
               for_all trans = elem ( TransSet )
                  TraverseTransition ( trans )
               end
             end
         end
end;
```

Figure 27:"Pseudo-code for TraverseState"

In all cases, a test for a key state is done. A key state is a Fork, a Join, a Decision or a Merge state. When a key state is found, a check is made to see if we are in one of the four situations: (a) Is the traversed thread sending a message? (b) Is it sending a reply? (c) Is it receiving a message? or (d) is it receiving a reply? A check to the established relationships between the active objects is done to determine the sending receiving logic here. Finally, the appropriate translation of adding a new entry or a new arc is done accordingly.

Here are the rules followed when generating the LQN model:

- Each task starts with only one entry. A new entry is added to the task if a new type of a communication request is received. If a request is received more than once with the same message ID, it is considered to be one request, with a visit ration = 2.

- 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. Phase 3 is left for future enhancement in case other relationships are added, such as the “pipeline” relationship.

- Activities are detected if a conditional or non-conditional branching state is encountered. In the case of conditional branching, an “OrFork” is used to connect alternate activities. A probability for each branch is calculated based on the given guard condition. An “OrJoin” is used to end the conditional branching, representing a “merge” state in Activity diagrams.

- In the case of non-conditional branching, an “AndFork” is used whenever a “Fork” state is used for concurrency. Similarly, and “AndJoin” is used to end the concurrency, representing a “Join” state in Activity diagrams. Note that in Activity diagrams, a “Fork” state might as well be used in the case of a server sending a reply. In such a case, no concurrency is in place.

- There are two types of arcs: Call arcs and Reply arcs. A Call arc is generated when a communication request is detected between an entry (or activity) and another entry

(or activity). A Reply arc is generated when a reply is detected. The visit ratio for each arc is calculated per phase.

The following graph illustrates the algorithm's two iterations. In the first iteration, a new entry is detected when a new request is received, while a new phase is detected right after a reply is sent. In the second iteration, communication arcs are detected when their corresponding patterns are matched while traversing the graph.

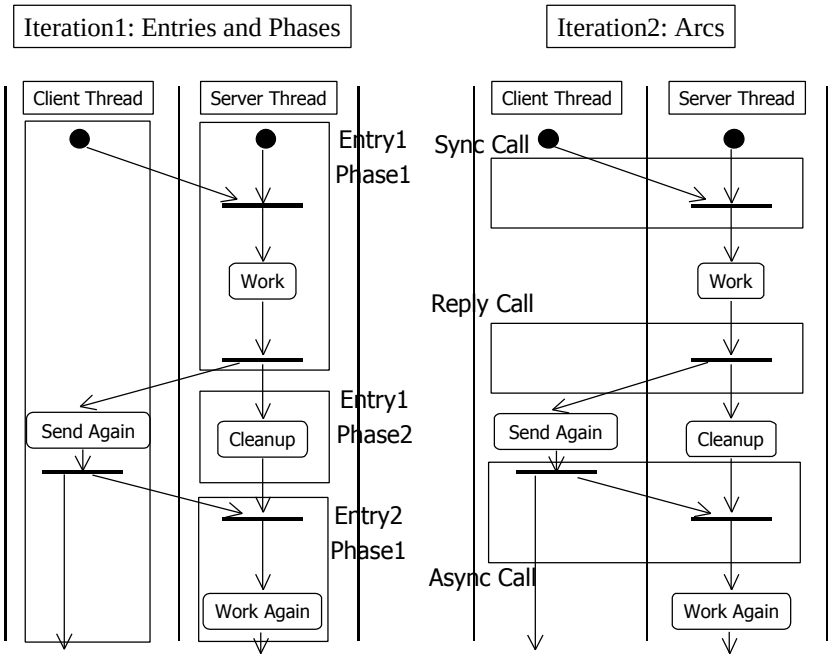


Figure 28: Extracting Entry, Phase and Arc information

Calculating The Service Time:

Service Times for entries and activities are calculated per phase. Each message or action entered in the Sequence diagram has an expected number of cycles on a processor. These

numbers of cycles are assigned to action states in the activity diagram. Action states in turn are assigned to entries or activities. The sum of the expected number of cycles in an entry (or activity) for all its assigned action states is multiplied by the probability of its occurrence and then divided by the processor speed of its task to obtain the entry's (or activity's) expected service time. The time spent in transferring messages is also added to the service time. For all arcs going out of the entry (or activity), the length of the message carried along on that arc is multiplied by the visit ration of that arc and then divided by the speed of the link that carries that message.

The following equation summarizes the way service times are calculated per entry (or activity) showing the two components: "pure" CPU execution and CPU overhead due to sending messages:

$$ServiceTimePerEntry = \sum ActionState.Cycles * ActionState.\Pr obability / \Pr ocessorSpeed + \sum OutArc.MessageLength * OutArc.VisitRatio / LinkSpeed$$

Out arcs include messages sent to other servers as well as replies sent back to clients. If the two tasks are co-allocated on a single processor, the overhead due to sending messages is not included as part of the CPU overhead.

Only the assignment of action states to entries and activities is done in this stage of transformation. Calculating the expected service time of entries and activities per phase is actually done in the next stage of transformation, as will be explained in the next section.

Correctness of the Transformation:

Many test cases were developed for testing each possible transformation and compare the output with the expected output in order to verify the correctness of the transformations. Test cases covered creating active instances, passive instances, synchronous calls (with and without arguments), asynchronous calls (with and without arguments), reply calls (with and without arguments), local actions, create actions, destroy actions, terminate actions, and loops. All previous actions were tried alone, in a conditional branch, and in a non-conditional branch, and were performed once on an active instance and another time on a passive instance.

4.2.2.4 Writing the LQN output file

After traversing all action sequences in the system, the output LQN model file is generated. This is done using the following transaction:

```
transaction GenerateLQNFile =
   use ret : integer
   do
        ret := openFile ( "test.lqn" )
      & WriteHeaderSection
      & WriteProcessorSection
      & WriteTaskSection
      & WriteEntrySection
      & ret := closeFile ( 1 )
   end
end;
```

Figure 29: "GenerateLQNFile transaction"

The first transaction opens the output file. The standard header section is then written to the output file. A query is done to get all the processors in the system, then a processor record is

filled with each processor's information. Finally, the whole record is written to the output file, creating the processor section.

A similar procedure is followed when writing the "Task" section. A query is done to get all tasks in the system, then a task record is filled with each task's information. If a task does not receive any incoming requests, it is marked as a reference task. The whole tasks record is then written once to the output file, creating the task section. Here is the code of the "writeTaskSection" transaction:

```
transaction WriteTaskSection =
use TaskSet : Task [0:n]; E_List, T_Ref, T_Proc : string;
        ret : integer
  do
    FindAllTasks ( out TaskSet )
  & for_all TElem := elem ( TaskSet )
    do
      GetEntryList ( TElem, out E_List )
    & GetTaskReferenceFlag ( TElem, out T_Ref )
    & GetTaskProcessor ( TElem, out T_Proc )
    & ret := updateTaskRecord ( TElem.TId, T_Ref, E_List, T_Proc,
TElem.TMultiplicity )
    end
  & ret := writeTaskRecord ( 1 )
 end
end;
```

Figure 30: "writeTaskSection transaction"

In "writeEntrySection", more work is done to write both the entry section and the activity section of the LQN file. After finding all entries and activities in the system, the service time per phase is calculated for each entry and activity and the entry record is updated. As explained in the previous section, action states are assigned to entries or activities. Their expected number of cycles on a processor, along with that processor speed, is used to

calculate the service time for an entry or activity. The set of arcs in the system is also found, and written to an arc record, with its visit ratio per phase. If a message is carried along with a call or reply arc, the message transformation time is also calculated using the message size and the given link speed on which this message was transferred. It is then added to the service time of the sender task entry. The final step closes the output file.

4.2.3 A Simple Example (cont.)

Continuing with the simple example, introduced in section 3.2.3, the second phase of transformation is described here. The output graph is phase 1 is used as the input graph of phase 2 of the transformation, along with other graphs that represent collaboration and deployment diagrams. It is given that there is a client-server relationship between the client active object and the server active object. This fact is represented with the node CLIENT_SERVER that links between the Client_Thread and the Server_Thread swimlanes in the PROGRES input graph of phase2.

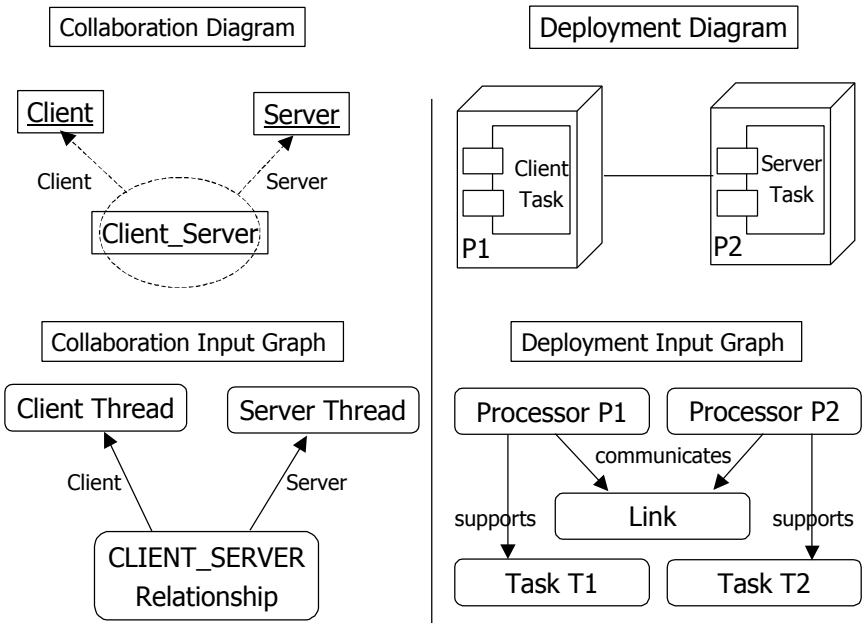


Figure 31: Simple Example, Collaboration and Deployment information

Also it is given from the deployment diagram that the client and the server are parts of different components, each supported by different processor nodes. This fact is represented in the input graph of phase two via two processor nodes, communicating through a link, that support two different tasks. Figure 31 illustrates the collaboration and the deployment diagrams, with their corresponding graphs.

The resulting graph from Phase1 transformation, together with the graphs representing deployment and collaboration information, constitutes the input graph of phase 2. After applying the transformation rules, and extracting the entries, phases and arcs information, the following graph is the result, which represents the corresponding LQN model.

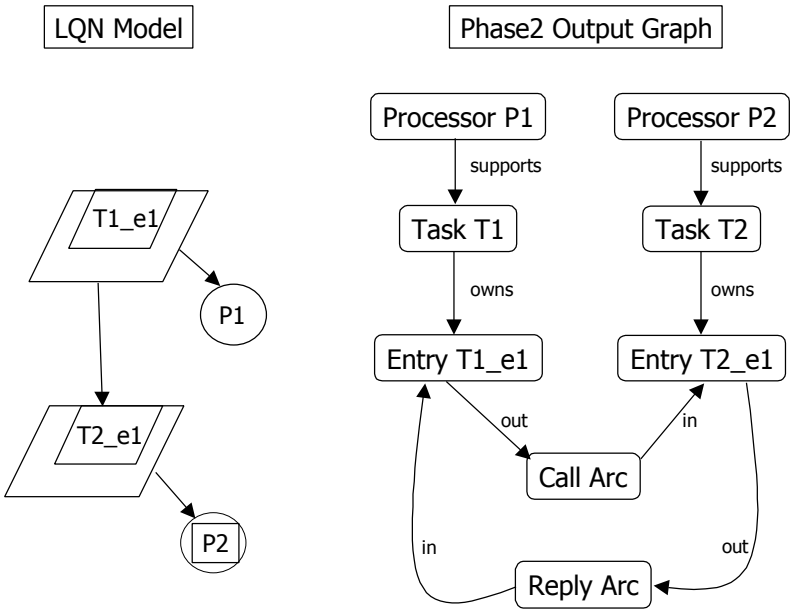


Figure 32: Simple Example, Phase2 output graph

The two tasks represent the two active threads of control. Each task has a default entry. The join state is transformed into a call arc between the two entries, while the fork state is transformed into a reply arc. Any work done in the Action state is transformed into some service time assigned to the entry of the task representing the server thread (namely T2_e1). The resulting graph represents an LQN model for the simple example shown on Figure 32. Note that reply arcs do not show on the LQN model.

More examples of the second phase of transformation can be found in Chapter 5 “Case Studies” (see for example Figure 49) as will be explained later.

5 CASE STUDIES

5.1 CORBA-Based Client Server Study

In a study that investigates the relative performance of different CORBA-based client-server architectures, based on a commercially available CORBA compliant ORB software called ***ORBLine***, [Abd-97][Abd98a, b], three architectures were designed and implemented, namely the Handle-driven ORB (H-ORB), the Forwarding ORB (F-ORB) and the Process Planner (P-ORB). The relative performance of these architectures was investigated under different workload conditions. The purpose of this chapter is to generate automatically the LQN models from UML specifications of the system and see if it gives acceptable results. A more detailed performance analysis of these systems by using "hand-built" LQN models was done in [Petriu-00a] (the thesis author is a co-writer of [Petriu-00a]) The models that were developed in [Petriu-00a] are equivalent to the models developed in this case study.

5.1.1 Why This Study Was Chosen

The distributed nature of the architectures studied in [Abd-97] and [Abd-98a,b] very well suits the nature of a problem solved by a Layered Queuing Network model. The three architectures studied are interestingly different in nature, which implies that the automation process will face three different challenges to solve. Furthermore, the measurements taken on the real system serve to validate the correctness and the accuracy of the models resulting from the automation process.

5.1.2 The Three Architectures

In [Abd-97], the case study was inspired from the model of a banking system. The bank offers two types of accounts, a Checking account and a Savings account. The bank maintains two independent classes of servers, namely Class A for checking accounts and Class B for saving accounts. There are two identical class A servers, A1 and A2, and two identical class B servers, B1 and B2. A client request asks for the current balance of both of his accounts. The request processing steps will vary depending on the underlying architecture, whether it is the H-ORB, the F-ORB or the P-ORB.

The Client Request Path is the path a request takes starting from the client node, passing through all intermediate nodes, getting processed in the sought server on the destination node and all the way back to the originating client. Each client requires one service from both of the server classes A and B and these services are assumed to be independent of each other and can be invoked concurrently.

The client request path varies depending on the underlying ORB architecture. In the H-ORB, the client gets the address of the server from the agent and communicates with the server directly. In the F-ORB, the client request is forwarded by the agent to the appropriate server. The server then returns the results of the computations to the client. In the P-ORB, the agent combines the two requests, forwards them concurrently to both servers, waits for the arrival of the two results, then combines the two results and sends them back to the client.

5.1.3 Workload Factors

Five factors that strongly affect the workload in [Abd-97] were investigated:

- ***Message Size (L):*** the length of the message (in bytes) the client sends per request to the sought server which in turn sends it back.

- ***Request Service Times S_A and S_B :*** the total CPU demand that each client request needs at each class of servers.

- ***Number of clients (N):*** the total number of active clients during the life of the experiment.

- ***Inter-Node Communication Delay (D):*** Client nodes may be several nodes away from agent and server nodes in a geographically dispersed distributed system. A delay is expected to occur with the increase of the number of intermediate nodes. Forcing the sending process to sleep for D time units simulates the delay experienced by a particular request.

- ***Degree of Cloning*:** the degree of cloning used for the agent process. A clone of a process is its copy that shares a message queue with its parent.

The following table summarizes the values used for the workload factors:

Factors	***Levels***
N	1,2,4,8,16,24
D (msec)	200, 250, 500, 1000
L (bytes)	4800, 9600, 19200
SA / SB (msec)	10/15, 50/75, 250/375
Degree Of Cloning	1, 4, 8

Table 1 : Levels for the Workload Factors

5.2 H-ORB

The handle-driven ORB is an architecture in which the agent returns a server's handle back to the requesting client. This handle contains all the information required to interact with the server application. Using this handle, the client can invoke a remote procedure or send a message to the server, which in turn sends the reply back to the client.

5.2.1 The H-ORB Request Path

In the Handle-driven ORB, the client request path proceeds as follows:

- The client obtains the handle for a server of class A from the default agent.
- The client invokes the server it has obtained the handle for (A1 or A2).
- The client obtains the handle for a server of class B from the default agent.
- The client invokes the server it has obtained the handle for (B1 or B2).

The Agent in the H-ORB is called the **Default Agent** and is supplied by ORBeline. Obtaining the source-code for the ORBeline default agent was not possible in [Abd-97]. Since the measurements were done on a local area network with short communication delays, an artificial network delay was introduced in the experiments. A **sleeping pattern** was developed to simulate this network delay. The client sleeps twice: first before sending to the default agent, and second after receiving the handle. Also, the client sleeps before sending the request to a server, while the server sleeps prior to sending the reply back to the client.

5.2.2 H-ORB in UML

The automation process starts with the UML specification of the software system. A full description of the H-ORB architecture is not given here. Instead, only the Client Request Path and the process allocation need to be specified in UML to serve as input for the automation process. This section gives these specifications in UML for the H-ORB architecture.

Sequence Diagram:

The Client Request Path of the H-ORB architecture could be described in UML using the following sequence diagram:

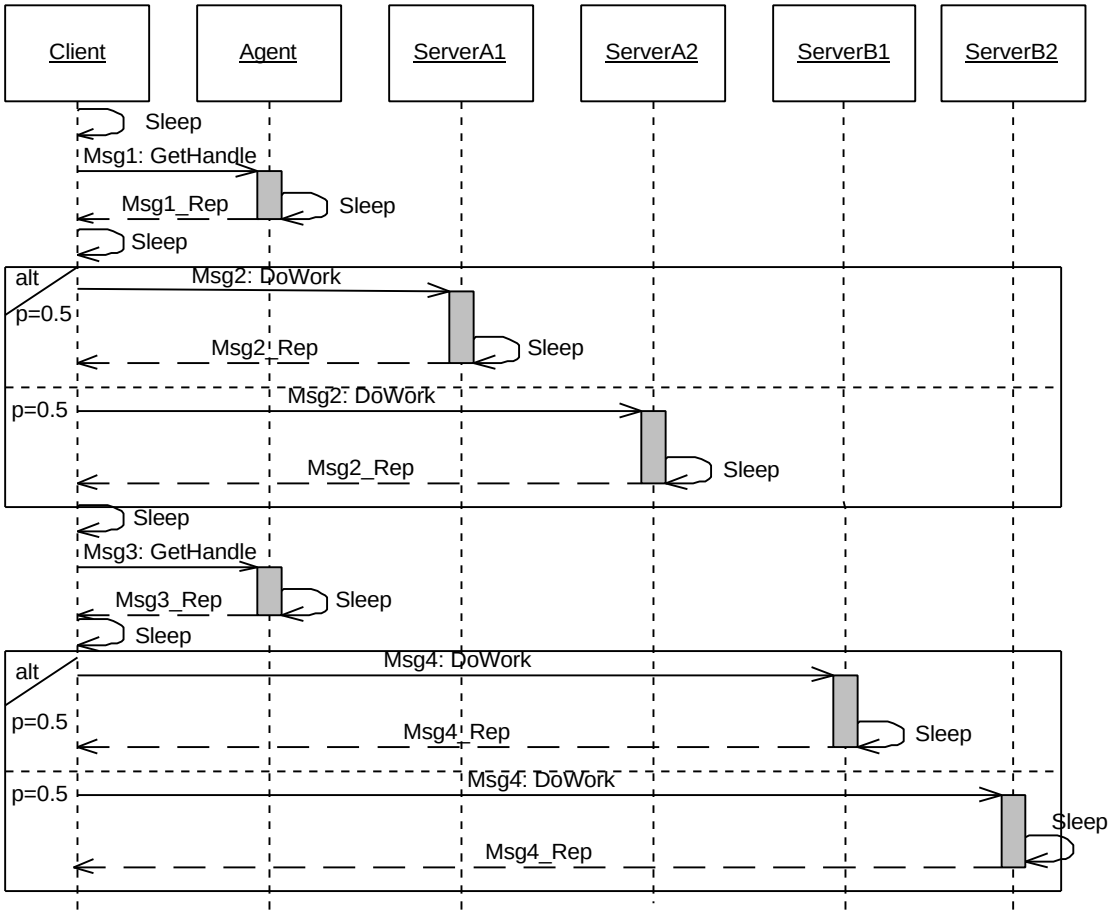


Figure 33: H-ORB Sequence Diagram

Note that all the client calls are synchronous; hence, the client is blocked waiting for the reply for its requests to both the servers and the default agent. The client chooses randomly between sending the request to ServerA1 or to ServerA2. The probability of choosing one path is equal to 50%, as shown as an appended annotation on the figure. The alternate path exists also when choosing between sending the request to ServerB1 or to ServerB2.

Collaboration Diagram:

A collaboration can be used to specify the implementation of design constructs; their context and interactions. This could be used to identify the presence of design patterns within a system design. In the H-ORB case, we can identify the CLIENT-SERVER relationship between the Client and the Agent, and the Client and ServerA1, Server A2, ServerB1 and ServerB2 objects. This relationship is illustrated in the following figure.

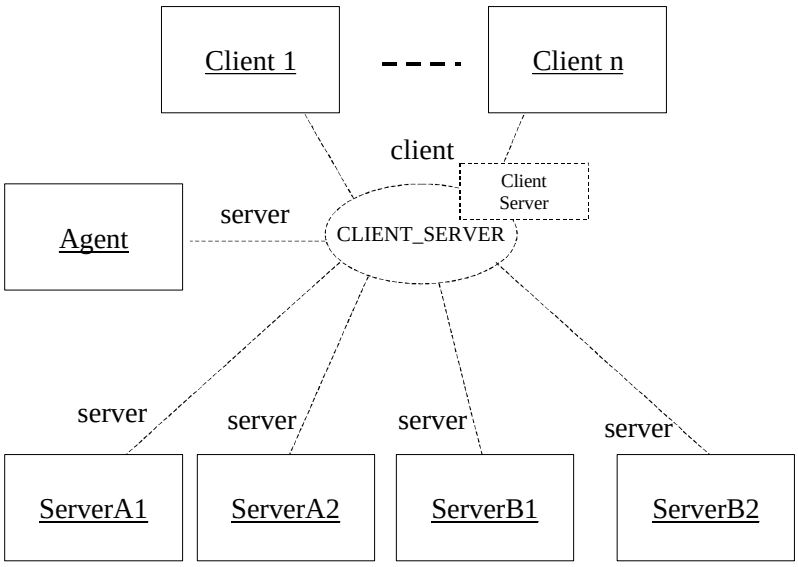


Figure 34: H-ORB Collaboration Diagram

Deployment Diagram:

The LQN model needs to know some information about the participating tasks and processors. For example, it needs to know as input the different processors, their multiplicity, scheduling, and connections, as well as the way tasks are allocated on them. The multiplicity of processor and tasks is also needed for input. This information is deduced from the case study in [Abd-97] as follows:

- There are two instances of the ServerA task (A1 and A2), each allocated on a separate processor.

- There are two instances of the ServerB task (B1 and B2), each allocated on a separate processor.

- The number of client tasks (N) varies as a workload input. Each client is allocated on a different processor (processor multiplicity = infinite).

- The agent can handle many requests at the same time. It has been modeled here as an infinite task on a single processor.

Processor and Network speed:

The service time per request is equal to the number of cycles a request needs from a task on a certain processor divided by the speed of this processor (in cycles per msec). Since we have as input the service time per request for both servers, we have set the processor speed for all processors to be 1 unit (say X cycles per msec). At the same time, we required as input to the

PROGRES program the number of cycles required per request. This means that we can specify 10 cycles (actually 10 * X cycles) for a service time that we know will take 10 (msec) to finish.

In [Abd-97], a LAN of 10 Mbps was used through out all experiments. Therefore, the network speed was set to be 1000 bits per msec as input to the PROGRES program. The following deployment diagram describes the H-ORB tasks and processors and their allocation in UML.

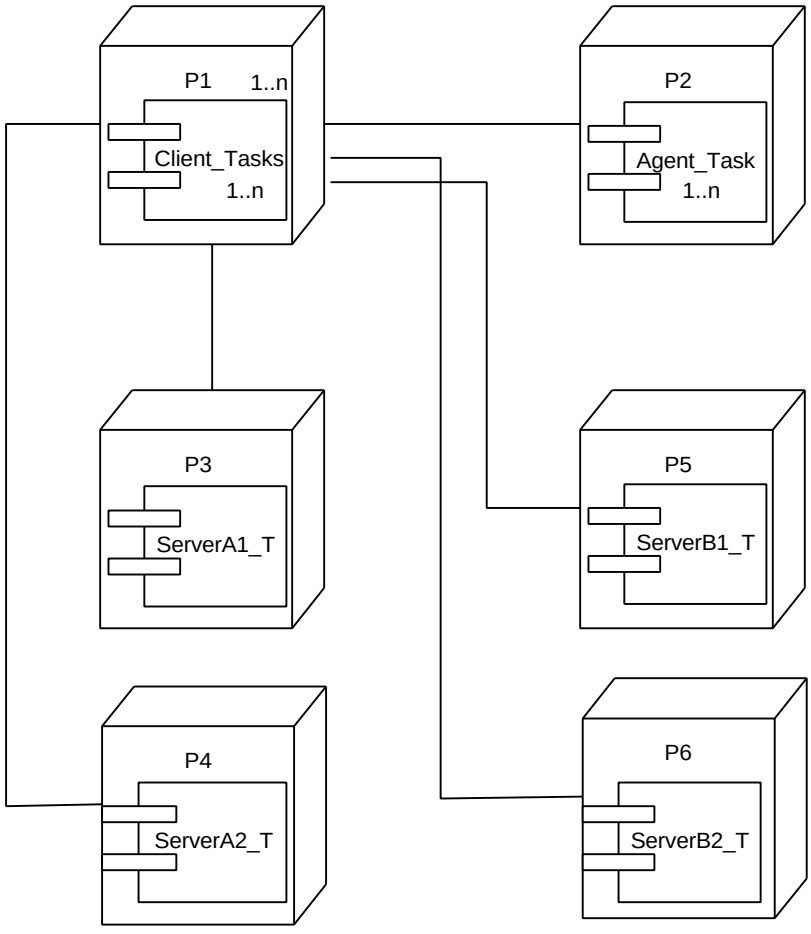


Figure 35: H-ORB Deplyment Diagram

5.2.3 H-ORB in PROGRES

The following table summarizes the workload values chosen in this work as input to the PROGRES program. It is comparable to the values of the case study experiment whose output is illustrated in Figure A-1 on page A-2 of appendix "A" of [Abd-97].

Selected workload values:

Factors	***Levels***
N	1,2,4,8,16,24
D (msec)	200
L (bytes)	4800
SA / SB (msec)	10/15
Degree Of Cloning	1
Agent service time (msec)	4

Table 2: H-ORB Workload Factors Selected Values

All performance parameters, such as service times and inter-node delays, were extracted from [Abd-97]. The Agent service time per request was not explicitly stated in [Abd-97]. However, it was not small enough to be ignored. Its value (4 msec) was inferred from the measurements.

5.2.3.1 PROGRES Input

Three functions determine the input for the PROGRES program; "ConfigurePlatform", "EstablishCollaborationInfo", and "CreateProblem". In the first one, we create the different processors and tasks. We specify for each processor its name, multiplicity, scheduling and speed. For tasks, we specify the task name and multiplicity as well as on which processor it will be allocated. Note that a multiplicity of 0 means infinity for both processors and tasks. Finally, we create communication links between processors and specify their transmission

speed in bits per msec. The input to this function is extracted mainly from the H_ORB Deployment diagram (see Figure 35).

```
transaction ConfigurePlatform =
      begin
           NewProcessorNode ( "P1", 0, "f", "1.0" )
         & NewProcessorNode ( "P2", 0, "f", "1.0" )
         & NewProcessorNode ( "P3", 1, "f", "1.0" )
         & NewProcessorNode ( "P4", 1, "f", "1.0" )
         & NewProcessorNode ( "P5", 1, "f", "1.0" )
         & NewProcessorNode ( "P6", 1, "f", "1.0" )
         & NewTaskComponent ( "T1", "P1", 1 )
         & NewTaskComponent ( "T2", "P2", 0 )
         & NewTaskComponent ( "T3", "P3", 1 )
         & NewTaskComponent ( "T4", "P4", 1 )
         & NewTaskComponent ( "T5", "P5", 1 )
         & NewTaskComponent ( "T6", "P6", 1 )
         & LinkProcessorNodes ( "P1", "P2", "10000.0" )
         & LinkProcessorNodes ( "P1", "P3", "10000.0" )
         & LinkProcessorNodes ( "P1", "P4", "10000.0" )
         & LinkProcessorNodes ( "P1", "P5", "10000.0" )
         & LinkProcessorNodes ( "P1", "P6", "10000.0" )
      end
   end;
```

Figure 36: H-ORB ConfigurePlatform Function.

In the second function, some pre-known relationships between objects in collaboration are established. For example, in the H-ORB architecture, we know that there is a Client-Server relationship between the Client and the Agent, and the Client and each of the two servers. The input to this function is extracted mainly from the H_ORB Collaboration diagram (see Figure 34: H-ORB Collaboration Diagram).

```
transaction EstablishCollaborationInfo =
   begin
        AddClientServerRelationship ( "Client", "HORB" )
      & AddClientServerRelationship ( "Client", "SERVER_A1" )
      & AddClientServerRelationship ( "Client", "SERVER_A2" )
      & AddClientServerRelationship ( "Client", "SERVER_B1" )
      & AddClientServerRelationship ( "Client", "SERVER_B2" )
   end
end;
```

Figure 37: H-ORB EstablishCollaborationInfo Function

Finally, the sequence diagram is described step by step in the “CreateProblem” function. The input to this function is extracted mainly from the H_ORB Sequence diagram (see Figure 33). All API in all functions are used as described previously in Sections 3.2.2.1 and 4.2.2.1.

Modeling Task Delay

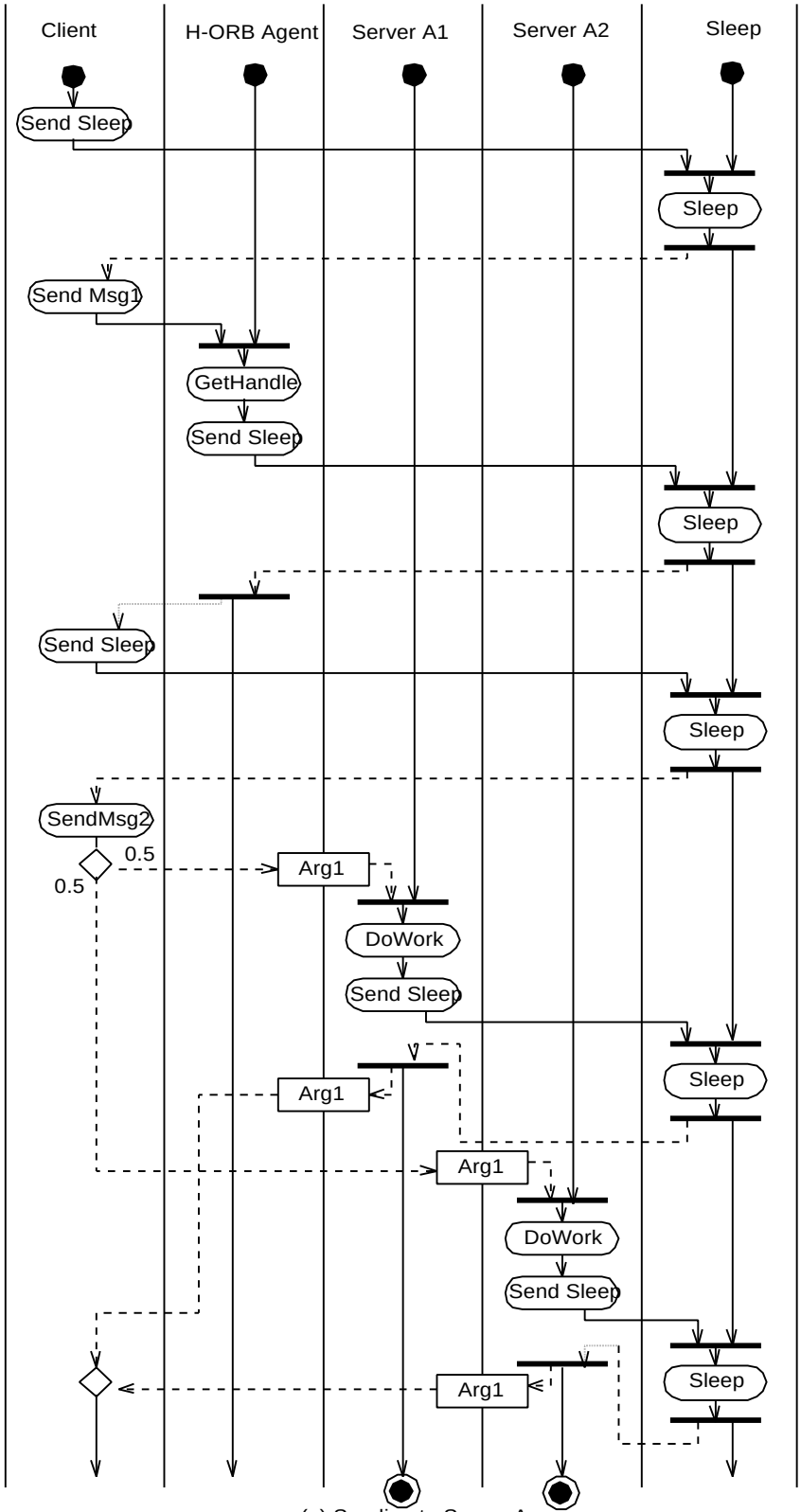
In this work, a simple task delay is modeled as a synchronous call to an infinite task that runs on an infinite processor. The function “Sleep”, introduced in section 3.2.2.2, was used to indicate the sleeping of an active object. When sleep is called, a check is done to see if the sleep task was created or not. If not, an infinite task named “Sleep_T” is created on an infinite processor “Sleep_P”. A Client-Server relationship between the active thread and this task is also created. Then a Synchronous call from this active thread to the Sleep Task is done with an expected number of cycles equals to the sleeping time. A reply from the Sleeping task to the active thread is then created and execution proceeds normally. If an active object calls sleep and there is already a Sleep task, the creation part is skipped and the Synchronous call, along with its relay are created. Calling the Sleep function mimics the sleeping pattern that was used in [Abd-97] case study explained above.

```
transaction CreateProblem =
    use C_Inst, H_Inst, SA1_Inst, SA2_Inst, SB1_Inst, SB2_Inst : INSTANCE;
      MainAS, AS1, AS2, AS3, AS4 : COMPOSITE_ACTION
    do
       CreateMainActionSequence ( "MainActionSequence", out MainAS )
      & CreateNewInstance ( "Client", "T1", true, out C_Inst )
      & CreateNewInstance ( "HORB", "T2", true, out H_Inst )
      & CreateNewInstance ( "SERVER_A1", "T3", true, out SA1_Inst )
      & CreateNewInstance ( "SERVER_A2", "T4", true, out SA2_Inst )
      & CreateNewInstance ( "SERVER_B1", "T5", true, out SB1_Inst )
      & CreateNewInstance ( "SERVER_B2", "T6", true, out SB2_Inst )
      & Sleep ( C_Inst, 1, MainAS, 200 )
      & CreateSyncCall ( C_Inst, H_Inst, "Msg1", "", 0, "", 3, MainAS, 0 )
      & CreateLocalAction ( H_Inst, "FindAddress", "", 4, MainAS, 4 )
      & Sleep ( H_Inst, 5, MainAS, 200 )
      & CreateReplyCall ( H_Inst, C_Inst, "Msg1_Rep", "", 0, "", 7, MainAS, 0 )
      & Sleep ( C_Inst, 8, MainAS, 200 )
      & CreateActionSequence ( C_Inst, "AS1", "0.5", 10, MainAS, 0, out AS1 )
      & CreateActionSequence ( C_Inst, "AS2", "0.5", 10, MainAS, 0, out AS2 )
      & CreateSyncCall ( C_Inst, SA1_Inst, "Msg2", "ARG1", 4800, "", 1, AS1, 0 )
      & CreateSyncCall ( C_Inst, SA2_Inst, "Msg2", "ARG1", 4800, "", 1, AS2, 0 )
      & CreateLocalAction ( SA1_Inst, "DoWork", "", 2, AS1, 10 )
      & CreateLocalAction ( SA2_Inst, "DoWork", "", 2, AS2, 10 )
      & Sleep ( SA1_Inst, 3, AS1, 200 )
      & CreateReplyCall ( SA1_Inst, C_Inst, "Msg2_Rep", "ARG1", 4800, "", 5, AS1, 0 )
      & Sleep ( SA2_Inst, 3, AS2, 200 )
      & CreateReplyCall ( SA2_Inst, C_Inst, "Msg2_Rep", "ARG1", 4800, "", 5, AS2, 0 )
      & Sleep ( C_Inst, 11, MainAS, 200 )
      & CreateSyncCall ( C_Inst, H_Inst, "Msg3", "", 0, "", 13, MainAS, 0 )
      & CreateLocalAction ( H_Inst, "FindAddress", "", 14, MainAS, 4 )
      & Sleep ( H_Inst, 15, MainAS, 200 )
      & CreateReplyCall ( H_Inst, C_Inst, "Msg3_Rep", "", 0, "", 17, MainAS, 0 )
      & Sleep ( C_Inst, 18, MainAS, 200 )
      & CreateActionSequence ( C_Inst, "AS3", "0.5", 20, MainAS, 0, out AS3 )
      & CreateActionSequence ( C_Inst, "AS4", "0.5", 20, MainAS, 0, out AS4 )
      & CreateSyncCall ( C_Inst, SB1_Inst, "Msg4", "ARG2", 4800, "", 1, AS3, 0 )
      & CreateSyncCall ( C_Inst, SB2_Inst, "Msg4", "ARG2", 4800, "", 1, AS4, 0 )
      & CreateLocalAction ( SB1_Inst, "DoWork", "", 2, AS3, 15 )
      & CreateLocalAction ( SB2_Inst, "DoWork", "", 2, AS4, 15 )
      & Sleep ( SB1_Inst, 3, AS3, 200 )
      & CreateReplyCall ( SB1_Inst, C_Inst, "Msg4_Rep", "ARG2", 4800, "", 5, AS3, 0 )
      & Sleep ( SB2_Inst, 3, AS4, 200 )
      & CreateReplyCall ( SB2_Inst, C_Inst, "Msg4_Rep", "ARG2", 4800, "", 5, AS4, 0 )
      & TerminateInstance ( C_Inst, 21, MainAS, 0 )
      & TerminateInstance ( H_Inst, 22, MainAS, 0 )
      & TerminateInstance ( SA1_Inst, 23, MainAS, 0 )
      & TerminateInstance ( SA2_Inst, 24, MainAS, 0 )
      & TerminateInstance ( SB1_Inst, 25, MainAS, 0 )
      & TerminateInstance ( SB2_Inst, 26, MainAS, 0 )
    end
  end;
```

Figure 38: H-ORB CreateProblem Function

5.2.3.2 PROGRES Output

The first phase of transformation generates a description of an Activity diagram in text format. The graphical representation of this Activity diagram is given in the following figure.



(a) Sending to Server A

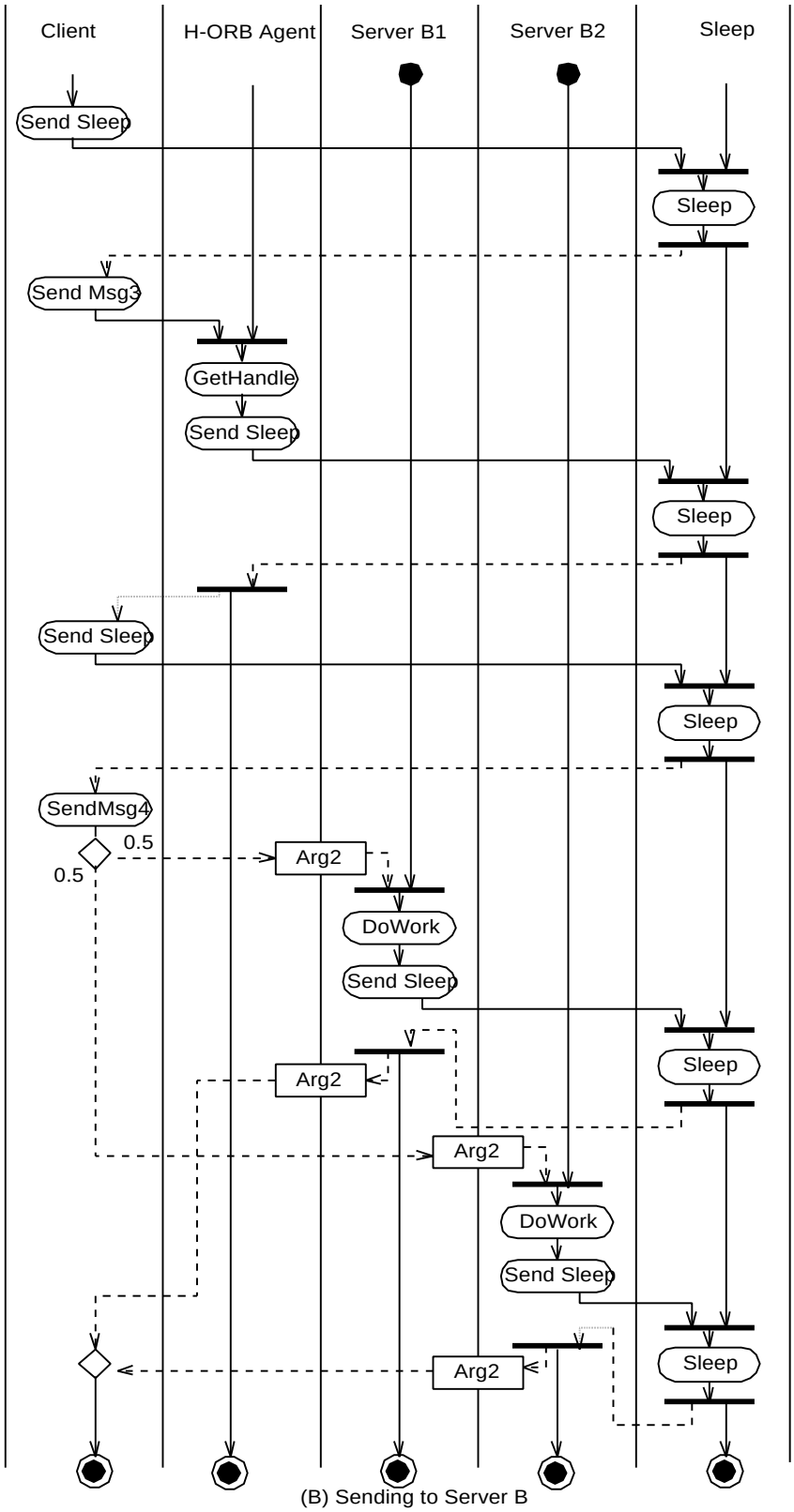


Figure 39: H-ORB Activity Graph

The PROGRES program was run, and it produced as output the LQN performance model given in the following figure. The textual description of this Model is illustrated in Figure 64, Appendix A.

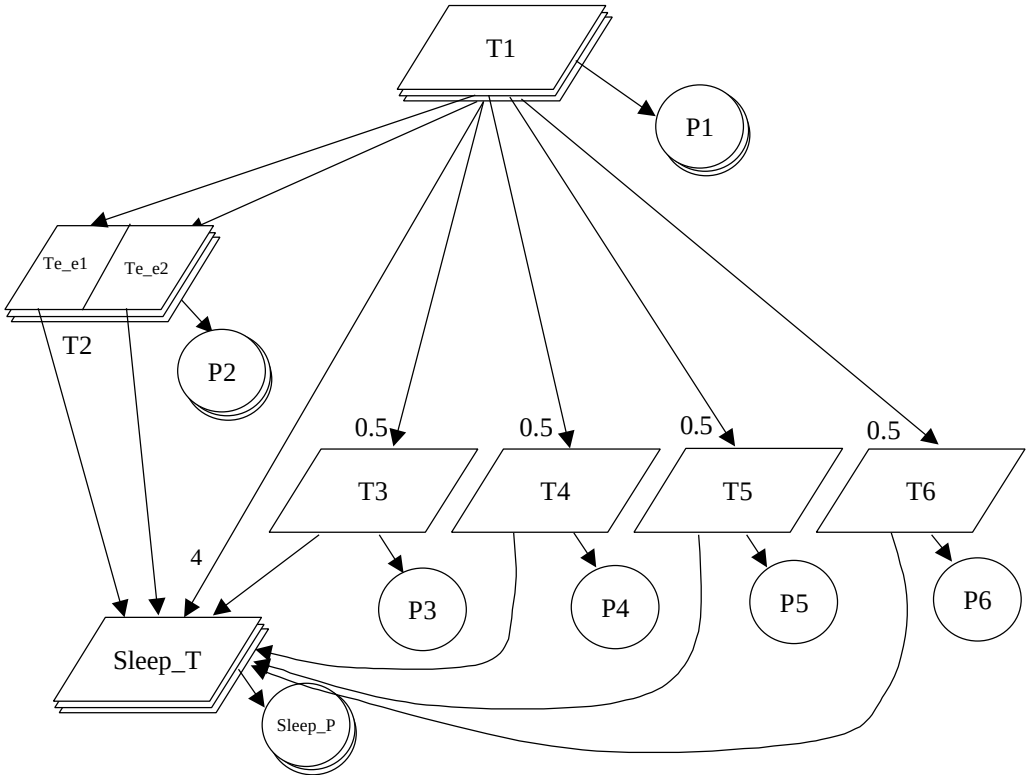


Figure 40: H-ORB LQN Performance Model

The calculated service times for each entry is summarized in the following table:

Entry	***Service Time***
T1_e1	807.68
T2_e1	204
T2_e2	204
T3_e1	213.84
T4_e1	218.84

Table 3: H-ORB Calculated Service Time per Entry

The model was then used as input to the LQN solver (LQNS), giving the following results for the Mean Client Response Time (in seconds) for the different values of the number of clients (N).

Number of Clients	*Model Results*	*Measured Values*	*Error %*
1	1.64836	1.7212	4.231931211
2	1.70365	1.7212	1.019637462
4	1.82544	1.75	-4.310857143
8	2.11086	1.9	-11.09789474
16	2.80472	2.5	-12.1888
24	3.58537	3.2	-12.0428125

Table 4: H-ORB Model Results VS Measured Values

A comparison between the results obtained and the measured values is depicted in the following graph.

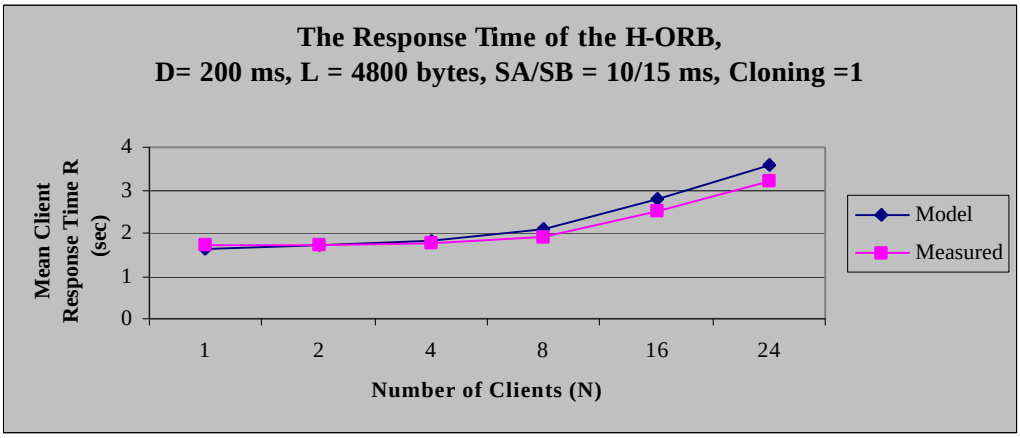


Figure 41: H-ORB Model results VS Measured values Graph

5.3 F-ORB

As stated in [Abd-97] case study, the Forwarding ORB (F-ORB) architecture differs from the previous architecture in the sense that the f-agent forwards the reply to the sought server rather than returning the handle to the requesting client. During each experiment, a fixed number of F-agents (number is equal to the chosen degree of cloning) are implemented. All F-agents are activated and set ready to receive and process any client request in cooperation with the default agent supplied by ORBeline. The F-Agent and the default agent are co-allocated on the same processor and are treated as one task.

5.3.1 The F-ORB Request Path

In the Forwarding ORB, the client request path proceeds as follows:

- The client selects an F-agent randomly and uses its handle to send the request.
- F-agent obtains handle for a server of class A (A1 or A2) and uses the handle obtained to relay the request to the server.
- The Server of class A returns the reply to the originating client
- The client uses F-agent's handle to forward its request destined for a class B (B1 or B2) server.
- F-agent obtains the handle for a class B server and uses the handle obtained to relay the request to the server.
- The Server of class B returns the reply to the originating client

All path steps use Asynchronous (send one way) calls to the called process. To simulate inter-node delay, all processes sleep before sending any call.

5.3.2 F-ORB in UML

This section gives the Sequence, the Collaboration, and the Deployment diagram of the F-ORB architecture.

Sequence Diagram: The Client Request Path of the F-ORB architecture could be described in UML using the following sequence diagram:

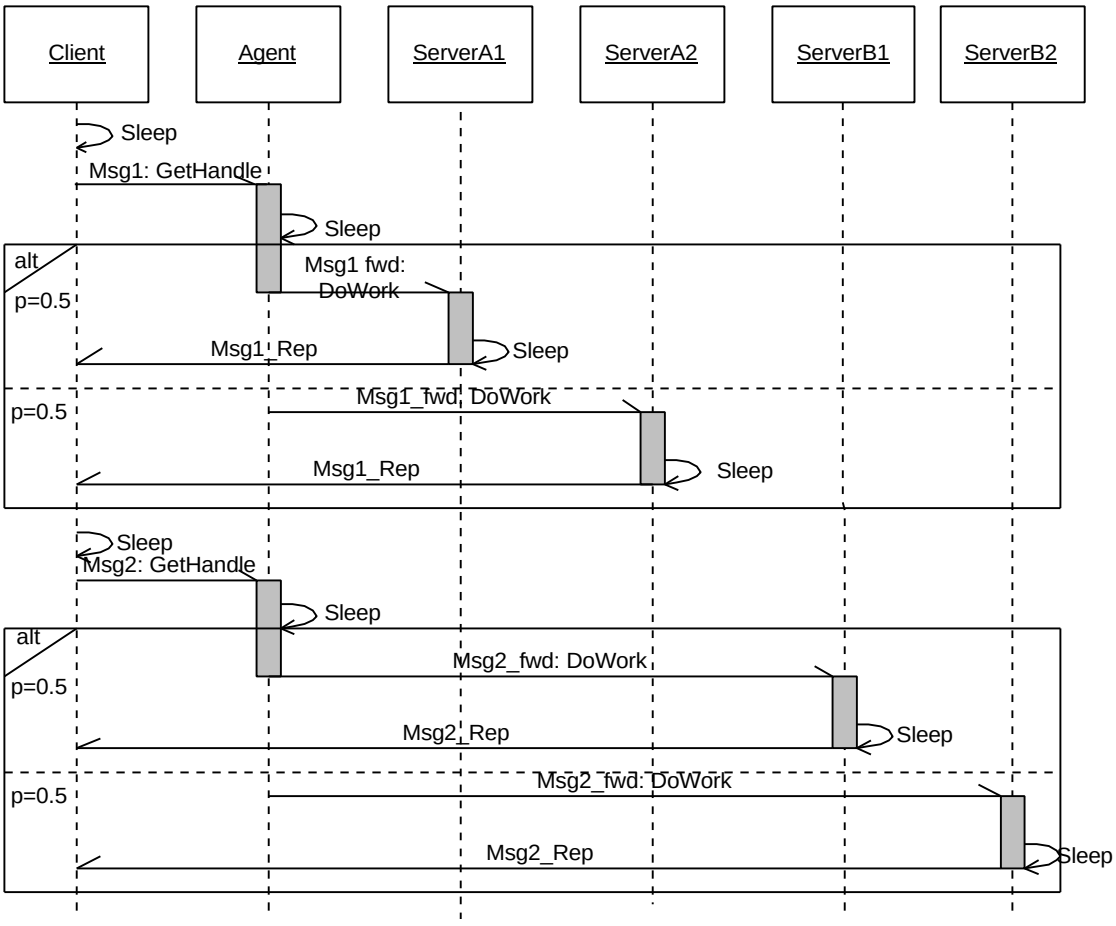


Figure 42: F-ORB Sequence Diagram

Collaboration Diagram:

A collaboration can be used to specify the implementation of design constructs; their context and interactions. This could be used to identify the presence of design patterns within a system design. In the F-ORB case, we can identify the FORWARDING relationship between the Client the Agent, and the two instances of ServerA (A1 and A2), and between the Client, the Agent and the two instances of ServerB (B1 and B2). This relationship is illustrated in the following figure.

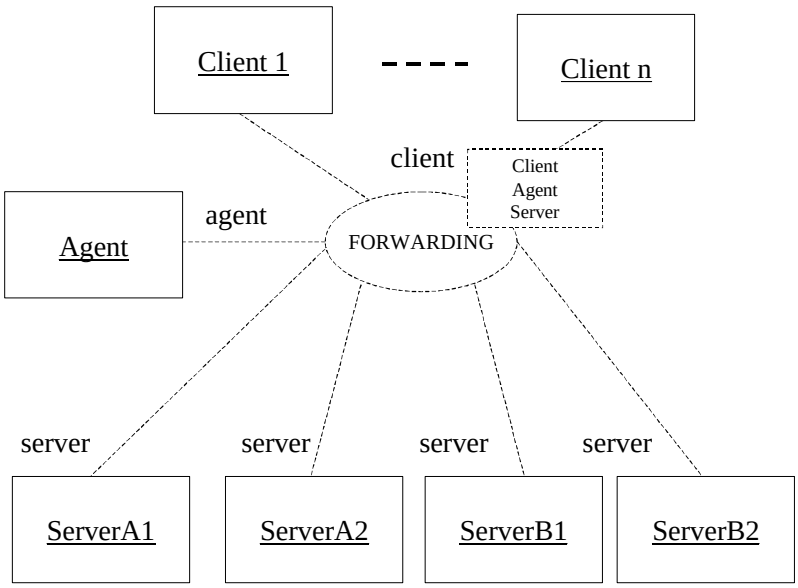


Figure 43: F-ORB Collaboration Diagram

Deployment Diagram:

The Deployment diagram of the F-ORB is similar to that of the H-ORB. The main differences are the links between processors and the multiplicity of the agent task. In the F-

ORB case, the F-agent is linked to both servers on top of all the links of the H-ORB. The multiplicity of the F-Agent task is controlled by the case study in [Abd-97] and is equal to the chosen degree of cloning.

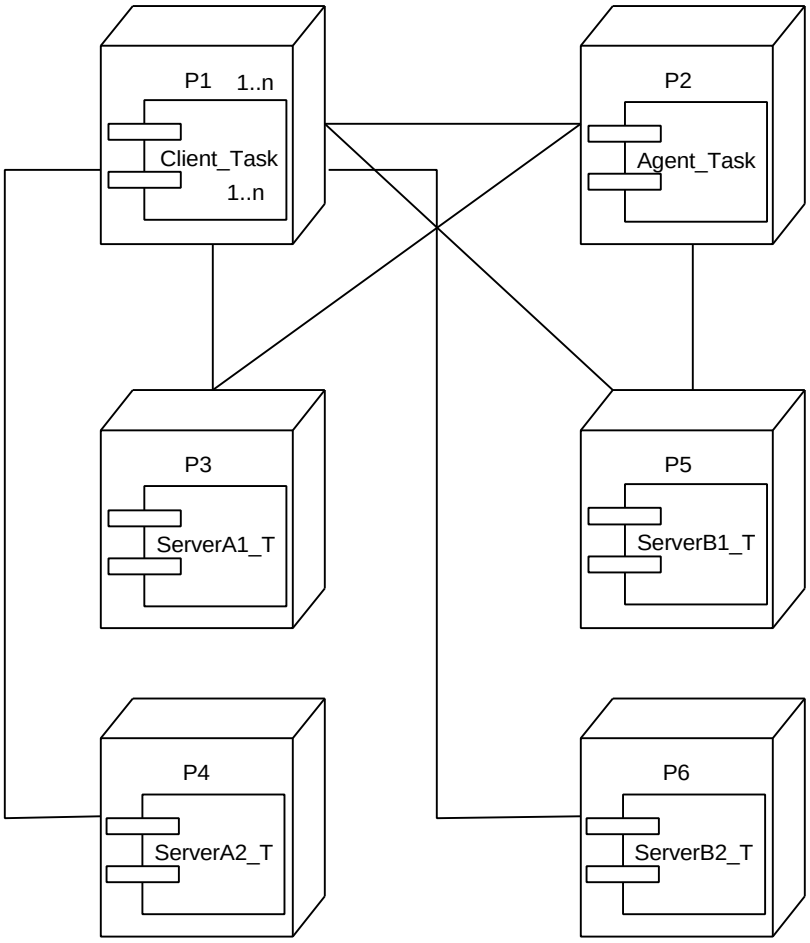


Figure 44: F-ORB Deployment Diagram

5.3.3 F-ORB in PROGRES

The selected workload factors of the F-ORB in this work are the same as the ones previously selected for the H-ORB, (summarized in Table 2: H-ORB Workload Factors Selected

Values). It is comparable to the values of [Abd-97] case study experiment whose output is illustrated in Figure A-1 on page A-2 of appendix “A” of [Abd-97].

5.3.3.1 PROGRES Input

The three functions that determine the input for the PROGRES program are shown in the following three figures. The main differences between the F-ORB and the previously described H-ORB in the “ConfigPlatform” are the processor links and the Agent task multiplicity. The input to this function is extracted mainly from the F_ORB Deployment diagram (see Figure 44).

```
transaction ConfigurePlatform =
   begin
        NewProcessorNode ( "P1", 0, "f", "1.0" )
      & NewProcessorNode ( "P2", 1, "f", "1.0" )
      & NewProcessorNode ( "P3", 1, "f", "1.0" )
      & NewProcessorNode ( "P4", 1, "f", "1.0" )
      & NewProcessorNode ( "P5", 1, "f", "1.0" )
      & NewProcessorNode ( "P6", 1, "f", "1.0" )
      & NewTaskComponent ( "T1", "P1", 1 )
      & NewTaskComponent ( "T2", "P2", 1 )
      & NewTaskComponent ( "T3", "P3", 1 )
      & NewTaskComponent ( "T4", "P4", 1 )
      & NewTaskComponent ( "T5", "P5", 1 )
      & NewTaskComponent ( "T6", "P6", 1 )
      & LinkProcessorNodes ( "P1", "P2", "10000.0" )
      & LinkProcessorNodes ( "P1", "P3", "10000.0" )
      & LinkProcessorNodes ( "P1", "P4", "10000.0" )
      & LinkProcessorNodes ( "P1", "P5", "10000.0" )
      & LinkProcessorNodes ( "P1", "P6", "10000.0" )
      & LinkProcessorNodes ( "P2", "P3", "10000.0" )
      & LinkProcessorNodes ( "P2", "P4", "10000.0" )
      & LinkProcessorNodes ( "P2", "P5", "10000.0" )
      & LinkProcessorNodes ( "P2", "P6", "10000.0" )
   end
end;
```

Figure 45: F-ORB ConfigPlatform Function

The relationship between the objects Client, Agent, and Server is not a client-server anymore. Instead, it could be seen as a forwarding relationship between the three entities. This

relationship is established using the function “AddForwardingRelationship” between the three participating objects. The input to this function is extracted mainly from the F_ORB Collaboration diagram (see Figure 43).

```
transaction EstablishCollaborationInfo =
   begin
        AddForwardingRelationship ( "Client", "FORB", "SERVER_A1" )
      & AddForwardingRelationship ( "Client", "FORB", "SERVER_A2" )
      & AddForwardingRelationship ( "Client", "FORB", "SERVER_B1" )
      & AddForwardingRelationship ( "Client", "FORB", "SERVER_B2" )
   end
end;
```

Figure 46: F-ORB EstablishCollaborationInfo Function

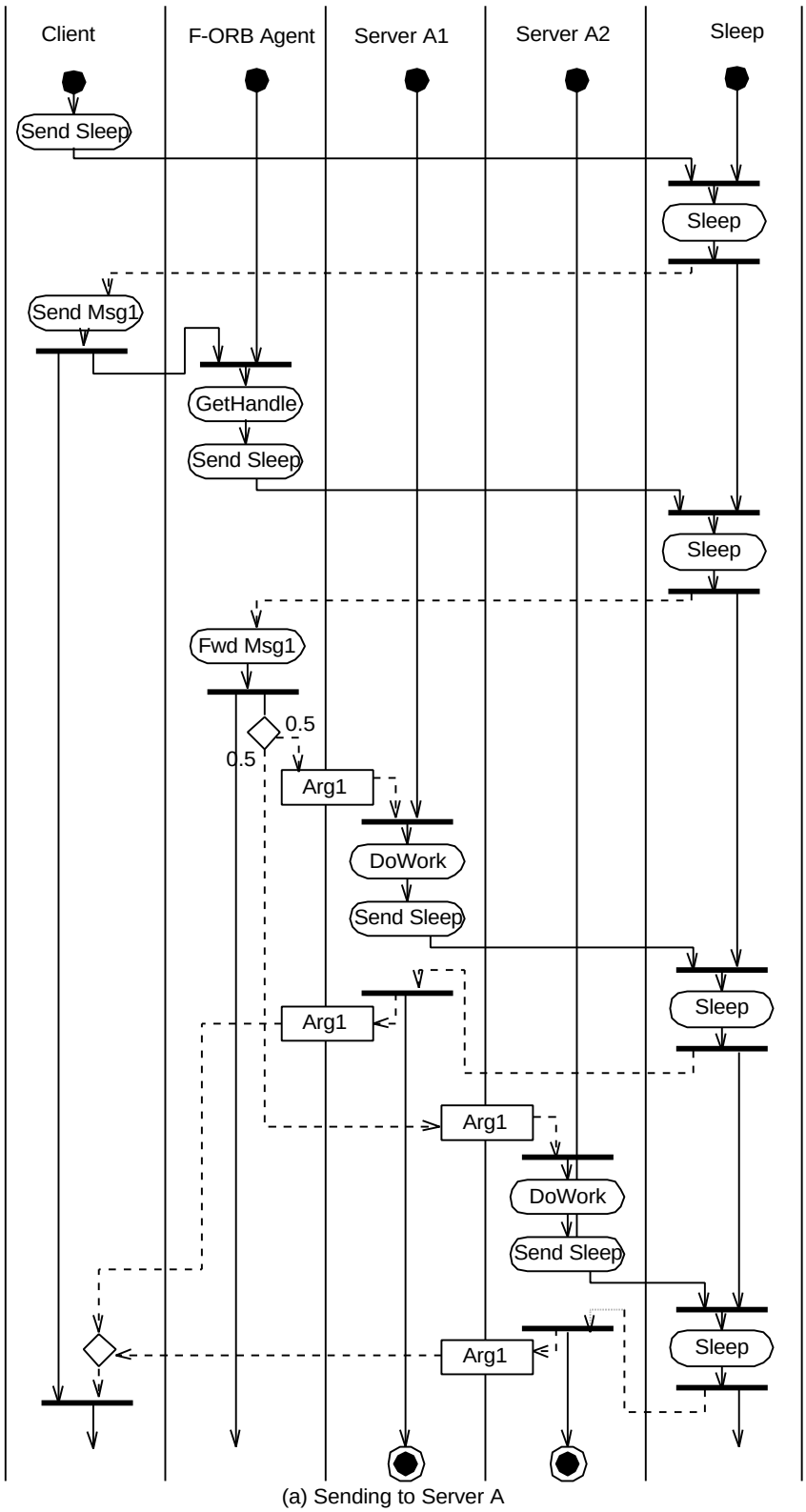
The sequence diagram is described step by step in the “CreateProblem” function. The input to this function is extracted mainly from the F_ORB Sequence diagram (see Figure 42). All API in all functions are used as described previously in Sections 3.2.2.1 and 4.2.2.1.

```
transaction CreateProblem =
    use C_Inst, F_Inst, SA1_Inst, SA2_Inst, SB1_Inst, SB2_Inst : INSTANCE;
      MainAS : COMPOSITE_ACTION
    do
       CreateMainActionSequence ( "MainActionSequence", out MainAS )
      & CreateNewInstance ( "Client", "T1", true, out C_Inst )
      & CreateNewInstance ( "FORB", "T2", true, out F_Inst )
      & CreateNewInstance ( "SERVER_A1", "T3", true, out SA1_Inst )
      & CreateNewInstance ( "SERVER_A2", "T4", true, out SA2_Inst )
      & CreateNewInstance ( "SERVER_B1", "T5", true, out SB1_Inst )
      & CreateNewInstance ( "SERVER_B2", "T6", true, out SB2_Inst )
      & Sleep ( C_Inst, 1, MainAS, 200 )
      & CreateAsyncCall ( C_Inst, F_Inst, "Msg1", "ARG1", 4800, "", 3, MainAS, 0 )
      & CreateLocalAction ( F_Inst, "FindAddress", "", 4, MainAS, 4 )
      & Sleep ( F_Inst, 5, MainAS, 200 )
      & CreateAsyncCall ( F_Inst, SA1_Inst, "Msg1", "ARG1", 4800, "0.5", 7, MainAS, 0 )
      & CreateLocalAction ( SA1_Inst, "DoWork" , "", 8, MainAS, 10 )
      & Sleep ( SA1_Inst, 9, MainAS, 200 )
      & CreateAsyncCall ( SA1_Inst, C_Inst, "Msg1_Rep", "ARG1", 4800, "", 11, MainAS, 0 )
      & CreateAsyncCall ( F_Inst, SA2_Inst, "Msg1", "ARG1", 4800, "0.5", 7, MainAS, 0 )
      & CreateLocalAction ( SA2_Inst, "DoWork", "", 12, MainAS, 10 )
      & Sleep ( SA2_Inst, 13, MainAS, 200 )
      & CreateAsyncCall ( SA2_Inst, C_Inst, "Msg1_Rep", "ARG1", 4800, "", 15, MainAS, 0 )
      & Sleep ( C_Inst, 16, MainAS, 200 )
      & CreateAsyncCall ( C_Inst, F_Inst, "Msg2", "ARG2", 4800, "", 18, MainAS, 0 )
      & CreateLocalAction ( F_Inst, "FindAddress", "", 19, MainAS, 4 )
      & Sleep ( F_Inst, 20, MainAS, 200 )
      & CreateAsyncCall ( F_Inst, SB1_Inst, "Msg2" , "ARG2", 4800, "0.5", 22, MainAS, 0 )
      & CreateLocalAction ( SB1_Inst, "DoWork", "", 23, MainAS, 10 )
      & Sleep ( SB1_Inst, 24, MainAS, 200 )
      & CreateAsyncCall ( SB1_Inst, C_Inst, "Msg2_Rep", "ARG2", 4800, "", 26, MainAS, 0 )
      & CreateAsyncCall ( F_Inst, SB2_Inst, "Msg2", "ARG2", 4800, "0.5", 22, MainAS, 0 )
      & CreateLocalAction ( SB2_Inst, "DoWork", "", 27, MainAS, 10 )
      & Sleep ( SB2_Inst, 28, MainAS, 200 )
      & CreateAsyncCall ( SB2_Inst, C_Inst, "Msg2_Rep", "ARG2", 4800, "", 30, MainAS, 0 )
      & TerminateInstance ( C_Inst, 31, MainAS, 0 )
      & TerminateInstance ( F_Inst, 32, MainAS, 0 )
      & TerminateInstance ( SA1_Inst, 33, MainAS, 0 )
      & TerminateInstance ( SA2_Inst, 34, MainAS, 0 )
      & TerminateInstance ( SB1_Inst, 35, MainAS, 0 )
      & TerminateInstance ( SB2_Inst, 36, MainAS, 0 )
    end
  end;
```

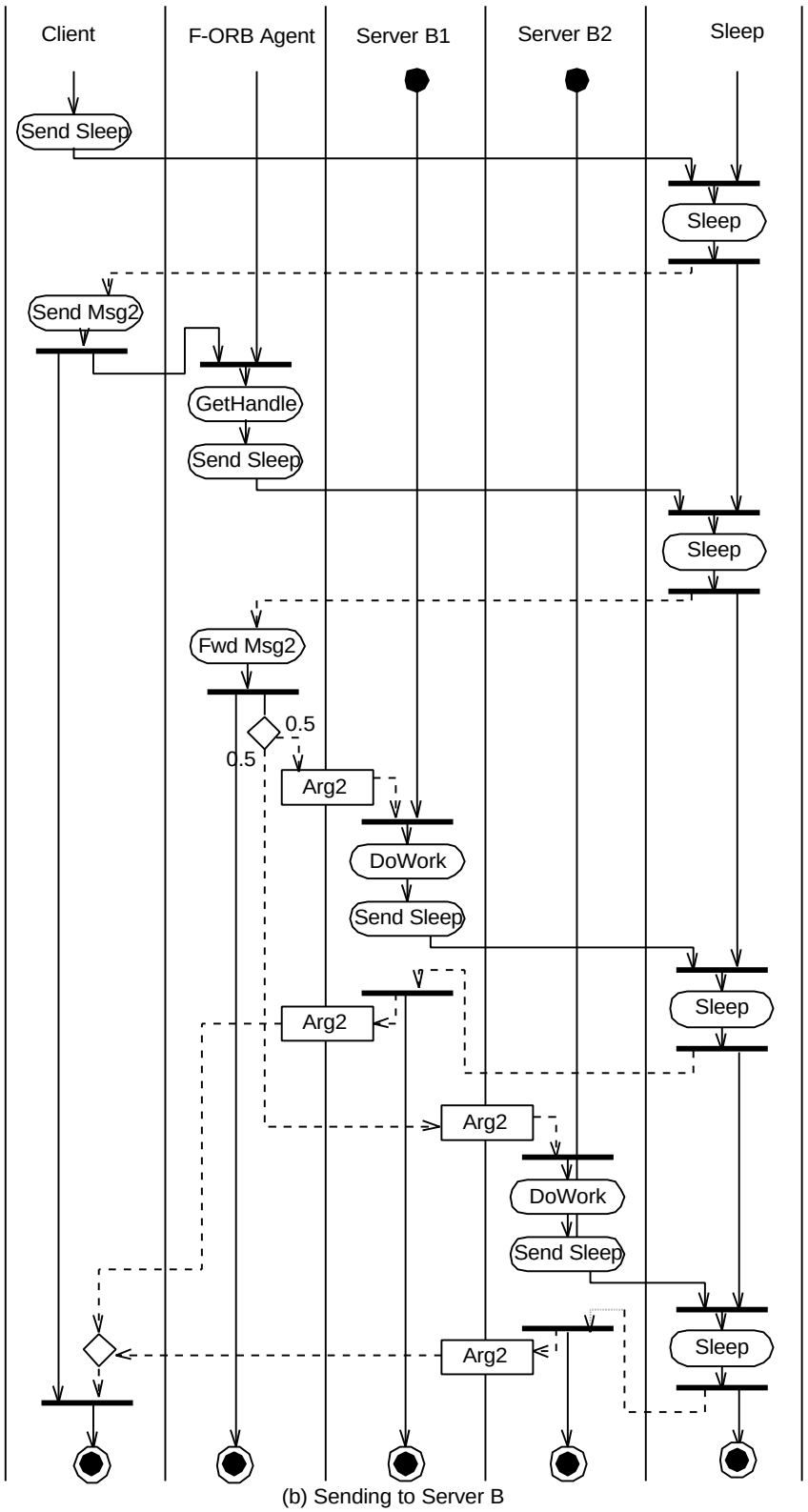
Figure 47: F-ORB CreateProblem Function

5.3.3.2 PROGRES Output

The first phase of transformation generates a description of an Activity diagram in text format. The graphical representation of this Activity diagram is given in the following figure.



(a) Sending to Server A



(b) Sending to Server B

Figure 48: F-ORB Activity Graph

When the PROGRES program was run, it produced as output the LQN given in the following figure. The textual description of this Model is illustrated in Figure 65, Appendix A.

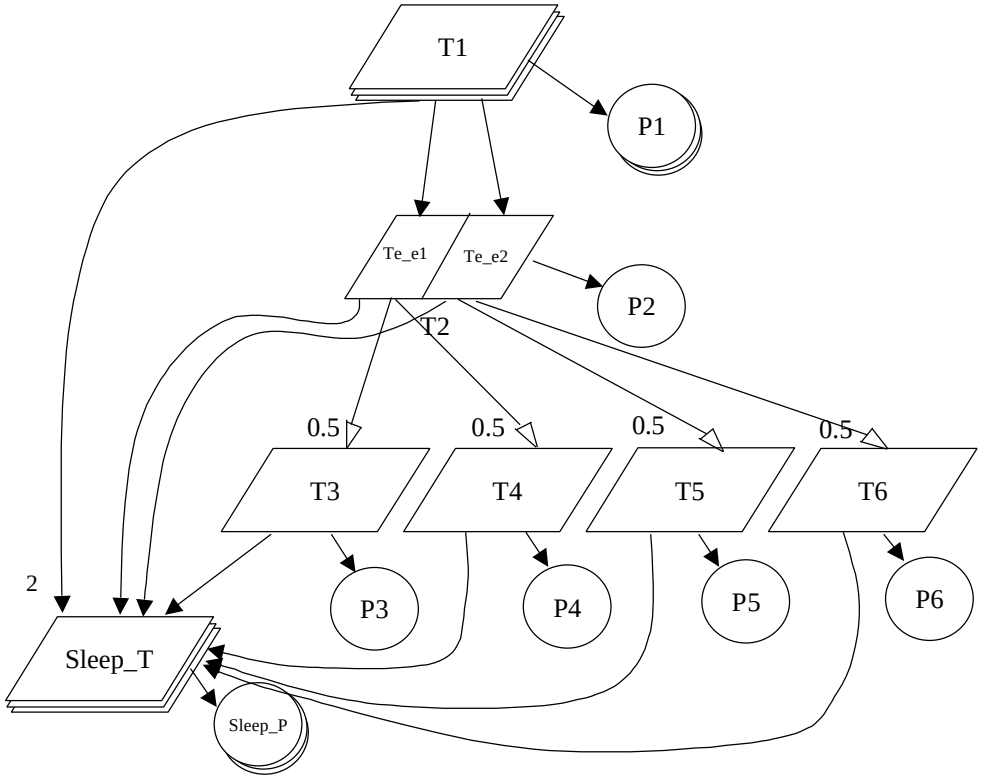


Figure 49: F-ORB LQN Performance Model

The calculated service times for each entry is summarized in the following table:

Entry	***Service Time***
T1_e1	407.68
T2_e1	207.84
T2_e2	207.84
T3_e1	213.84
T4_e1	218.84

Table 5: F-ORB Calculated Service Time per Entry

The model was then used as input to the LQN solver (LQNS), giving the following results for the Mean Client Response Time (in seconds) for the different values of the number of clients (N).

Number of Clients	*Model Results*	*Measured Values*	*Error %*
1	0.82336	1.3142	37.34895754
2	1.16632	1.32	11.64242424
4	2.0361	1.8	-13.11666667
8	3.74539	3.3	-13.49666667
16	7.09926	6.4	-10.9259375
24	10.4351	10	-4.351

Table 6: F-ORB Model Results VS Measured Values

A comparison between the results obtained and the measured values is depicted in the following graph.

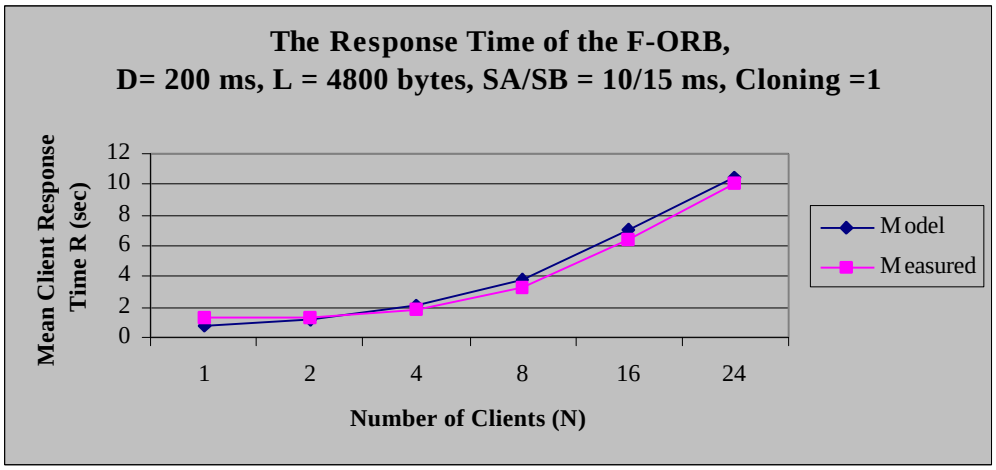


Figure 50: F-ORB Model Results VS Measured Values

5.4 P-ORB

In the Process Planner (P-ORB) architecture, the client sends its two requests combined in one message to an implemented P-agent. The P-agent decomposes the request into its simple constituent services, invokes the respective servers and when all services are performed, it relays back a single coherent reply to the originating client. The p-agent invokes both servers in a pseudo parallel asynchronous mode since the design assumes no interdependencies between the two constituent requests. Both servers are invoked using the send one-way call; however, there is only one inter-node delay involved. During each experiment, a fixed number of P-agents are activated and set ready to receive and process any client request in cooperation with the default agent supplied by ORBeline The P-agent and the default agent are co-allocated on the same processor and are considered to be one task.

5.4.1 The P-*ORB. Request Path*

In the P-ORB, the client request path proceeds as follows:

- The client selects a P-agent randomly and uses its handle to send the request.
- The client forwards the request to P-agent using send one-way call.
- The P-agent decomposes the request into its constituents and obtains a handle for a dispatcher of a server of class A, and a second handle for a dispatcher of a class B server.
- The P-agent uses these handles to relay the request to the respective servers in a pseudo parallel asynchronous mode (one simulated inter-node delay for both of the send one way calls).

- The selected Server of class A returns the reply to the calling p-agent in a send one-way mode after sleeping for the preset inter-node delay.

- The selected Server of class B returns the reply to p-agent in a pseudo asynchronous mode after sleeping the preset inter-node delay.

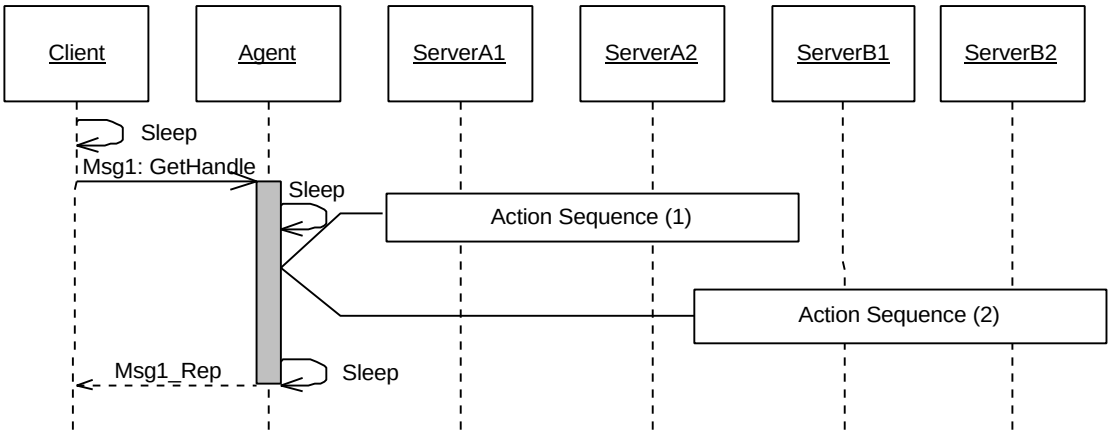
- The P-agent packs all constituent replies into a final, single and coherent reply and sends it to the originating client, after sleeping the preset inter-node delay.

5.4.2 P-ORB in UML

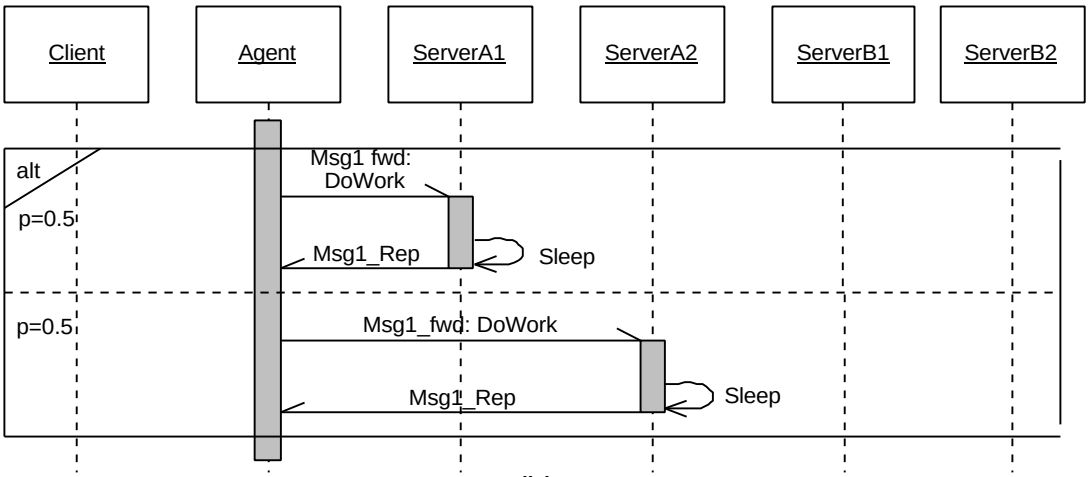
This section gives the Sequence diagram and the Deployment diagram of the P-ORB architecture.

Sequence Diagram:

The Client Request Path of the P-ORB architecture could be described in UML using the following sequence diagram:

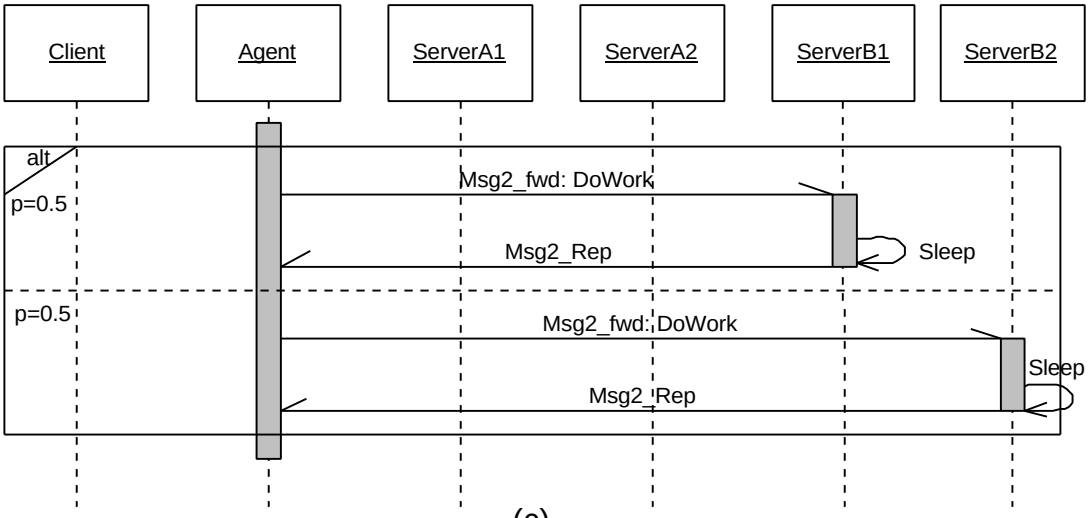


(a)
Main Action Sequence



(b)

Action Sequence (1): Agent sends message to Server A



(c)

Action Sequence (2): Agent sends message to Server B

Figure 51: P-ORB Sequence Diagram

Collaboration Diagram:

A collaboration can be used to specify the implementation of design constructs; their context and interactions. This could be used to identify the presence of design patterns within a system design. In the P-ORB case, we can identify the CLIENT-SERVER relationship between the Client and the Agent, the Agent and ServerA, and the Agent and ServerB objects. This relationship is illustrated in the following figure.

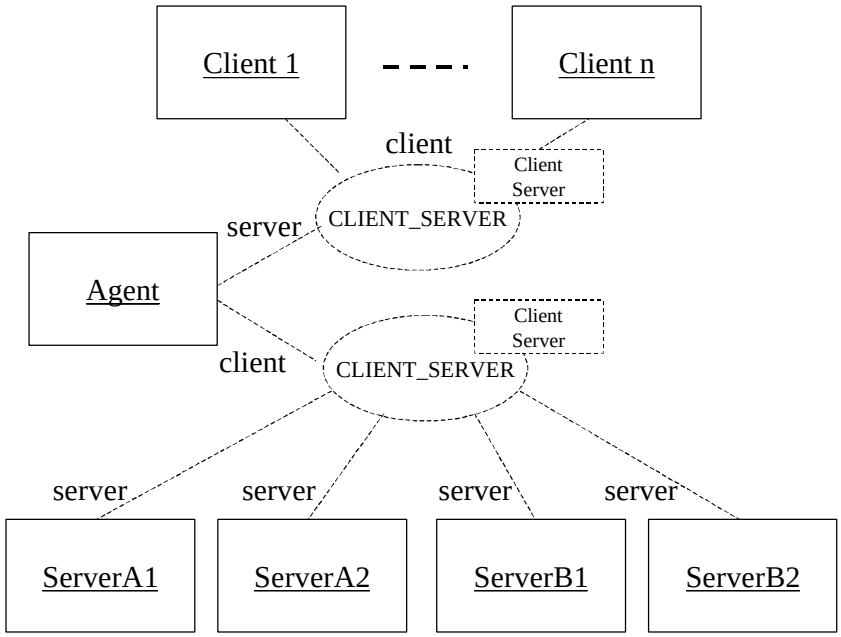


Figure 52: P_ORB Collaboration Diagram

Deployment Diagram:

The Deployment diagram of the P-ORB is similar to that of the F-ORB. The main differences are the links between processors. In the P-ORB case, the client needs not to be connected to the servers. Just like in the F-ORB case, the multiplicity of the P-Agent task is controlled by [Abd-97] case study and is equal to the chosen degree of cloning.

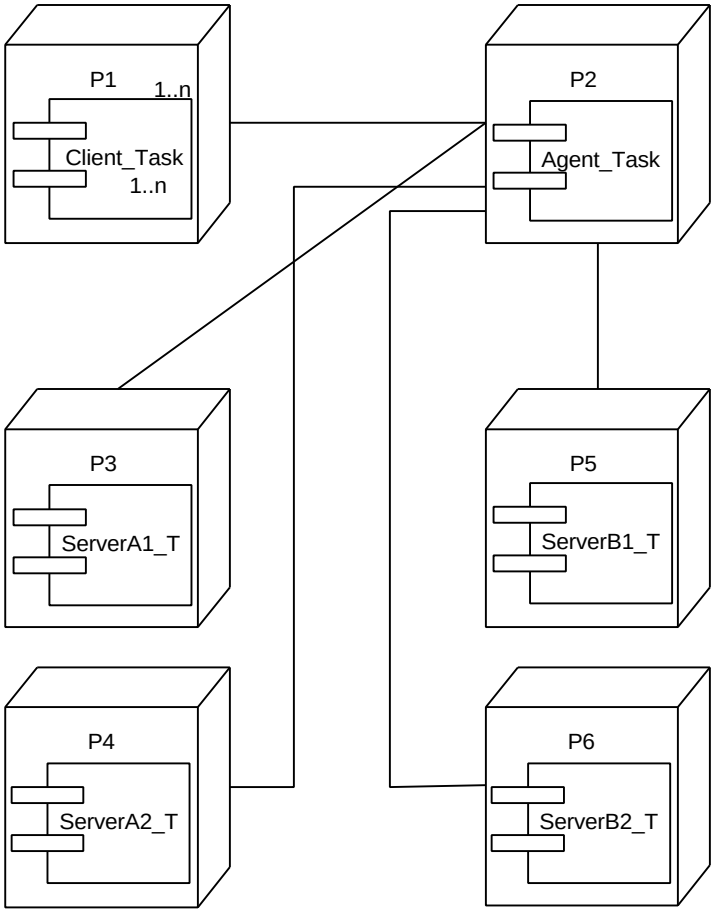


Figure 53:P-ORB Deployment Diagram

5.4.3 P-ORB in PROGRES

The selected workload factors of the P-ORB in this work are the same as the ones previously selected for the H-ORB, (summarized in Table 2: H-ORB Workload Factors Selected Values). It is comparable to the values of [Abd-97] case study experiment whose output is illustrated in Figure A-1 on page A-2 of appendix "A" of [Abd-97].

5.4.3.1 PROGRES Input

The three functions that determine the input for the PROGRES program are shown in the following three figures. The main differences between the P-ORB and the F-ORB in the "ConfigPlatform" are the processor links. The input to this function is extracted mainly from the P_ORB Deployment diagram (see Figure 53).

```
transaction ConfigurePlatform =
   begin
        NewProcessorNode ( "P1", 0, "f", "1.0" )
      & NewProcessorNode ( "P2", 1, "f", "1.0" )
      & NewProcessorNode ( "P3", 1, "f", "1.0" )
      & NewProcessorNode ( "P4", 1, "f", "1.0" )
      & NewProcessorNode ( "P5", 1, "f", "1.0" )
      & NewProcessorNode ( "P6", 1, "f", "1.0" )
      & NewTaskComponent ( "T1", "P1", 1 )
      & NewTaskComponent ( "T2", "P2", 1 )
      & NewTaskComponent ( "T3", "P3", 1 )
      & NewTaskComponent ( "T4", "P4", 1 )
      & NewTaskComponent ( "T5", "P5", 1 )
      & NewTaskComponent ( "T6", "P6", 1 )
      & LinkProcessorNodes ( "P1", "P2", "10000.0" )
      & LinkProcessorNodes ( "P2", "P3", "10000.0" )
      & LinkProcessorNodes ( "P2", "P4", "10000.0" )
      & LinkProcessorNodes ( "P2", "P5", "10000.0" )
      & LinkProcessorNodes ( "P2", "P6", "10000.0" )
   end
end;
```

Figure 54: P_ORB ConfigurePlatform Function

In the P-ORB architecture, a Client-Server relationship is established between the Client and the Agent, and between the Agent and each of the four servers. The input to this function is extracted mainly from the P_ORB Collaboration diagram (see Figure 52).

```
transaction EstablishCollaborationInfo =
      begin
           AddClientServerRelationship ( "Client", "PORB" )
         & AddClientServerRelationship ( "PORB", "SERVER_A1" )
         & AddClientServerRelationship ( "PORB", "SERVER_A2" )
         & AddClientServerRelationship ( "PORB", "SERVER_B1" )
         & AddClientServerRelationship ( "PORB", "SERVER_B2" )
      end
```

Figure 55: P-ORB EstablishCollaborationInfo Function

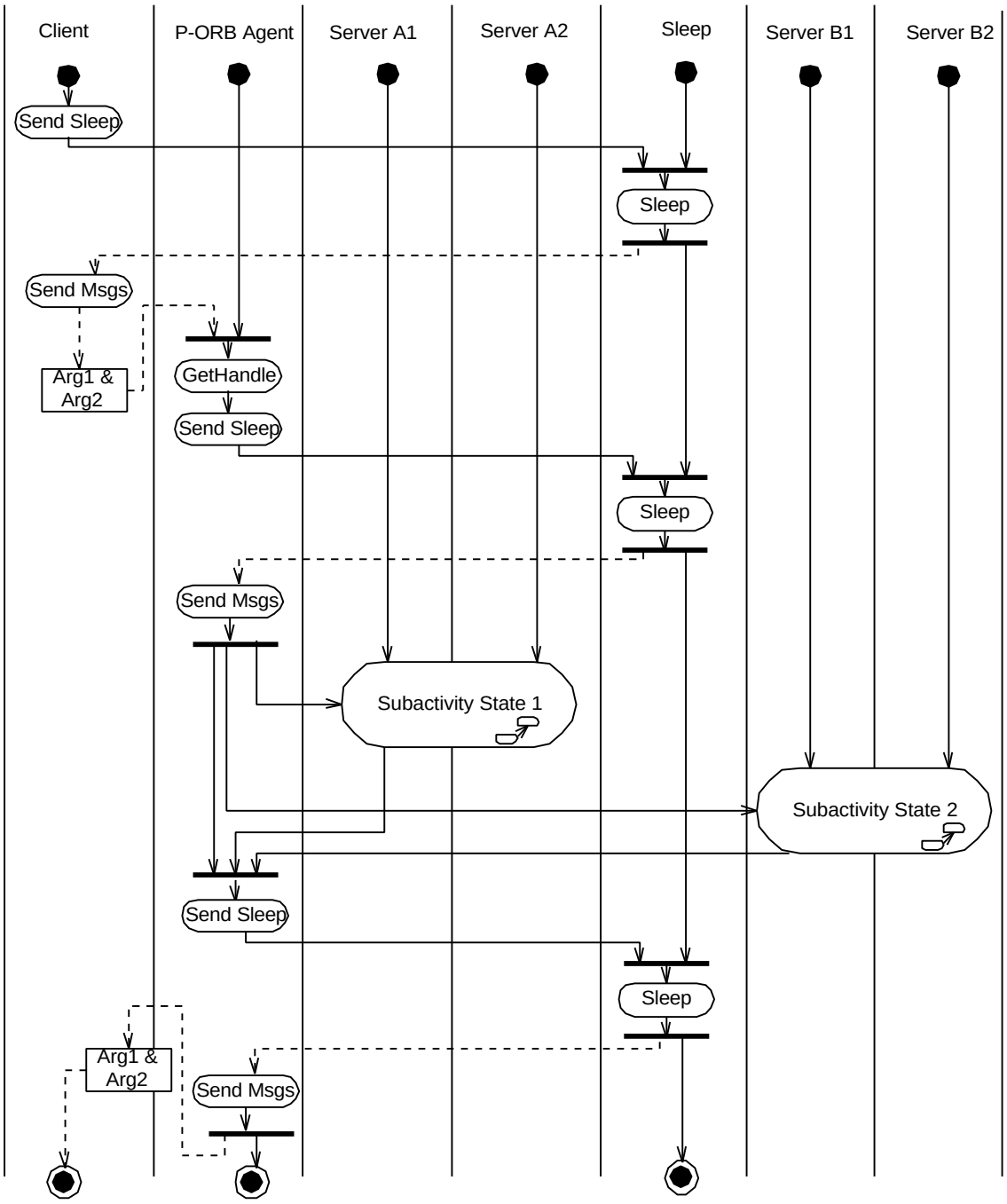
The sequence diagram is described step by step in the “CreateProblem” function. The input to this function is extracted mainly from the P_ORB Sequence diagram (see Figure 51). All API in all functions are used as described previously in Sections 3.2.2.1 and 4.2.2.1.

```
transaction CreateProblem =
    use C_Inst, P_Inst, SA1_Inst, SA2_Inst, SB1_Inst, SB2_Inst : INSTANCE;
       MainAS, AS1, AS2: COMPOSITE_ACTION
    do
        CreateMainActionSequence ( "MainActionSequence", out MainAS )
      & CreateNewInstance ( "Client", "T1", true, out C_Inst )
      & CreateNewInstance ( "PORB", "T2", true, out P_Inst )
      & CreateNewInstance ( "SERVER_A1", "T3", true, out SA1_Inst )
      & CreateNewInstance ( "SERVER_A2", "T4", true, out SA2_Inst )
      & CreateNewInstance ( "SERVER_B1", "T5", true, out SB1_Inst )
      & CreateNewInstance ( "SERVER_B2", "T6", true, out SB2_Inst )
      & Sleep ( C_Inst, 1, MainAS, 200 )
      & CreateSyncCall ( C_Inst, P_Inst, "Msg12", "ARG1", 4800, "", 3, MainAS, 0 )
      & CreateLocalAction ( P_Inst, "FindAddress", "", 4, MainAS, 8 )
      & Sleep ( P_Inst, 5, MainAS, 200 )
      & CreateActionSequence ( P_Inst, "AS1", "", 7, MainAS, 0, out AS1 )
      & CreateActionSequence ( P_Inst, "AS2", "", 7, MainAS, 0, out AS2 )
      & CreateAsyncCall ( P_Inst, SA1_Inst, "Msg1", "ARG1", 4800, "", 1, AS1, 0 )
      & CreateLocalAction ( SA1_Inst, "DoWork", "", 2, AS1, 10 )
      & Sleep ( SA1_Inst, 3, AS1, 200 )
      & CreateAsyncCall ( SA1_Inst, P_Inst, "Msg1_Rep", "ARG1", 4800, "", 5, AS1, 0 )
      & CreateAsyncCall ( P_Inst, SA2_Inst, "Msg1", "ARG1", 4800, "", 1, AS1, 0 )
      & CreateLocalAction ( SA2_Inst, "DoWork", "", 6, AS1, 10 )
      & Sleep ( SA2_Inst, 7, AS1, 200 )
      & CreateAsyncCall ( SA2_Inst, P_Inst, "Msg1_Rep", "ARG1", 4800, "", 9, AS1, 0 )
      & CreateAsyncCall ( P_Inst, SB1_Inst, "Msg2", "ARG2", 4800, "", 1, AS2, 0 )
      & CreateLocalAction ( SB1_Inst, "DoWork", "", 2, AS2, 15 )
      & Sleep ( SB1_Inst, 3, AS2, 200 )
      & CreateAsyncCall ( SB1_Inst, P_Inst, "Msg2_Rep", "ARG2", 4800, "", 5, AS2, 0 )
      & CreateAsyncCall ( P_Inst, SB2_Inst, "Msg2", "ARG2", 4800, "", 1, AS2, 0 )
      & CreateLocalAction ( SB2_Inst, "DoWork", "", 6, AS2, 15 )
      & Sleep ( SB2_Inst, 7, AS2, 200 )
      & CreateAsyncCall ( SB2_Inst, P_Inst, "Msg2_Rep", "ARG2", 4800, "", 9, AS2, 0 )
      & Sleep ( P_Inst, 8, MainAS, 200 )
      & CreateReplyCall ( P_Inst, C_Inst, "Msg12_Rep", "ARG2", 4800, "", 10, MainAS, 0 )
      & TerminateInstance ( C_Inst, 21, MainAS, 0 )
      & TerminateInstance ( P_Inst, 22, MainAS, 0 )
      & TerminateInstance ( SA1_Inst, 23, MainAS, 0 )
      & TerminateInstance ( SA2_Inst, 24, MainAS, 0 )
      & TerminateInstance ( SB1_Inst, 25, MainAS, 0 )
      & TerminateInstance ( SB2_Inst, 26, MainAS, 0 )
    end
  end;
```

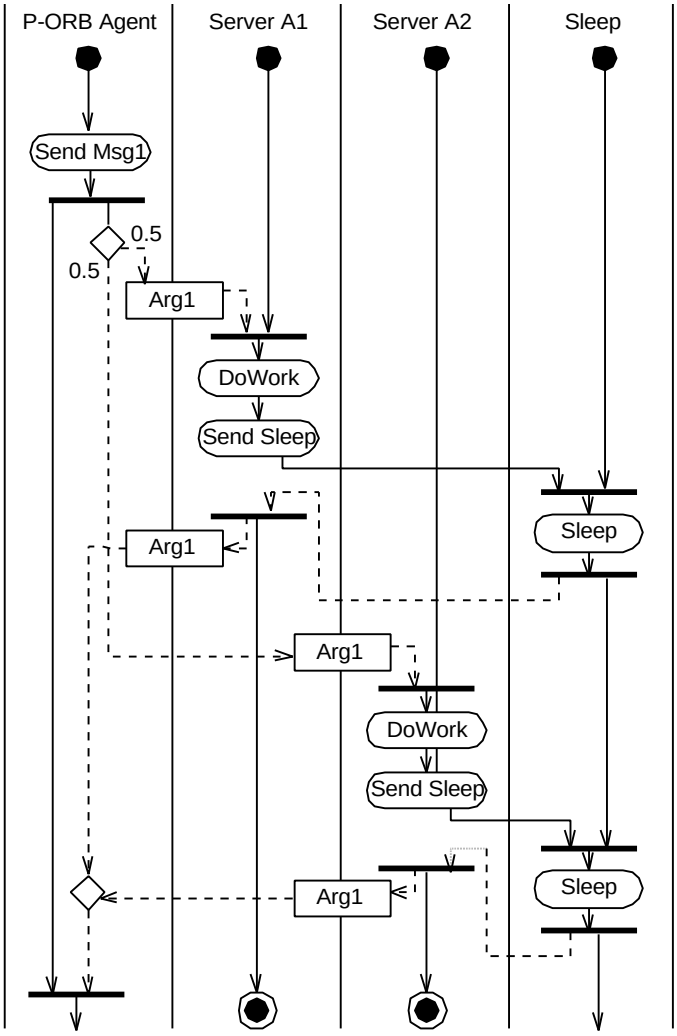
Figure 56: P-ORB CreateProblem Function

5.4.3.2 PROGRES Output

The first phase of transformation generates a description of an Activity diagram in text format. The graphical representation of this Activity diagram is given in the following figure.



(a) Main Activity Diagram



(b) Subactivity State 1

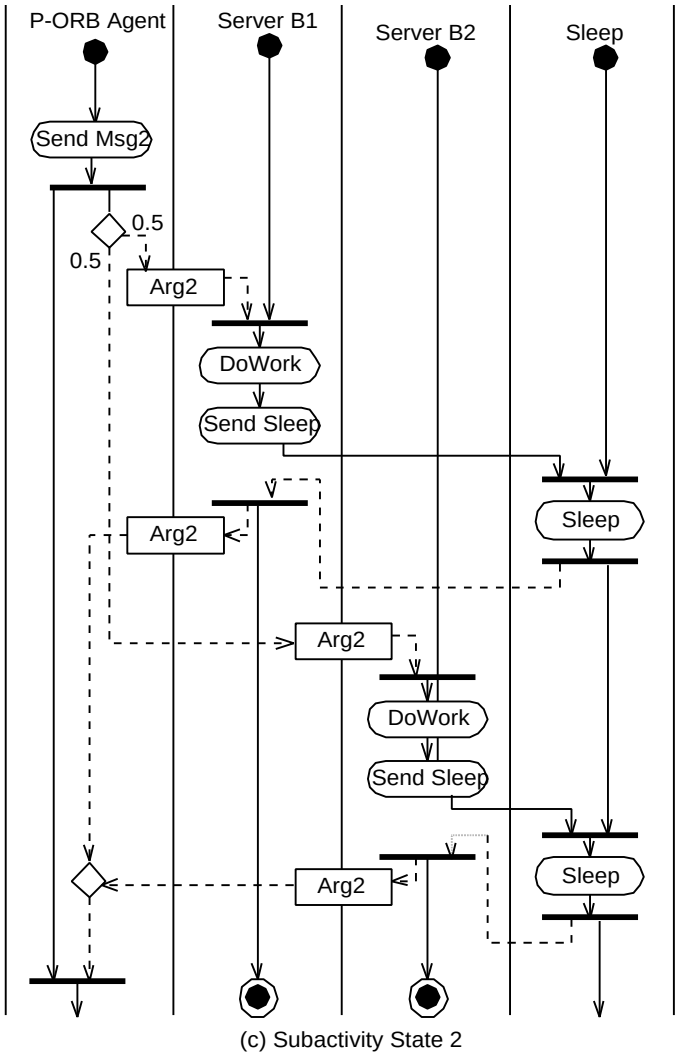


Figure 57: P-ORB Activity Graph

The PROGRES program was run, and it produced as output the LQN Model illustrated in the following figure. The textual description of this Model is illustrated in Figure 66, Appendix A.

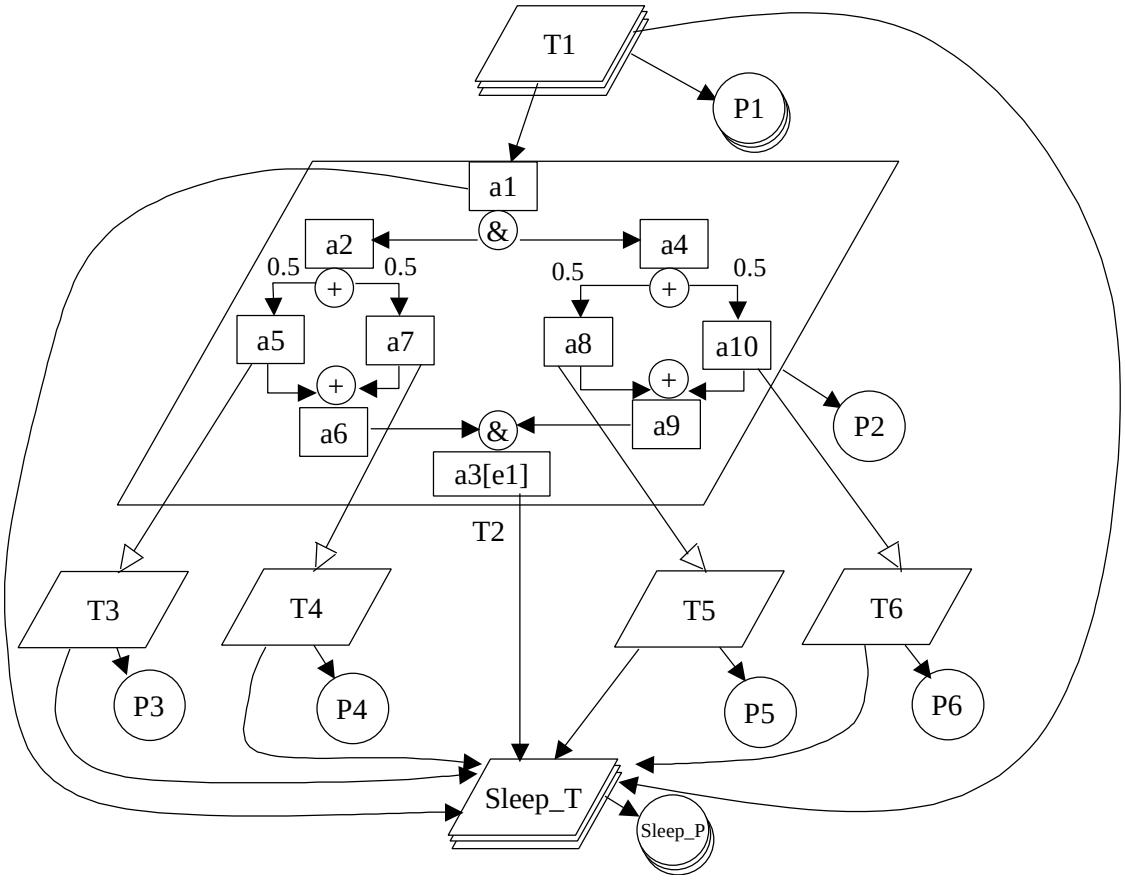


Figure 58: P-ORB LQN Performance Model

The “Activities” feature of the LQN models enabled us to model parallelism in the P-ORB case. Forking using an AND-Fork model activities that are done in parallel, while forking using an OR-Fork model alternate paths of execution. The replying activity in the P-ORN model is “a3”.

The calculated service times for each entry /activity is summarized in the following table:

Entry	***Service Time***
T1_e1	203.84
T2_e1_a1	0
T2_e1_a2	200
T2_e1_a3	200

T2_e1_a4	200
T3_e1	213.84
T4_e1	218.84

Table 7: P-ORB Calculated Service Time per Entry/Activity

The model was then used as input to the LQN solver (LQNS), giving the following results for the Mean Client Response Time (in seconds) for the different values of the number of clients (N).

Number of Clients	*Model Results*	*Measured Values*	*Error %*
1	0.73509	0.9015	18.45923461
2	1.09699	1.2	8.584166667
4	2.08761	2.1	0.59
8	4.20283	4.2	-0.067380952
16	8.45052	8.8	3.971363636
24	12.6999	13	2.308461538

Table 8: P-ORB Model Results VS Measured Values

A comparison between the results obtained and the measured values is depicted in the following graph.

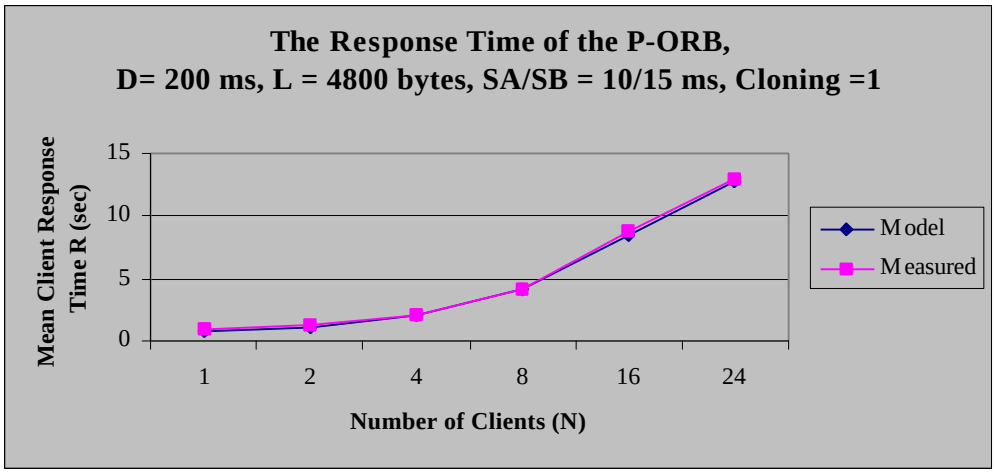


Figure 59: P-ORB Model Results VS Measured Values Graph

5.5 Varying the H-ORB

The question of "What is the effect on the Mean Client Total Response Time if a certain workload value changes" is a question that a model was built to answer. Here we ask two questions of this kind regarding the H-ORB case: What happens to the Mean Client Total Response Time if the Inter-node-delay changes?, and what happens if the message size changes?

5.5.1 Varying The Inter-Node Delay (H-ORB V1)

Starting with the H-ORB architecture, the Inter-Node Delay value was changed and fed as input to the PROGRES program. Since we need to compare to a measured value, we had to change the message length as well to have a comparable experiment in [Abd-97] case study.

The following table summarizes the workload values chosen as input to the PROGRES program in the case of H-ORB V1. It is comparable to the values of [Abd-97] case study experiment whose output is illustrated in Figure C-4 on page C-4 of appendix "C" of [Abd-97].

Factors	***Levels***
N	1,2,4,8,16,24
D (msec)	500
L (bytes)	150
SA / SB (msec)	10/15
Degree Of Cloning	1
Agent service time (msec)	4

Table 9: H-ORB V1 Workload Factors Selected Values

5.5.1.1 PROGRES Input

The previous changes translate into the following “CreateProblem” function. The sleep time, which resembles the inter-node delay, has changed from 200 to 500 (ms) compared to the H-ORB case (see Figure 38 for comparison). The rest of the function remains the same. Only changes are shown in the following figure; the rest is skipped.

```
transaction CreateProblem =
    use C_Inst, H_Inst, SA1_Inst, SA2_Inst, SB1_Inst, SB2_Inst : INSTANCE;
      MainAS, AS1, AS2, AS3, AS4 : COMPOSITE_ACTION
    do
       CreateMainActionSequence ( "MainActionSequence", out MainAS )
                    |
      & Sleep ( C_Inst, 1, MainAS, 500 )
                    |
     & Sleep ( H_Inst, 5, MainAS, 500 )
                    |
      & Sleep ( C_Inst, 8, MainAS, 500 )
                    |
      & Sleep ( SA1_Inst, 3, AS1, 500 )
                    |
      & Sleep ( SA2_Inst, 3, AS2, 500 )
                    |
      & Sleep ( C_Inst, 11, MainAS, 500 )
                    |
      & Sleep ( H_Inst, 15, MainAS, 500 )
                    |
      & Sleep ( C_Inst, 18, MainAS, 500 )
                    |
      & Sleep ( SB1_Inst, 3, AS3, 500 )
                    |
      & Sleep ( SB2_Inst, 3, AS3, 500 )
                    |
      & TerminateInstance ( C_Inst, 21, MainAS, 0 )
                    |
    end
  end;
```

Figure 60: H-ORB V1 CreateProblem Function

5.5.1.2 PROGRES Output

When the PROGRES program was run, it produced as output an LQN model that is similar to the H-ORB model (see Figure 40). The only change is in the service time of each entry in the Entry section. The calculated service times for each entry is summarized in the following table:

Entry	***Service Time***
T1_e1	2000.24
T2_e1	504
T2_e2	504
T3_e1	510.12
T4_e1	515.12

Table 10: H-ORB V1 Calculated Service Time per Entry

The model was then used as input to the LQN solver (LQNS), giving the following results for the Mean Client Response Time (in seconds) for the different values of the number of clients (N).

Number of Clients	***Model Results***	***Measured Values***	***Error %***
1	4.03348	4.2	3.964761905
2	4.16717	4.2	0.781666667
4	4.46066	4.2	-6.206190476
8	5.144	4.7	-9.446808511
16	6.79149	6.1	-11.33590164
24	8.63862	7.8	-10.75153846

Table 11: H-ORB V1 Model Results VS Measured Values

A comparison between the results obtained and the measured values is depicted in the following graph.

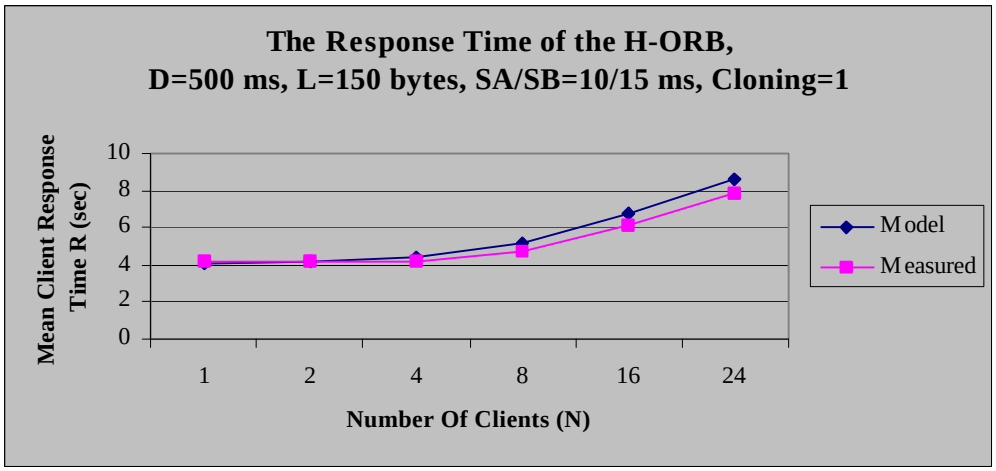


Figure 61: H-ORB V1 Model Results VS Measured Value Graph

5.5.2 Varying The Message Length (H-ORB V2)

Starting with the H-ORB architecture, the Message Length value was changed and fed as input to the PROGRES program.

The following table summarizes the workload values chosen as input to the PROGRES program in the case of H-ORB V2. It is comparable to the values of the case study experiment whose output is illustrated in Figure A-9 on page A-7 of appendix "A" of [Abd-97].

Factors	***Levels***
N	1,2,4,8,16,24
D (msec)	200
L (bytes)	19200
SA / SB (msec)	10/15
Degree Of Cloning	1
Agent service time (msec)	4

Table 12: H-ORB V2 Workload Factors Selected Values

5.5.2.1 PROGRES Input

The previous changes translate into the following “CreateProblem” function. The main difference between it and the H-ORB input is that the size of the transferred message has changed from 4800 to 19200 bytes (see Figure 38 for comparison). Only changes are shown here; the rest of the function is skipped.

```
transaction CreateProblem =
use C_Inst, H_Inst, SA1_Inst, SA2_Inst, SB1_Inst, SB2_Inst : INSTANCE;
      MainAS, AS1, AS2, AS3, AS4 : COMPOSITE_ACTION
       CreateMainActionSequence ( "MainActionSequence", out MainAS )
                    |
     & CreateSyncCall ( C_Inst, SA1_Inst, "Msg2", "ARG1", 19200, "", 1, AS1, 0 )
     & CreateSyncCall ( C_Inst, SA2_Inst, "Msg2", "ARG1", 19200, "", 1, AS2, 0 )
                    |
     & CreateReplyCall ( SA1_Inst, C_Inst, "Msg2_Rep", "ARG1", 19200, "", 5, AS1, 0 )
     & CreateReplyCall ( SA2_Inst, C_Inst, "Msg2_Rep", "ARG1", 19200, "", 5, AS2, 0 )
                    |
     & CreateSyncCall ( C_Inst, SB1_Inst, "Msg4", "ARG2", 19200, "", 1, AS3, 0 )
     & CreateSyncCall ( C_Inst, SB2_Inst, "Msg4", "ARG2", 19200, "", 1, AS4, 0 )
                    |
    & CreateReplyCall ( SB1_Inst, C_Inst, "Msg4_Rep", "ARG2", 19200, "", 5, AS3, 0 )
    & CreateReplyCall ( SB2_Inst, C_Inst, "Msg4_Rep", "ARG2", 19200, "", 5, AS4, 0 )
                    |
     & TerminateInstance ( C_Inst, 21, MainAS, 0 )
   end
 end;
```

Figure 62: H-ORB V2 CreateProblem Function

5.5.2.2 PROGRES Output

When the PROGRES program was run, it produced as output an LQN model that is similar to the H-ORB model (see Figure 40). The only change is in the service time of entries in the Entry section. The calculated service times for each entry is summarized in the following table:

Entry	***Service Time***
T1_e1	830.72

T2_e1	204
T2_e2	204
T3_e1	225.36
T4_e1	230.36

Table 13: H-ORB V2 Calculated Service Time per Entry

The model was then used as input to the LQN solver (LQNS), giving the following results for the Mean Client Response Time (in seconds) for the different values of the number of clients (N).

Number of Clients	***Model Results***	***Measured Values***	***Error %***
1	1.69444	1.7933	5.512741873
2	1.75132	1.8	2.704444444
4	1.877	1.8	-4.277777778
8	2.17348	2	-8.674
16	2.90084	2.7	-7.438518519
24	3.72285	3.5	-6.367142857

Table 14: H-ORB V2 Model Results VS Measured Values

A comparison between the results obtained and the measured values is depicted in the following graph.

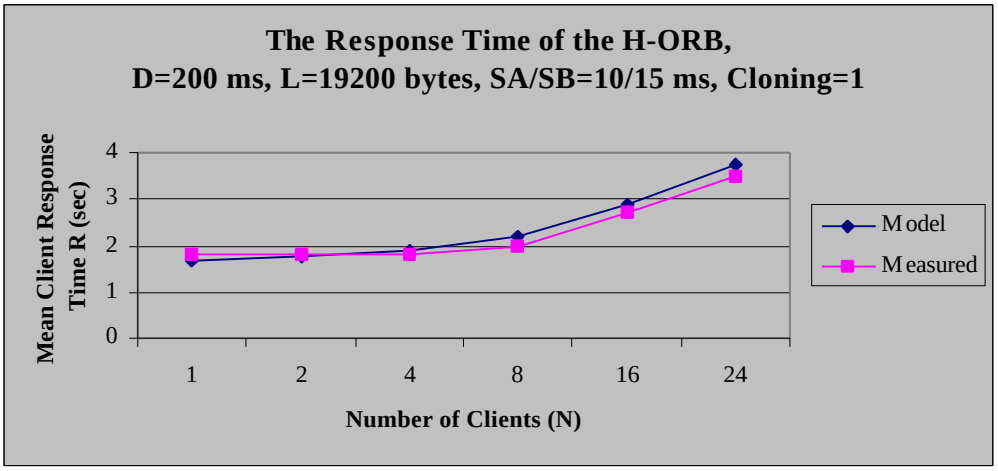


Figure 63: H-ORB V2 Model Results VS Measured Values

General Remarks

- The results of the comparisons between the measurements done in [Abd-97] case study and between the results obtained by the generated models show that the generated models are very reasonable, if not very good representations of the real systems. The results show an error percentage of less than (+/-13.5), except for a couple of cases (F-ORB, N=1 and P-ORB, N=1).

- The "Activities" feature of the LQN models enabled us to model parallelism in the P-ORB case. Forking using AND-Forks models activities that are done in parallel, while forking using OR-Forks models alternate paths of execution.

- In all cases, we needed to add annotation to UML specifications to specify performance related parameters since UML does not provide all the necessary information for a complete definition of the performance model.

6 CONCLUSION

The objective of this work is to develop a systematic methodology to automatically translate software specifications written in the Unified Modeling Language (UML) into Layered Queueing networks (LQN) performance models using Graph Grammar techniques. This objective was met by developing a PROGRES program that transforms specifications of software systems written using UML diagrams into LQN models.

The transformation processes was done in two phases. The input to the first phase is a graph that represents the software system's UML Sequence diagram plus some additional information on software architecture. The output is a graph that represents the system's Activity diagram, which serves as the input to the next phase. The output of the second phase is a graph that represents the equivalent LQN model of the system that is written to a text file. For each phase, a new algorithm was developed that traverses the input graph and produces its equivalent output graph.

Collaboration diagrams were used to add information about the relationships between the active objects in the system. Deployment diagrams were used to add the physical properties of the system. In all cases, annotations were added to the UML specifications to specify performance related parameters since UML does not provide all the necessary information for a complete definition of the performance model.

A case study is developed to validate and verify the transformation process, where the measurements of three different software architectures of a distributed software systems are compared to the results of their equivalent generated LQN models. The comparison shows very reasonable, if not very good, representation of the real systems. In most cases, the error% was less than 13.5%, which still gives a good approximation of reality. However, the main purpose of the thesis was not to conduct a thorough performance analysis of these systems, but to show that the LQN model automatically obtained from UML specifications give acceptable results.

This suggests that comparing between alternate architectures can be done easily if the equivalent models are automatically generated. This saves the amount of effort and time spent in developing the models by hand and gives fast feedback to the designers. The PROGRES transformation represents a proof of concept for the proposed method of deriving LQN models from UML specifications.

Future work is necessary to develop a better input and output techniques for the PROGRES program. For example, the PROGRES input graphs could be obtained from an XML file that describes the UML model. Similarly, the intermediate output of the PROGRES program, which represents the equivalent Activity diagram could be written in XML format for further usage in a UML tool.

Other UML diagrams may contribute to the input of the process as well. For example, some designers prefer using Collaboration diagrams to specify interactions rather than using sequence diagrams.

The issue of performance annotations needs further study. After the Performance Profile proposal is refined and adopted by OMG, the graph transformation must be made consistent with the profile.

APPENDIX “A”

H-ORB resulting LQN model for number of clients = 1.

```
"Please Add Comment"
0.00001
100
1
0.9
-1

P  7
p P2 f i
p P3 f m 1
p Sleep_P f i
p P4 f m 1
p P5 f m 1
p P6 f m 1
p P1 f i
-1

T  7
t T2 n T2_e1 T2_e2 -1 P2 i
t T6 n T6_e1 -1 P6 m 1
t Sleep_T n Sleep_T_e1 -1 Sleep_P i
t T3 n T3_e1 -1 P3 m 1
t T1 r T1_e1 -1 P1 m 1
t T5 n T5_e1 -1 P5 m 1
t T4 n T4_e1 -1 P4 m 1
-1
```

```
E  8
s T5_e1 18.84 0 0 -1
f T5_e1 0 0 0 -1
s T2_e1 4 0 0 -1
f T2_e1 0 0 0 -1
s T2_e2 4 0 0 -1
f T2_e2 0 0 0 -1
s Sleep_T_e1 200 0 0 -1
f Sleep_T_e1 1 0 0 -1
s T6_e1 18.84 0 0 -1
f T6_e1 0 0 0 -1
s T1_e1 7.68 0 0 -1
f T1_e1 0 0 0 -1
s T4_e1 13.84 0 0 -1
f T4_e1 0 0 0 -1
s T3_e1 13.84 0 0 -1
f T3_e1 0 0 0 -1
y T1_e1 Sleep_T_e1 4 0 0 -1
y T2_e1 Sleep_T_e1 1 0 0 -1
y T2_e2 Sleep_T_e1 1 0 0 -1
y T3_e1 Sleep_T_e1 1 0 0 -1
y T4_e1 Sleep_T_e1 1 0 0 -1
y T5_e1 Sleep_T_e1 1 0 0 -1
y T6_e1 Sleep_T_e1 1 0 0 -1
y T1_e1 T3_e1 0.5 0 0 -1
y T1_e1 T4_e1 0.5 0 0 -1
y T1_e1 T2_e1 1 0 0 -1
y T1_e1 T5_e1 0.5 0 0 -1
y T1_e1 T6_e1 0.5 0 0 -1
y T1_e1 T2_e2 1 0 0 -1
-1
```

Figure 64: H-ORB LQN Model File

F-ORB resulting LQN model for number of clients = 1.

```
"Please Add Comment"
0.00001
100
1
0.9
-1

P  7
p P1 f i
p P4 f m 1
p Sleep_P f i
p P5 f m 1
p P6 f m 1
p P2 f m 1
p P3 f m 1
-1

T  7
t T1 r T1_e1 -1 P1 m 1
t T2 n T2_e2 T2_e1 -1 P2 m 1
t Sleep_T n Sleep_T_e1 -1 Sleep_P i
t T5 n T5_e1 -1 P5 m 1
t T6 n T6_e1 -1 P6 m 1
t T4 n T4_e1 -1 P4 m 1
t T3 n T3_e1 -1 P3 m 1
-1
```

```
E  8
s T1_e1 7.68 0 0 -1
f T1_e1 0 0 0 -1
s T3_e1 13.84 0 0 -1
f T3_e1 0 0 0 -1
s Sleep_T_e1 200 0 0 -1
f Sleep_T_e1 1 0 0 -1
s T5_e1 18.84 0 0 -1
f T5_e1 0 0 0 -1
s T4_e1 13.84 0 0 -1
f T4_e1 0 0 0 -1
s T2_e2 7.84 0 0 -1
f T2_e2 0 0 0 -1
s T6_e1 18.84 0 0 -1
f T6_e1 0 0 0 -1
s T2_e1 7.84 0 0 -1
f T2_e1 0 0 0 -1
y T1_e1 Sleep_T_e1 2 0 0 -1
y T2_e1 Sleep_T_e1 1 0 0 -1
y T2_e2 Sleep_T_e1 1 0 0 -1
y T3_e1 Sleep_T_e1 1 0 0 -1
y T4_e1 Sleep_T_e1 1 0 0 -1
y T5_e1 Sleep_T_e1 1 0 0 -1
y T6_e1 Sleep_T_e1 1 0 0 -1
y T1_e1 T2_e1 1 0 0 -1
y T1_e1 T2_e2 1 0 0 -1
z T2_e1 T3_e1 0.5 0 0 -1
z T2_e1 T4_e1 0.5 0 0 -1
z T2_e2 T5_e1 0.5 0 0 -1
z T2_e2 T6_e1 0.5 0 0 -1
-1
```

Figure 65: F-ORB LQN Model File

P-ORB resulting LQN model for the number of clients = 1.

```
G
"Please Add Comment"
0.00001
100
1
0.9
-1

P  7
p P1 f i
p P3 f m 1
p Sleep_P f i
p P5 f m 1
p P6 f m 1
p P2 f m 1
p P4 f m 1
-1

T  7
t T2 n T2_e1 -1 P2 m 1
t T3 n T3_e1 -1 P3 m 1
t Sleep_T n Sleep_T_e1 -1 Sleep_P i
t T5 n T5_e1 -1 P5 m 1
t T6 n T6_e1 -1 P6 m 1
t T1 r T1_e1 -1 P1 m 1
t T4 n T4_e1 -1 P4 m 1
-1
```

```
E  7
s T1_e1 3.84 0 0 -1
f T1_e1 0 0 0 -1
s T5_e1 18.84 0 0 -1
f T5_e1 0 0 0 -1
s Sleep_T_e1 200 0 0 -1
f Sleep_T_e1 1 0 0 -1
s T6_e1 18.84 0 0 -1
f T6_e1 0 0 0 -1
s T3_e1 13.84 0 0 -1
f T3_e1 0 0 0 -1
s T4_e1 13.84 0 0 -1
f T4_e1 0 0 0 -1
A T2_e1 T2_e1_a1
y T1_e1 T2_e1 1 0 0 -1
y T1_e1 Sleep_T_e1 1 0 0 -1
y T3_e1 Sleep_T_e1 1 0 0 -1
y T4_e1 Sleep_T_e1 1 0 0 -1
y T5_e1 Sleep_T_e1 1 0 0 -1
y T6_e1 Sleep_T_e1 1 0 0 -1
-1

A T2
s T2_e1_a4 0
s T2_e1_a3 0
s T2_e1_a2 0
s T2_e1_a1 0
s T2_e1_a5 0
s T2_e1_a6 0
s T2_e1_a7 0
s T2_e1_a8 0
s T2_e1_a9 0
s T2_e1_a10 0
z T2_e1_a5 T3_e1 1
z T2_e1_a6 T4_e1 1
z T2_e1_a8 T5_e1 1
z T2_e1_a9 T6_e1 1
y T2_e1_a2 Sleep_T_e1 1
y T2_e1_a4 Sleep_T_e1 1
y T2_e1_a3 Sleep_T_e1 1
:
T2_e1_a1 -> T2_e1_a4  & T2_e1_a2;
T2_e1_a2 -> (0.5) T2_e1_a5 + (0.5) T2_e1_a6;
T2_e1_a5 + T2_e1_a6 -> T2_e1_a7;
T2_e1_a4 -> (0.5) T2_e1_a8 + (0.5) T2_e1_a9;
T2_e1_a8 + T2_e1_a9 -> T2_e1_a10;
T2_e1_a10  & T2_e1_a7 -> T2_e1_a3;
T2_e1_a3[T2_e1]
-1
```

Figure 66: P-ORB LQN Model File

BIBLIOGRAPHY

[Abd-97]

Abdul-Fatah, Istabrak. "Performance of CORBA-Based Client-Server Architecture" M.Eng. thesis, Department of Systems and Computer Engineering, Carleton University, April 1997.

[Abd-98a]
Abdul-Fatah, Istabrak and S. Majumdar, "The Effect of Object-Agent Interaction on the Performance of CORBA Systems", Proc. IEEE International Conference on Performance, Computing, and Communication, Tempe, Arrizona, Feb.1998, pp.67-74.

[Abd-98b]
Abdul-Fatah, Istabrak and S. Majumdar, "Performance Comparison of Architectures for Client-Server Interactions in CORBA", Proc. IEEE International Conference on Performance, Computing, and Communication, Tempe, Arrizona, Feb.1998.

[Balsamo-01]
Simonetta Balsamo and Marta Simeoni, "On transforming UML models into performance models", *The Workshop on Transformations in the Unified Modeling Language*, Genova, Italy, April 7th 2001.

[Booch-99]
Booch, Grady, James Rumbaugh and Ivar Jacobson. *The Unified Modeling Language User Guide*, MA: Addison Wesley Longman Inc., 1999.

[Corte-00]
Vittorio Cortellessa and Raffaela Mirandola, "Deriving a Queueing Network based Performance Model from UML Diagrams", *Proceedings of the Second International Workshop on Software and Performance*, Ottawa, Canada, pp 58-70, Sept. 2000.

[Franks-95]
G. Franks, A. Hubbard, S. .Majumdar, D. Petriu, J. Rolia, C.M. Woodside, "A toolset for Performance Engineering and Software Design of Client-Server Systems", *Performance Evaluation*, Vol. 24, Nb. 1-2, pp 117-135, November 1995.

[Franks-99]
Grag Franks. Performance Analysis of Distributed Server Systems, PhD thesis, Department of Systems and Computer Engineering, Carleton University, 1999.

[Iona-97]

Iona Technologies, *Orbix Programmers' Guide*, Dublin, Irland, 1997.

[ITUT-99]

International Telecommunication Union (ITU-T Z.120), Message Sequence Chart (MSC), 11/99.

[Jacob-98]
Jacobson, Ivar, Grady Booch and James Rumbaugh. *The Unified Software Development Process*, MA: Addison Wesley Longman Inc., 1998.

[Menasce-94]
D.A. Menasce, V.A.F. Almeida, L.W. Dowdy, *Capacity Planning and Performance Modeling*, Prentice Hall, 1994

[MOD-00]
Modeling Language Guide, Rational Rose RealTime, version 2000.02.10, 2000.

[Neilson-95]
J.E. Neilson, C.M. Woodside, D.C. Petriu and S. Majumdar, "Software Bottlenecking in Client-Server Systems and Rendez-vous Networks", IEEE *Trans. On Software Engineering*, Vol. 21, No. 9, pp. 776-782, Sept. 1995.

[OMG-99]
OMG Unified Modeling Language Specification, www.rational.com/uml/index.jsp , version 1.3, June 1999.

[OMG-00]
OMG document ad/2000-08-04, a draft response to the RFP, http://www.sce.carleton.ca/faculty/woodside.html, Aug. 14, 2000.

[ORB-94a]
PostModern Computing Technologies Inc., *ORBeline User's Guide*, Mountain View, CA 94043, 1994.

[ORB-94b]
PostModern Computing Technologies Inc., *ORBeline ReferenceGuide*, Mountain View, CA 94043, 1994.

[Petriu-98]
Dorina Petriu, X. Wang, "Deriving Software Performance Models from Architectural Pattern by Graph Transformations", *Proceedings of the Sixth International Workshop on Theory and Application of Graph Transformations TAGT'98*, Paderborn, Germany, Nov. 1998.

[Petriu-99]
Dorina Petriu, X. Wang, "From UNL Descriptions of High-level Software Architecture to LQN Performance Models", *Proceedings of Applications of Graph Transformations with Industrial Relevance AGTIVE'99*, Monastery Rolduc, Kerkrade, The Netherlands, Sept. 1999.

[Petriu-00a]
Dorina Petriu, Hoda Amer, Shikharesh Majumdar, Istabrak Al-Fatah, "Using Analytic Models for Predicting Middleware Performance", *Proceedings of the Second International Workshop on Software and Performance*, Ottawa, Canada, pp 189-194, Sept. 2000.

[Petriu-00b]
Dorina Petriu, Christiane Shousha, Anant Jalnapurkar, "Architecture-Based Performance Analysis Applied to a Telecommunication System", *IEEE Transactions on Software Engineering*, Vol.26, Nb.11, pp 1049-1065, Nov. 2000.

[Petriu-00c]
D.C. Petriu, Y.Sun, "Consistent Behaviour Representation in Activity and Sequence Diagrams", in UML'2000 The Unified Modeling Language - Advancing the Standard, Lecture Notes in Computer Science 1939, pp.369-382, Springer Verlag, 2000.

[Quart-00]
Quatrani, Terry, *Visual Modeling With Rational Rose 2000 and UML*, MA: Addison Wesley Longman Inc., 2000.

[Ramesh-98]
S.Ramesh, H.G.Perros, "A Multi-Layer Client-Server Queueing Network Model with Synchronous and Asynchronous Messages", *Proceedings of the First International Workshop on Software and Performance*, Santa Fe, USA, pp.107-119, Oct. 1998.

[Rolia-95]
J.A. Rolia, K.C. Sevcik, "The Method of Layers", *IEEE Trans. on Software Engineering*, Vol. 21, Nb. 8, pp. 689-700, August 1995.

[Schürr-94]
Schürr A.: *PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems*, Technical Report AIB 94-11, RWTH Aachen, Germany,1994

[Schürr-97]
Andy Schürr, *"PROGRES for Beginners"*, Lehrstuhl für Informatik III, RWTH Aachen, Ahornstr. 55,D-52074 Aachen, Germany, 1997.

[Schürr-97b]
Schürr A.: *Developing Graphical (Software Engineering) Tools with PROGRES*, Formal Demonstration, in: Proc. 19th Int. Conf. on Software Engineering ICSE'97, Boston, Massachusetts, 18.-23. Mai 1997, Los Alamitos: IEEE Computer Society Press (1997), 618-619

[Schürr-99]
Andy Schürr, *"A Guided Tour Through The PROGRES Environment"*, Lehrstuhl für Informatik III, RWTH Aachen, Ahornstr. 55,D-52056 Aachen, Germany, 1999.

[Smith-90]
C.U. Smith, *Performance Engineering of Software Systems*, Addison Wesley, 1990.

[Smith-00]
C.U. Smith, "SPE for Web Applications: New Challenges?", keynote speech at The 2nd International Workshop on Software And Performance WOSP'2000, Ottawa, Canada Sept. 2000.

[Wang-99]
Xin Wang, Deriving Software Performance Models From Architectural Patterns By Graph Transformation, M.Eng. thesis, Department of Systems and Computer Engineering, Carleton University, 1999.

[Woodside-89]
C.M. Woodside, “Throughput calculation for basic stochastic rendezvous networks”, Performance Evaluation, Vol. 9, Nb. 2, pp. 143-160, 1989.

[Woodside-95]
C.M. Woodside, J.E. Neilson, D.C. Petriu, S. Majumdar, “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.

[Woodside-01]
C.M. Woodside, "Draft Chapter on a UML Profile for General Performance Analysis", http://www.sce.carleton.ca/faculty/woodside.html, Apr 3, 2001.