Software Performance Evaluation: Graph Grammar-based Transformation of UML Design Models into Performance Models

Hoda Amer and Dorina C. Petriu

Carleton University
Department of Systems and Computer Engineering
Ottawa, Canada, K1S 5B6

Contac t for Correspondence:

Dorina C. Petriu
Carleton University
Department of Systems and Computer Engineering
1125 Colonel By Drive
Ottawa, ON, Canada, K1S 5B6
E-mail: petriu@sce.carleton.ca

Running title: Software Performance Evaluation of UML Models

Abstract. The quality of many software intensive systems is determined to a large extent by their performance characteristics, such as response time and throughput. The developers of such systems should be able to assess and understand the performance effects of various design decisions starting at an early stage, when changes are easy and less expensive, and continuing throughout the software life cycle. This can be achieved by constructing and analyzing quantitative performance models that capture the interact ions between the main system components and point to the system's performance trouble spots. The paper proposes a graph grammar-based transformation from UML design models into Layered Queueing Network (LQN) performance models. The LQN model structure is generated from the high-level software architecture showing the architectural patterns used in the system, and from deployment diagrams indicating the allocation of software components to hardware devices. The LQN model parameters are obtained from detailed models of key performance scenarios, represented as UML interaction or activity diagrams annotated with performance information (according to the recently proposed UML performance profile). The proposed transformation from UML to LQN was implemented with PROGRES, a well-known visual language and environment for programming with graph rewriting systems. The proposed technique was applied to the performance analysis of three CORBA-based client server systems, and the performance model results are reasonably close to measurements obtained from the actual implementations.

1. Introduction

The quality of many software intensive systems, ranging from real-time embedded systems to telecommunication and web-based applications, is determined to a large extent by their performance characteristics, such as response time and throughput. The developers of such systems should be able to assess and understand the performance effects of various design decisions starting at an early stage, when changes are easy and less expensiv e, and continuing throughout the software life cycle.

Software Performance Engineering (SPE), introduced in (Smith, 1990), is a technique that proposes to use quantitative methods and performance models in order to assess the performance effects of different design and implementation alternatives during the development of a system. SPE promotes the idea that the integration of performance analysis into the software development process, from the earliest stages to the end, can insure that the system will meet its performance objectives. This would eliminate the need for "late-fixing" of performance problems, a frequent practical approach that postpones any performance concerns until the system is completely implemented. Late fixes tend to be very expensive and inefficient, and the product may never reach its original performance requirements.

The process of building a system's performance model before the system is completely implemented starts with identifying a small set of key performance scenarios representative of the way in which the system will be used (Smith, 1990; Smith and Williams, 2001). The performance analysts must understand first the system behaviour for each scenario by talking with the system developers and/or by using design specifications. The analyst, helped by the developers, will follow the execution path through the software for each scenario, from component to component, identifying the quantitative demands for resources made by each component (such as CPU execution time and I/O operations), as well as the various reasons for queueing delays (such as competition for hardware and software resources). The scenario descriptions thus obtained are mapped onto a performance model. By solving the model, the analyst will obtain performance results such as response times, throughput, utilization of different resources by different software components, etc. Trouble spots can be thus identified, and alternative solutions for eliminating them assessed in a similar way.

The paper aims to bridge the gap between design specifications and performance modelling by proposing an automatic graph grammar-based transformation from UML (OMG, 1999) design models annotated with performance information into Layered Queueing Network (LQN) performance models. This way, the consistency of the performance model with the design model can be easily assured.The LQN model structure is generated from the high-level software architecture showing the architectural patterns used in the system, and from deployment diagrams indicating the allocation of software components to hardware devices. The LQN model parameters are obtained from key performance scenarios represented as UML interaction or activity diagrams annotated with performance information. The annotations are made according to the recent proposal for a UML Profile for Schedulability, Performance and Time (OMG, 2001), whose goal is to enable the construction of models that can be used for making quantitative performance predictions. The proposed performance profile extends the UML metamodel with stereotypes, tagged values and constraints that make possible to attach quantitative performance annotations (such as resource demands, execution probabilities, arrival rates, etc.) to the UML model elements.

The UML to LQN transformation proposed in this paper was implemented with PROGRES, a well-known visual language and environment for programming with graph rewriting systems (Schürr, A., 1990; Schürr, A., 1994; Schürr, A., 1997). The paper extends previous work of the authors: (Petriu, Shousha and Jalnapurkar, 2000) showed how to build LQN performance models based on the architectural patterns used, but the transformation was not automated, whereas (Petriu and Wang, 2000) proposed a PROGRES-based transformation of architectural patterns into LQN. This paper completes the picture by showing how the LQN parameters can be obtained, as well, from UML models annotated with performance information according to the newly proposed UML profile.

The technique proposed in the paper is applied to three CORBA-based client-server architectures. The goal was to generate automatically the LQN models from UML specifications of the system and see if it gives acceptable results, by comparing them with measurement results from (Abdul-Fatah and Majumdar, 1998).

The performance modelling formalism used in the paper is the *Layered Queueing Network* (LQN) model, an extension of the well-known Queueing Network (QN) model. LQN was developed especially for modelling concurrent and/or distributed software systems (Woodside, 1988; Woodside et al, 1995, Franks at al, 1995). The LQN components represent either software processes or hardware devices. LQN determines the delays due to contention, synchronization and serialization at both software and hardware levels (see section 2.1 for a more detailed description).

Since the introduction of SPE, there has been a significant research effort to integrate performance analysis into the software development process throughout all lifecycle phases. One aspect of this research is the derivation of performance models from software design specifications. A survey of the techniques developed in the recent years for deriving performance models from UML models is given in (Balsamo and Simeoni, 2001). The SPE methodology is followed very closely in (Cortelesa and Mirandola, 2000). Information from UML use case, deployment, and sequence diagrams is used to generate SPE scenario descriptions in the form of flow diagrams similar to those used in (Smith, 1990; Smith and Williams, 2001). The flow diagrams are then mapped onto QN models. Another work presented in (Kähkipuro, 2001) introduces a UML-based notation and framework for describing performance models, and a set of special techniques for modeling component-based distributed systems. The idea of patterns is used in (Gomaa and Menasce, 2000) to investigate the design and performance modelling of interconnection patterns for client/server systems. Compared to the existing work in this direction, our paper is the only one that uses graph-grammar based transformations and high-level architectural patterns to generate software performance models.

The paper is organized as follows: background information on LQN model, architectural patterns, UML performance profile and PROGRES is given in section 2; the principle of UML to LQN transformation is discussed in section 3; the PROGRES-based transformation is presented in section 4; a case-study of three middleware-based client/server systems is presented in section 5; and the conclusions in section 6.

2. Background

2.1. Layered Queueing Network model

Queueing Network (QN) models are a widely used technique for predicting the performance of computing systems. Although QN 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,88; Woodside et al 1995, Franks at al 1995) and the Method of Layers (Rolia and Sevcik, 1995) 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. 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 *tasks*) are software entities and hardware devices, and whose arcs denote service requests (see Fig. 1). The LQN tasks are classified into three categories: *pure clients* (also named

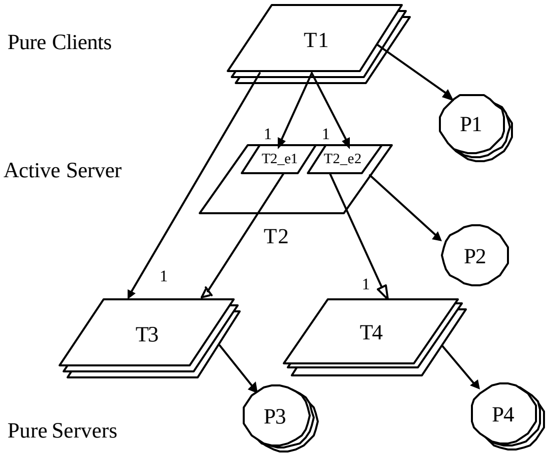


Figure 1. Simple LQN model

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 servers* 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. 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 Fig.2 for the notation).

An LQN task may offer more than one kind of service, each modeled by a so-called *entry* drawn as a smaller parallelogram nested in a task. 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.

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*. 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. In the case of an *asynchronous* message, the client does not block after sending the message and the server does not reply back. A *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.

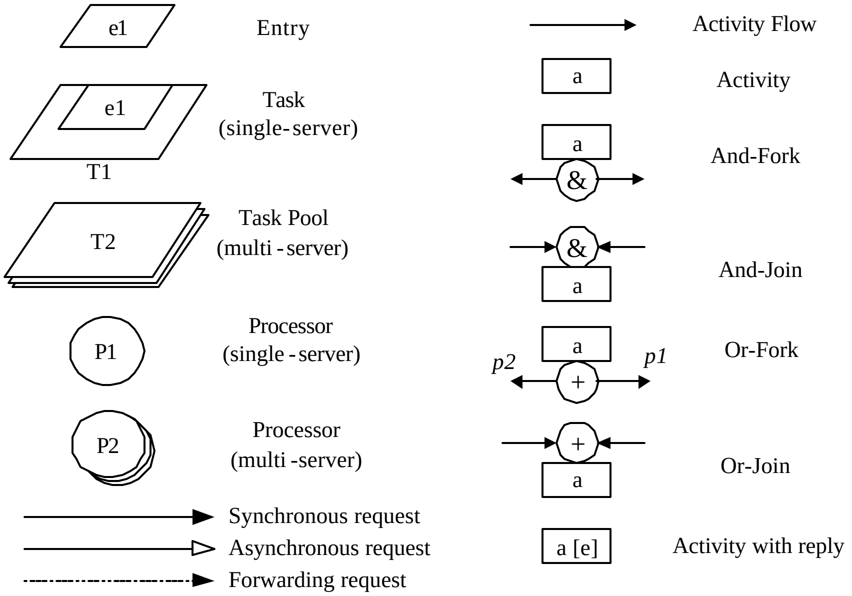


Figure 2. LQN graphical notation

A phase may be *deterministic* or *stochastic*, and is subject to the following assumptions:

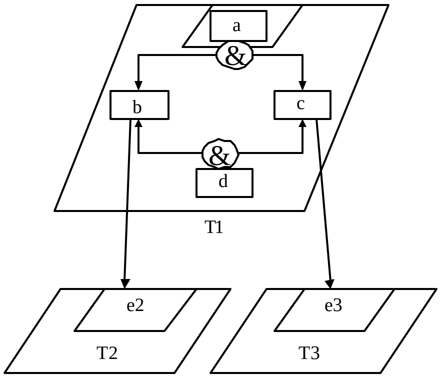


Figure 3. LQN task with activities

- The total CPU demand of a phase (whose mean value *s* is given as a parameter) is divided up into *n+1* exponentially distributed slices separated by requests to lower level servers. Each slice has the mean of *s/(n+1)*, where *n* is the number of requests made in that phase.
- The number of requests to lower level servers is geometrically distributed with a specified mean (given as a parameter) in a stochastic phase, and is deterministic in a deterministic phase.

A more recent extension to LQN (Franks, 2000) lets an entry be further decomposed into activities if more details are required to describe its execution. This is typically required when entries have fork and join interactions. Activities are components that represent the lowest level of detail in LQN. They can be connected together not only sequentially, but with fork and join interactions as well, in the form of a directed graph (see Fig. 2 and 3). 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*. Joins happens when multiple threads of control are connected together. As expected, the AND-joins introduce synchronization delays. An activity may have service time demand on the processor on which its task runs, just like a phase. Also, activities can make requests to other tasks by way of *synchronous* or *asynchronous* messages.

The parameters of an LQN model are as follows:

- customer (client) classes and their associated populations or arrival rates;
- for each phase (activity) of a software task entry: average execution time;
- for each phase (activity) making a request to a device: average service time at the device, and average number of visits;
- for each phase (activity) making a request to another task entry: average number of visits

- for each request arc: average communication delay;
- for each software and hardware server: scheduling discipline, multiplicity.

Although the LQN toolset presented in (Franks, 2000) includes both simulation and analytical solvers, the analytical solver was used to solve the LQN models generated in the paper.

2.2 UML Performance profile

According to (OMG,2001) the UML Performance Profile provides facilities for:

- capturing performance requirements within the design context
- associating performance-related Q0S characteristics with selected elements of the UML model
- specifying execution parameters which can be used by modelling tools to compute predicted performance characteristics
- presenting performance results computed by modelling tools or found by measurement.

The Profile describes a domain model, shown in Fig. 4, which identifies basic abstractions used in performance analysis. *Scenarios* define response paths through the system, and can have QoS requirements such as response times or throughputs. Each scenario is executed by a job class, called here a *workload*, which can be closed or open and has the usual characteristics (number of clients or arrival rate, etc.) Scenarios are composed of *scenario steps* that can be joined in sequence, loops, branches, fork/joins, etc. A scenario step may be an elementary operation at the lowest level of granularity, or may be a complex sub-scenario composed of many basic steps. Each step has a mean number of repetitions, a host execution demand, other demand to resources and its own QoS characteristics. Resources are another basic abstraction, and can be active or passive, each with their own attributes. The Performance profiles maps the classes from Fig. 4 to stereotypes that can be applied to a number of UML model elements, and each class attribute to a tagged value. For example, the basic abstraction PStep is mapped to the stereotype <<PAstep>> that can be applied to the following UML model elements: Message, Stimulus (when the scenario is represented by an interaction diagram) and ActionState, SubactivityState (when the scenario is represented by an activity diagram).

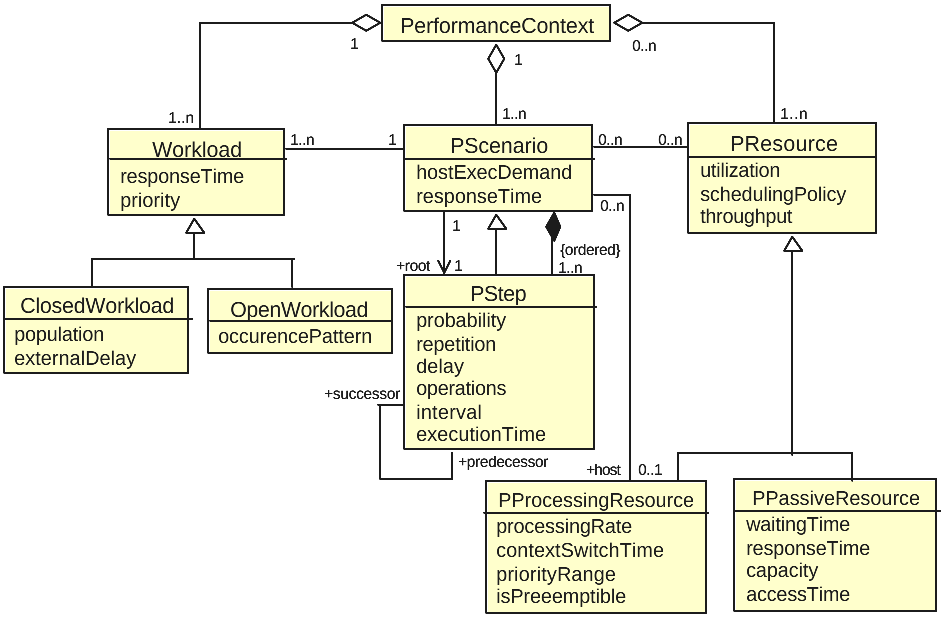
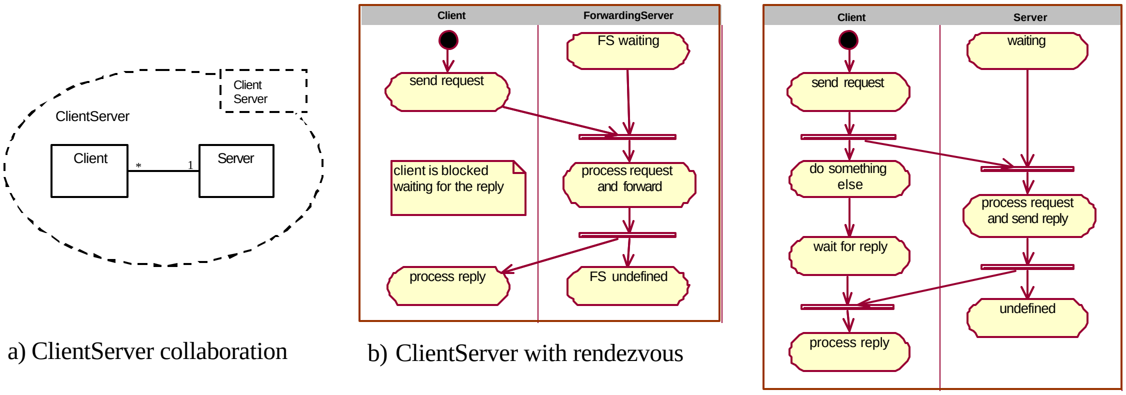


Figure 4. Domain model in the UML Performance Profile

2.3. Architectural Patterns

Frequently used architectural solutions are identified in literature as *architectural patterns* (such as pipeline and filters, client/server, client/broker/server, layers, master-slave, blackboard, etc.) (Shaw, 1996; Buchmann et al, 1996). A pattern introduces a higher-level of abstraction design artifact by describing a specific type of collaboration between a set of prototypical components playing well-defined roles, and helps our understanding of complex systems. Each architectural pattern describes two inter-related aspects: its *structure* (what are the components) and *behaviour* (how they interact). In the case of high-level architectural patterns, the components are usually concurrent entities that are executed in different threads of control, compete for resources, and may require some synchronization. The patterns are represented as UML *collaborations* (not to be confused with UML *collaboration diagrams*, a type of interaction diagrams). The symbol for a collaboration is an ellipse with dashed lines that may have an "embedded" square showing the roles played by different pattern participants (Booch et al, 1999; OMG, 1999).

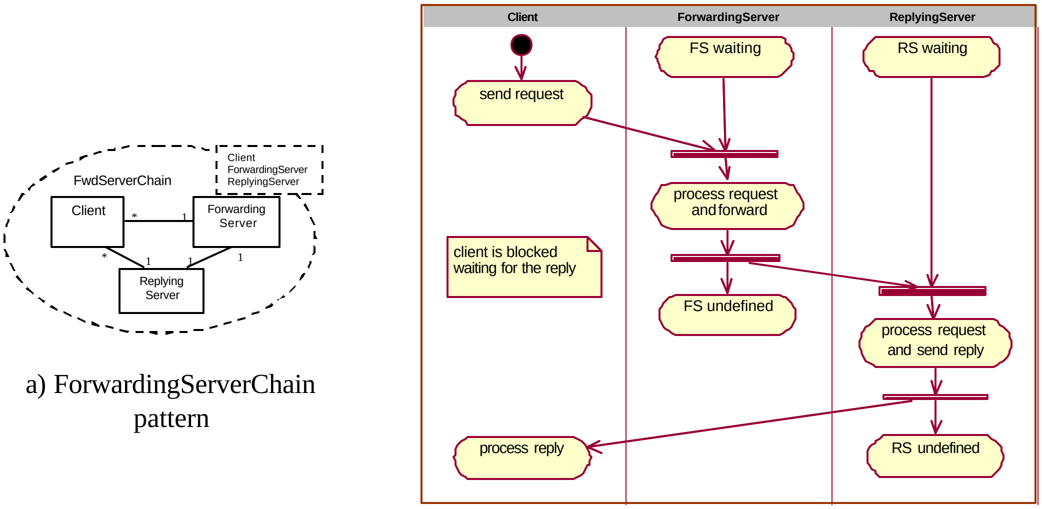


a) ClientServer collaboration

b) ClientServer with rendezvous

c) ClientServer with asynchronous messages

Figure 5. ClientServer architectural pattern



a) ForwardingServerChain pattern

b) Behaviour of the ForwardingServerChain pattern

Figure 6. ForwardingServerChain architectural pattern

In Fig. 5 and 6 are shown the structure and behaviour of two patterns used in our case study: ClientServer and ForwardingServerChain. The ClientServer pattern has two alternatives: the one shown in Fig.5.b is using a rendezvous communication style (where the client sends the requests then remains blocked until the sender replies), whereas the one from Fig. 5.c is using an asynchronous communication style (where the client continues its work after sending the request, and later on will accept the server's replay). The ForwardingServerChain, shown in Fig.6, is an extension of the ClientServer pattern, where the client's request is served by a series of servers instead of a single one.

There may be more than two servers in the chain (although only two are shown in Fig.6). A server in the middle plays the role of ForwardingServer, as it forwards the request to the next server in the chain after doing its part of service. The last server in the chain plays the role of ReplyingServer, as it sends the reply back to the client. More architectural patterns and the corresponding rules for translating them into LQN are described by the authors in (Petriu, Shousha and Jalnapurkar, 2000) and (Petriu and Wang, 2000).

2.4. Graph Rewriting Systems (PROGRES)

The graph-grammar formalism is appropriate for the UML to LQN transformation because both UML and LQN models are described by graphs. We are using a known graph rewriting tool named PROGRES (PROgramming with Graph Rewriting Systems) to implement the formal transformations from UML to LQN (Schürr, 1990; Schürr, 1994; Schürr, 1997). This section describes briefly the concepts used, but it does not go into details.

The essential idea of all implemented graph grammars or graph rewriting systems is that they are a generalization of string grammars (used in compilers) or term rewriting systems. The terms “graph grammars” and “graph rewriting systems” are often considered synonymous. However, strictly speaking, a graph grammar is a set of production rules that generates a language of terminal graphs and produces nonterminal graphs as intermediate results. On the other hand, a graph rewriting system is a set of rules that transforms one instance of a given class of graphs into another instance of the same class of graphs without distinguishing terminal and nonterminal results.

In order to use PROGRES for the transformation of an attributed *input graph* (representing a UML model) into an attributed *output graph* (representing an LQN model) we have to define a *graph schema* that describes the static properties of the graph, i.e., the types/classes of the graph elements and their legal combinations. The schema shows the types of nodes and edges that can appear in both the input and the output graph. In the intermediary transformation stages, the graphs contain mixed nodes and edges. Applying a set of production rules in a controlled way performs the desired graph transformations. A production rule has a *left-hand side* defining a graph pattern that will be replaced by the *right-hand side* (another graph pattern). A rule also shows how to compute the attributes of the new nodes from the attributes of the nodes that were replaced. In addition to the production rules, PROGRES also offers tests, queries, transactions, and functions, that are used to completely define the rewriting rules by which we can query and change the graph (Schürr, 1994).

3. Principle of Transformation from UML to LQN

The starting point for our algorithm is a UML model containing the following elements:

- High-level software architecture represented by one or more collaboration diagrams showing the concurrent (distributed) component instances represented as active objects and the architectural patterns they participate in.
- Allocation of high-level software components to hardware devices, modelled as a deployment diagram.
- A set of key performance scenarios annotated with performance information (see section 5 for a concrete example). Each scenario can be given either as a sequence or as an activity diagram.

The output of our transformation algorithm is an LQN model that can be read and solved by the existing LQN solvers. This section presents the principle of the transformation at the UML and LQN notation level, whereas the following sections goes more deeply into the PROGRES transformation.

The following pseudocode describes the main steps of the algorithm:

```
1. Generate the LQN model structure
      1.1. determine LQN software tasks from the high-level architecture
      1.2. determine LQN hardware devices from deployment diagram
2. Generate LQN details on entries, phases, activities from scenarios
      2.1 transform scenarios represented as sequence diagrams into activity diagrams
      2.2 for each scenario process the corresponding activity diagram(s)
            2.2.1 match the communication pattern from the architectural pattern with
                  the messages between components given in the activity diagram
            2.2.2 identify the activity diagram elements corresponding to different LQN
                  entries, phases, and activities, and create the LQN elements
3. Traverse the LQN elements, compute their parameters and write out the model file.
```

Step 1 generates the model structure (i.e., the software and hardware tasks and their connecting arcs) from the high-level architecture of the UML model and from the deployment of software components to hardware devices. Two kinds of UML diagrams are taken into account in this step: a high-level collaboration diagram that shows the concurrent/distributed high-level component instances and the patterns in which they participate, and the deployment diagram. Figures 7 and 8 show these two diagrams for one of the case-study systems discussed in section 5, namely the Forwarding-ORB example. Each high-level software component is mapped to a LQN task, and each hardware device (processor, disk, communication network, etc.) is mapped to a LQN hardware task. The arcs between LQN nodes correspond to the links from the UML diagrams. It is important to mention that in the first transformation step from UML to LQN we take into account only the structural aspect of the architectural patterns; their behavioural aspect will be considered in the next step.

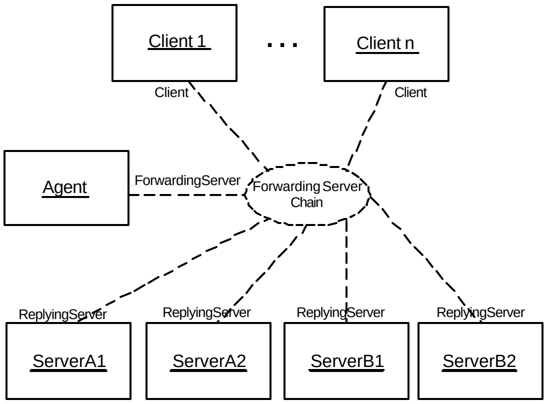


Figure 8. High-level architecture for the F-ORB system

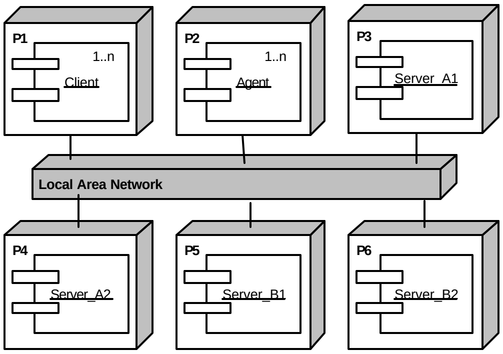


Figure 9. Deployment diagram for the F-ORB system

LQN task details are obtained in step 2 from scenario models. In general, scenarios can be represented in UML by sequence, collaboration or activity diagrams. (The first two are very close as descriptive power and have similar metamodel representation). UML statecharts are another kind of diagrams for behaviour description, but are not appropriate for describing scenarios. A statechart describes the behaviour of an object, not the cooperation between several objects, as needed in a scenario.

There are some well-known differences between sequence and activity diagrams. Sequence diagrams are very good at showing the responsibilities of different objects and the linear execution of sequential steps, but are not representing well concurrent flows of control. The present UML standard still lacks convenient features for representing loops, branches and fork/join structures in sequence diagrams. Other authors who are building performance models from UML designs have also pointed out this deficiency of the current UML standard, and are using instead extended sequence diagrams that look like the Message Sequence Chart standard (see, for example, Smith and Williams, 2001).

On the other hand, activity diagrams show very well the flow of control, but are not so good in showing the objects responsible for different actions. This problem was somewhat alleviated by the introduction of swimlanes in activity diagrams. Another difference is that the sequence diagrams are somewhat more compact as a notation than the activity diagrams. Considering the trade-offs, we have decided to use activity diagrams for generating LQN detailed elements, since concurrency and flow of control are important in our transformation algorithm. However, we also accept scenarios modelled as sequence diagrams if the designers consider them good enough for their purposes, but we transform them into activity diagrams before proceeding with the transformation to LQN.

Fig. 9 illustrates our approach to transforming sequence diagrams into activity diagrams (step 2.1 of the algorithm). We reserve a swimlane for each execution thread corresponding to a concurrent component (i.e., active object), which will contain also the operations done by passive objects executed in the same thread. The information on how to group the active and passive objects comes from the high-level architecture, where active instances may contain or use passive instances. In Fig. 9, for example, there are three active objects, x, y and z, in the sequence diagram, and each has a corresponding swimlane in the activity diagram. The concept behind the transformation from sequence to activity diagram is to follow the message flow in the sequence diagram by taking also into account the execution threads of the active objects involved. We have defined transformation rules for the following cases (see Petriu and Sun, 2000 for more details):

a. Sequential messages passed between objects in the same execution thread are mapped to an Action States in the corresponding swimlane (Figure 9, message "b").

b. Messages with condition guards that are alternatives of the same condition in the sequence diagram are mapped to a branching structure in the activity diagram (Figure 9, message "g" ,"h")

c. A synchronous message between objects running in different threads of control is treated as a join operation on the receiving side in the corresponding activity diagram, and its reply is treated as the corresponding fork (Figure 9, messages "a","r"). The object flow may be also shown, according to the current UML standard. The sender's execution thread is suspended from the moment it sent the message until receiving back the reply.

d. An asynchronous creation of an active object marks a fork operation in the corresponding activity diagram (Figure 9, new(z)).

e. An asynchronous message sent to another thread of control indicates a join operation on the receiver side and a fork operation on the sender side in the corresponding activity diagram The object flow is may be also shown.

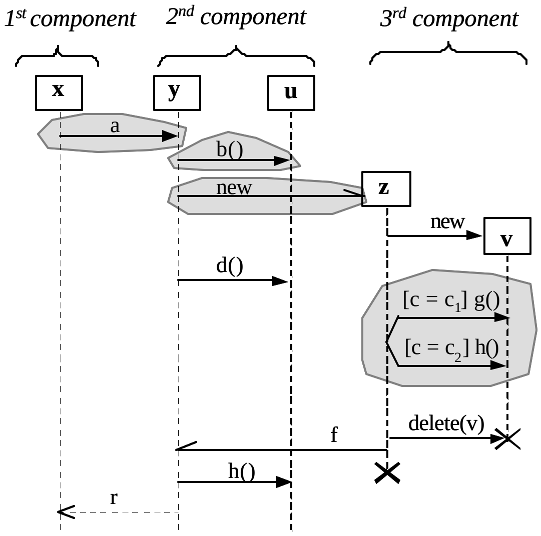


Figure 9a. Example of sequnce diagram with concurrent execution flows

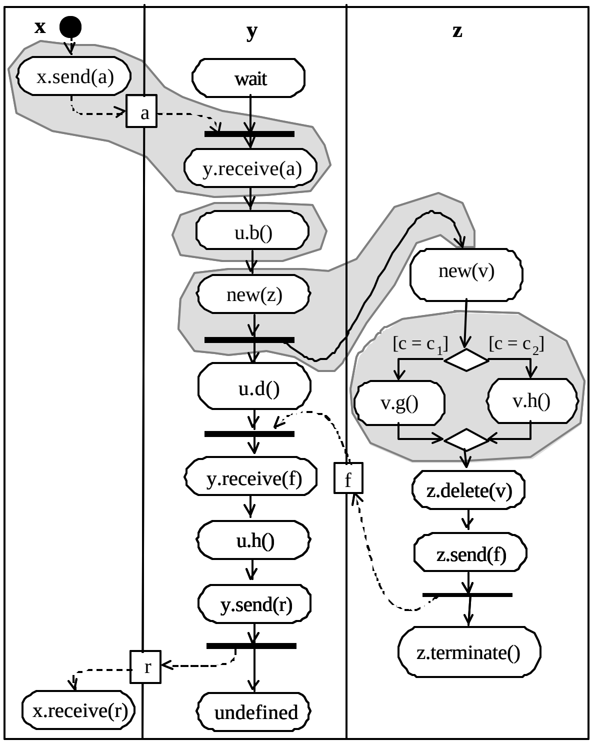
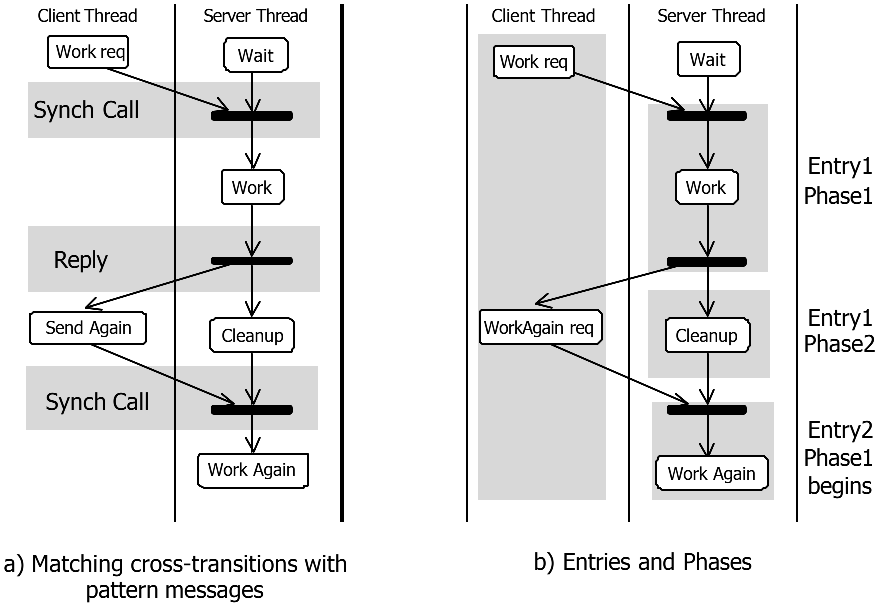


Figure 9b. Corresponding activity diagram obtained by automatic transformation

Step 2.2 of the algorithm processes the activity diagram for each scenario and identifies the LQN lower level elements: entries, phases and activities. Step 2.2.1 starts by traversing the activity diagram with the purpose of identifying the messages exchanged between concurrent components (i.e., those



a) Matching cross-transitions with pattern messages

b) Entries and Phases

Figure 10. Extracting entry and phase information from the activity diagram

that are crossing the swimlane boundaries). The intent is to overlay over the activity diagram the behavioural aspect of the architectural patterns involved, in order to verify whether the scenario is consistent with the patterns. Figure 10 illustrates this idea, by showing some inter-component messages that are identified and matched with the client-server pattern.

This information is used to generate the LQN elements (entries, phases and activities) as internal graph nodes in step 2.2.2. A task entry is generated for each kind of service offered by the corresponding software component instance. The set of all services offered by an instance is determined by looking at the messages received by this instance in all the scenarios considered for performance analysis. The LQN elements are generated as follows:

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

necessary for other patterns, such as "pipeline and filters", but they are not described in the paper (see Petriu, Shousha and Jalnapurkar, 2000) for more details on other patterns.

- LQN activities are created if a conditional or non-conditional branching state is encountered. In the case of conditional branching, an LQN “OrFork” is created to connect alternate activities. A probability for each branch is calculated based on the given guard condition. An “OrJoin” is created to end the conditional branching, corresponding to a “merge” pseudostate in the activity diagram.
- In the case of non-conditional branching, an LQN “AndFork” is created whenever a “Fork” pseudostate is used in the activity diagram to create a concurrent thread. However, the "Fork" pseudostates used by the servers for sending a reply at the end of phase 1 are an exception to this rule, and no explicit LQN “AndFork” is created in this case (see Fig. 10 for an intuitive explanation).
- An LQN request arc is generated when a communication is detected between a client entry (phase, activity) and a server entry, according to the corresponding high-level pattern. The visit ratio of the arc is given by the number of repetitions of the scenario step originating the request multiplied by the number of requests made in that step. If more scenario steps contained in the same phase are sending a request to the same entry, we have to add the visit ratio contributions of all these scenario steps.

The last step of the algorithm traverses the internal nodes representing the newly generated LQN model, computes its parameters (service times and visit ratios) and writes the model description to a text file in a format that can be read by the existing LQN solvers.

The service times for each phase (activity) has two parts: (a) the total CPU execution time for all scenario steps (i.e., activity diagram States) contained in the phase and (b) the communication overhead. According to the UML Performance profile, each scenario step has a tagged value PAdemand indicating its demand for CPU time, and another one, PAprob, giving the probability of its occurrence. (We assume here that the PAdemand value is converted in number of cycles for a given

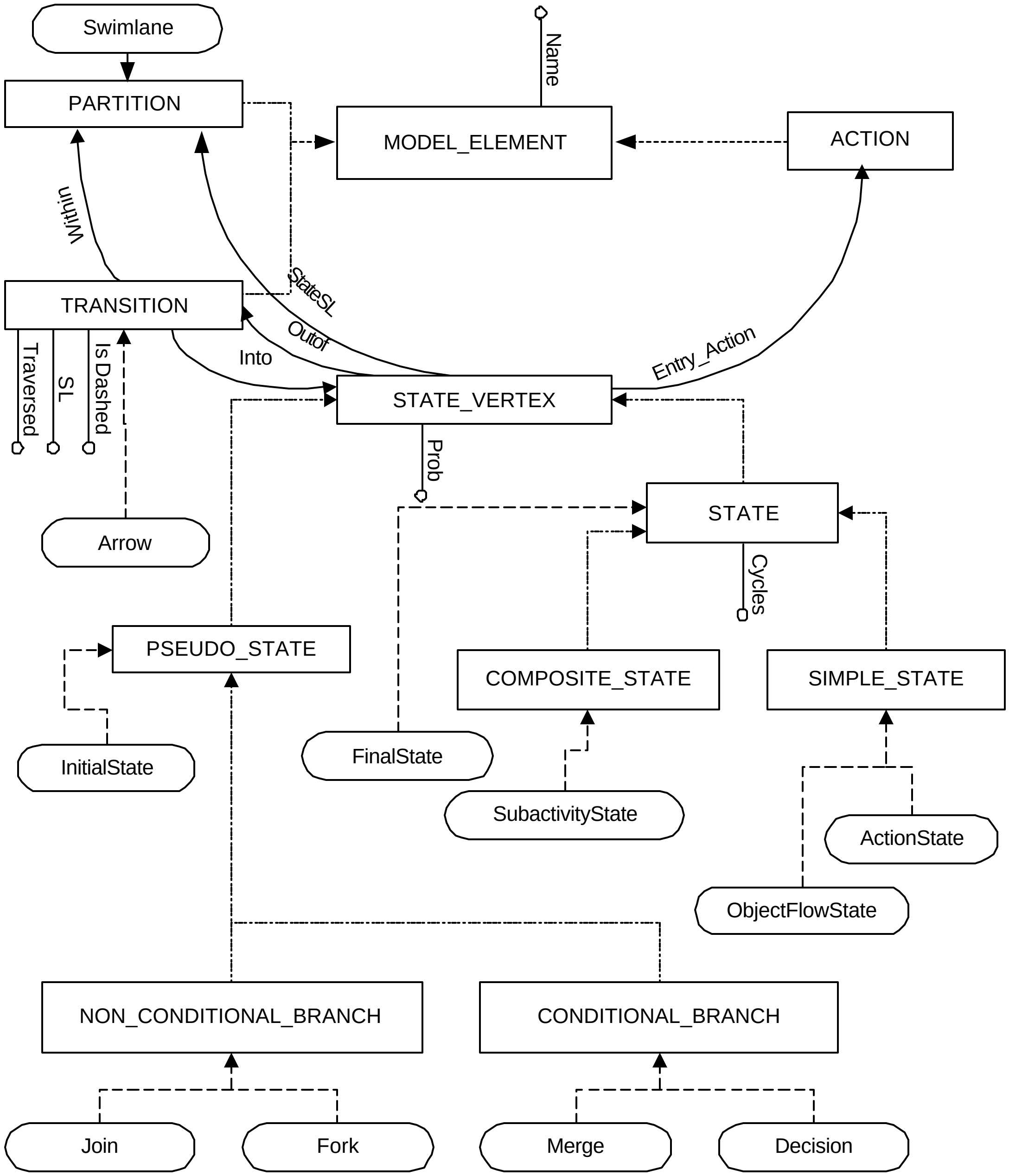


Figure 4. PROGRES sub-schema describing activity diagrams

processor, and is assigned to the attribute Cycle of the State that represents that step). Also, a processor has a tagged value representing its processing rate. The first line of the equation given below computes part (a) of the service time. Part (b) of the service time, the communication overhead is approximated separately for all messages sent (second line of the equation) and for all messages

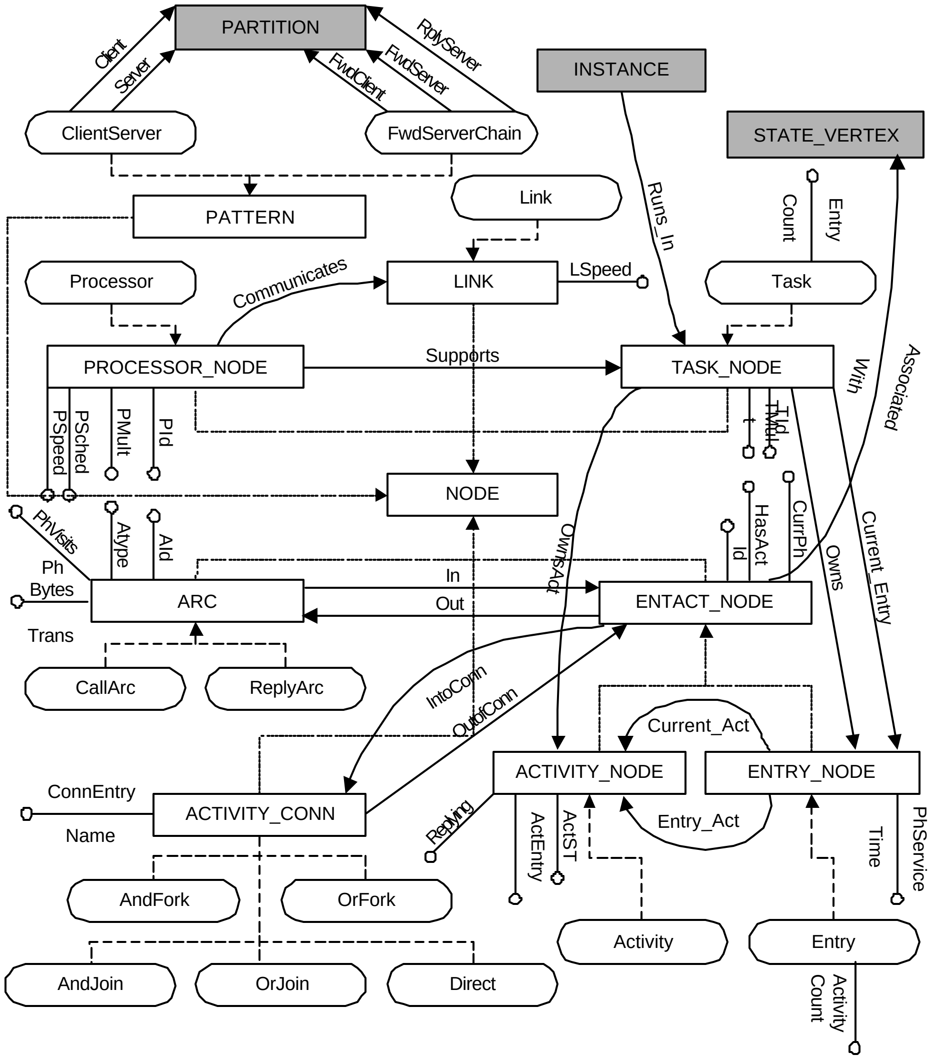


Figure 12. PROGRES sub-schema describing LQN models

received by that phase or activity (third line of the equation). SendOvhd represents the mean execution overhead for invoking the "send" primitive on a given platform, whereas the second term represents the time the CPU waits for sending a message. It is computed as the length of the message multiplied by the visit ratio of the outgoing arc and divided by the speed of the link that carries that message. This term has an impact on the total service time only for long messages sent over slow communication links (for short communication messages sent over fast communication links, the values of the term is insignificant). If a message is exchanged between two tasks co-allocated on the same processor, the last term is not included in the CPU time. The overhead due to receiving messages is estimated in the same way. The entire formula is as follows:

$$\begin{aligned}&ServiceTime = \sum ActionState.Cycles * ActionState.\mathrm{Pr}ob / \mathrm{Pr}ocRate + \\ &+ \sum (SendOvhd + OutArc.MsgLength * OutArc.VisitRatio / LinkSpeed) + \\ &+ \sum (RcvOvhd + InArc.MsgLength * InArc.VisitRatio / LinkSpeed)\end{aligned}$$

In this section we have presented the transformation approach at a high conceptual level, by using the UML and LQN graphical notation. The next section presents briefly issues regarding the implementation of the transformation algorithm with PROGRES, which performs transformation on the internal data structure representing the two models.

4. PROGRES-based transformation

As mentioned in section 2.4, a PROGRES program has two parts: a schema that defines the static properties of the graph, and a set of features (composed of production rules, tests, queries, transactions, and functions) that are used to completely define the rewriting rules for transforming the graph.

In our case, the schema has two parts: a) a simplified subset of the UML metamodel (re-written according to the PROGRES syntax) which represents the diagram types contained in the UML input model, as explained in the previous section, and b) our own definition of the LQN model. We have not translated the whole UML metamodel into PROGRES, only the parts that are necessary. Even these parts are simplified, in the sense that we skipped over some abstract classes to reduce the size of the schema, and expressed as PROGRES node attributes only those UML metamodel attributes that were strictly necessary for building the performance model. We have also added attributes that correspond to the tagged values from the UML performance profile, such as those giving the CPU demand and probability of an execution step.

Due to its size, we show here only a part of the schema: Fig. 11 represents the sub-schema for activity diagrams, and Fig. 12 the sub-schema for LQN models. PROGRES uses inheritance (possible multiple inheritance) to define hierarchies of node classes. *Square boxes* represent *node classes*, and the inheritance relationships are represented with *dotted edges*. Node classes correspond to abstract

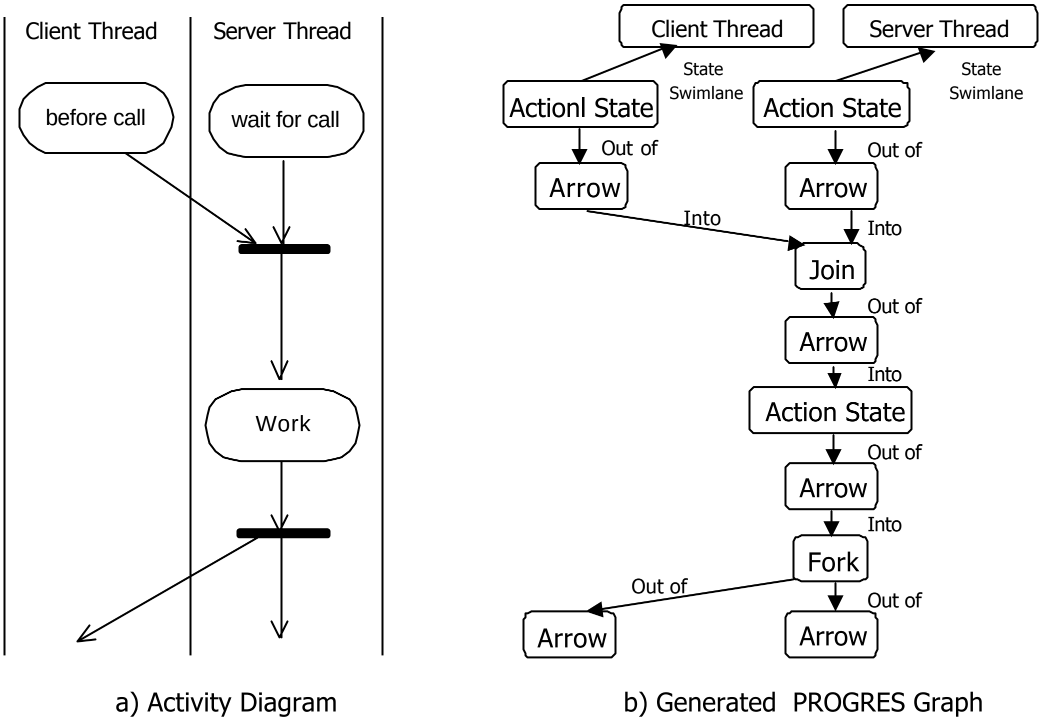


Figure 13. Example of an activity diagram and its PROGRES graph

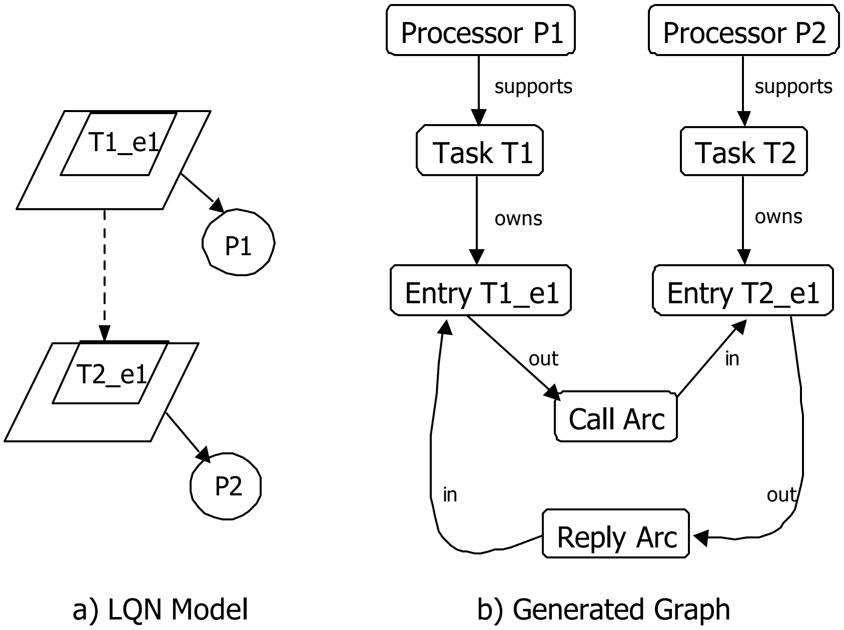


Figure 14. Example of an LQN model and its PROGRES graph

classes in UML, i.e., node classes do not have any direct node instances. A node class has an optional list of attributes. *Rounded-corner boxes* represent *node types*, which are connected with their uniquely defined classes by the means of *dashed edges*. Node types are leaves of the node class hierarchy, and are used to create node instances in a PROGRES graph. A node type specializes only one class and inherits all its properties. *Solid edges* between edge classes represent *edge types,* which define the relationships between node instances. *Node attributes* are shown as small circles attached to the class or type boxes. The classes shadowed in Fig. 12 are repeated from Fig. 11.

The schema describes the static properties of the internal data structure (a graph) that represents at the beginning only the UML input model, and grows gradually so that by the end represents also the LQN output model. Fig. 13 gives an example of a PROGRES graph that represents a simple activity diagram, and Fig.14 one that represents a simple LQN model. As expected, the internal data structure is more complex and less understandable than the UML or LQN graphical notation. However, it is at the greater level of detail that the algorithm presented in section 3 was implemented. The detailed presentation of the implementation is beyond the scope of this paper (more details can be found in Amer, 2001).

5. Case Study

In this section are presented the results of the UML to LQN transformation algorithm applied to three CORBA-based client-server systems. The LQN model generated by using PROGRES were solved under different workloads with an existing LQN analytic solver (Franks, 2000). The results were compared with measurements obtained from actual implementations, taken from a performance study by Abdul-Fatah and Majumdar, 1998. Based on a Commercial-Off-The-Shelf (COTS) middleware product called Orbeline (currently sold as Visibroker by Inprise) and a synthetic workload running on a network of Sun workstations using Solaris 2.6, Abdul-Fatah and Majumdar have implemented three performance prototypes, and measured them by using Solaris system calls buried into the prototype software. The three systems were the Handle-driven ORB (H-ORB), the Forwarding ORB (F-ORB) and the Process Planner (P-ORB). The H-ORB was the basic Orbeline product, whereas the F-ORB and P-ORB were built using additional processes in conjunction with the Orbeline middleware. In (Petriu, Amer et al, 2000) is presented an analytical performance study of two of the systems, H-ORB and F-ORB, by using "hand-built" LQN models. The LQN models developed in (Petriu, Amer et al, 2000) are equivalent to the models derived automatically by our UML to LQN transformation.

In their study, Abdul-Fatah and Majumdar have implemented a synthetic application in which two distinct services, A and B, are using the ORB. A client executes a cycle repeatedly, making one request to Server A and one to Server B. Two copies of A, called A1 and A2, as well as two copies of B, called B1 and B2, are provided. The two copies of each server enable the system to handle more load and allow us to investigate the impact of load balancing that is provided by many commercial ORB products. The client performs a *bind* operation before every request. 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 agent forwards the client request to the appropriate server. The server then returns the results of the computations directly 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 results and sends them back to the client. When a service is requested form a particular server, the server process executes a loop and consumes a pre-determined amount of CPU time. The synthetic application is used because it provides flexibility in experimentation with various levels of different workload parameters, such as the service time at each server, and the inter-node delay.

The synthetic application is characterized by a number of parameters that are briefly summarized.

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

Service Demands (SA, SB): The time required by server A and B, respectively, to provide the service. Whenever a particular server A (or B) is invoked it consumes SA (or SB) units of CPU time.

Inter-Node Delay (D): Since the experiments were performed on a local area network, the inter-node delay that would appear in a wide-area was simulated by making a sender process to sleeps for D units of time before sending a message. However, in case of the H-ORB agent there was no access to the source code, and the inter-node delay for the handle returning operation was simulated by making the client sleep for D units of time before receiving the message.

Message Length (L): The size of the actual message sent by the client to the server, or returned by the server. (In the experiments, the message content was not important, only the size).

***Degree of Cloning*:** the concurrency degree of the agent process. A clone of a process is a copy that shares the message queue with its parent. A cloned process is represented in LQN as a multi-server.

The following table summarizes the values for the workload factors by Abdul-Fatah and Majumdar:

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

5.1. H-ORB

The input to the UML to LQN transformation for the H-ORB case is represented by a) the deployment diagram from Fig.8 (which is identical for all three cases), b) a collaboration diagram similar with

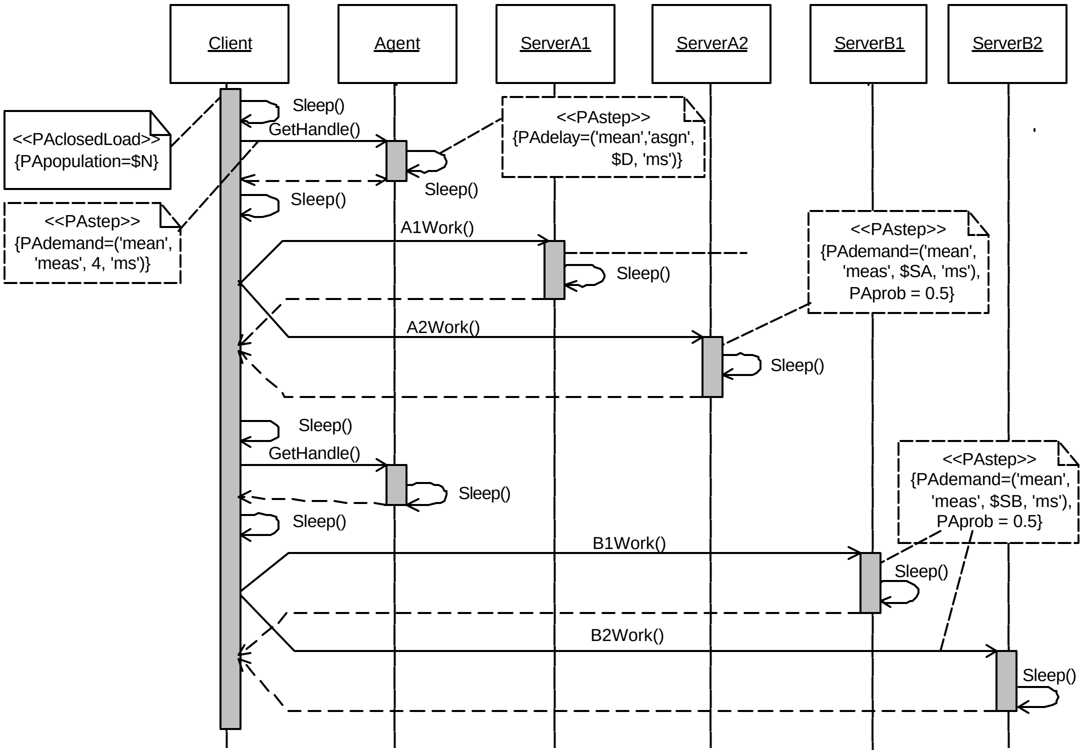

Client
Agent
ServerA1
ServerA2
ServerB1
ServerB2
<<PAclosedLoad>>
{PApopulation=$N}
<<PAstep>>
{PAdemand=('mean', 'meas', 4, 'ms')}
<<PAstep>>
{PAdelay=('mean','asgn', $D, 'ms')}
<<PAstep>>
{PAdemand=('mean', 'meas', $SA, 'ms'), PAprob = 0.5}
<<PAstep>>
{PAdemand=('mean', 'meas', $SB, 'ms'), PAprob = 0.5}
Sleep()
GetHandle()
A1Work()
A2Work()
B1Work()
B2Work()

Figure 15. Client request scenario in H-ORB

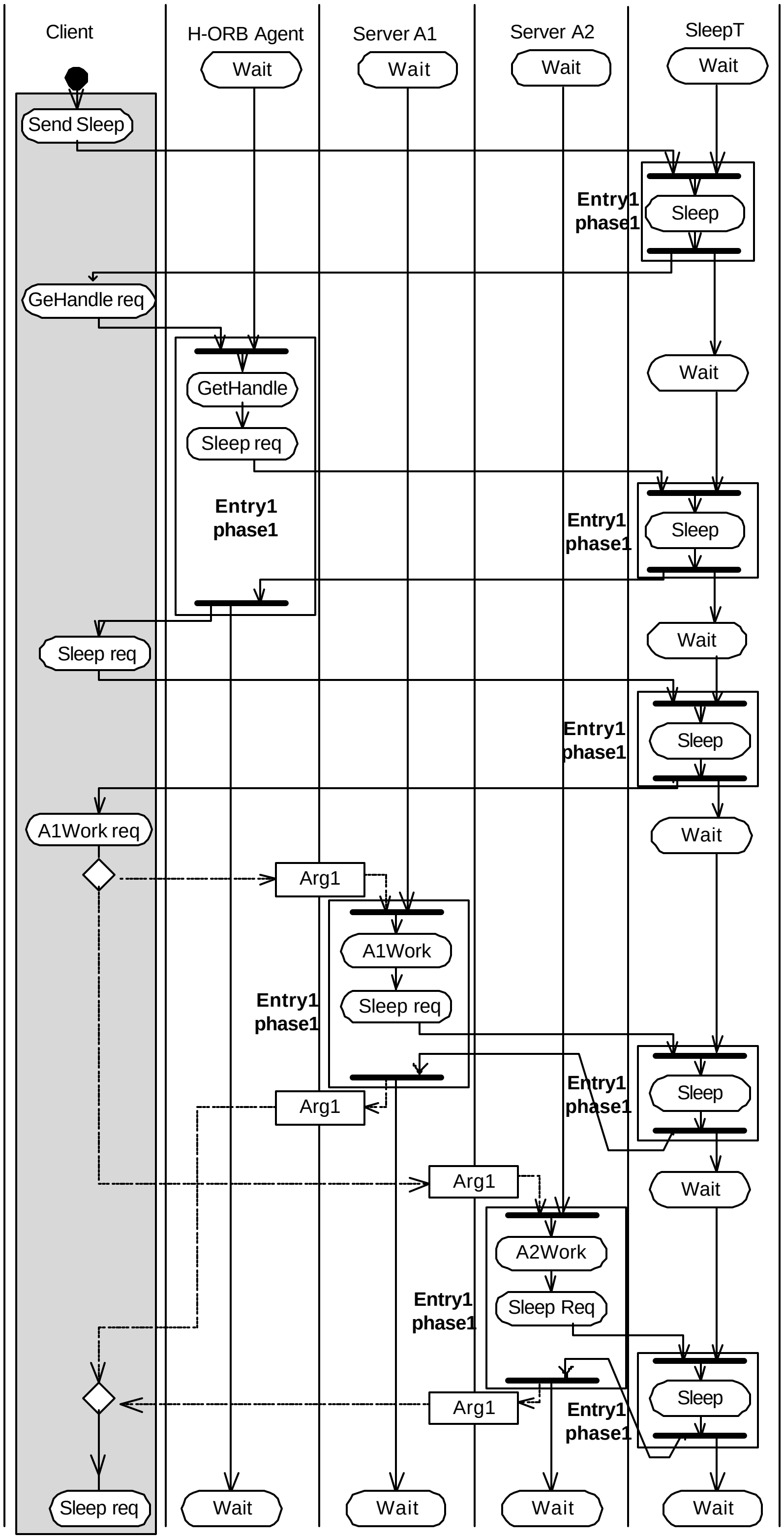


Figure 16. Partial activity diagram automatically generated for H-Orb

Fig.7, where the client/server pattern is used twice (between clients and agent, and between clients and servers), and c) the sequence diagram from fig.15. The sequence diagram is annotated with performance information according to the UML performance profile. The <<PAclosedLoad>> stereotype indicates that the scenario is used under a closed load with a population of N. ($N indicates a variable, to be substituted by a concrete value when the model is actually generated). A <<PAstep>> stereotype is applied to each of the actions triggered by the following messages: GetHandle(), Sleep(), A1Work(), A2Work(), B1Work() and B2Work(). All scenario steps are characterized by a certain PAdemand value (which represents the CPU execution time), with the exception of Sleep(), which is characterized by a PAdelay value (delay without consuming CPU time). The server operations have also a PAprob value of 0.5, which indicates their probability of being chosen.

Fig. 16 shows the beginning of the activity diagram that was generated automatically from the sequence diagram from Fig. 15. (The activity diagram covers the communication between the client and the agent, and between the client one of the servers A). The activity diagram contains a swimlane for each concurrent component. In this case, the sequence diagram did not contain any passive objects. An additional swimlane was created for an "artificial" Sleep task needed to implement the

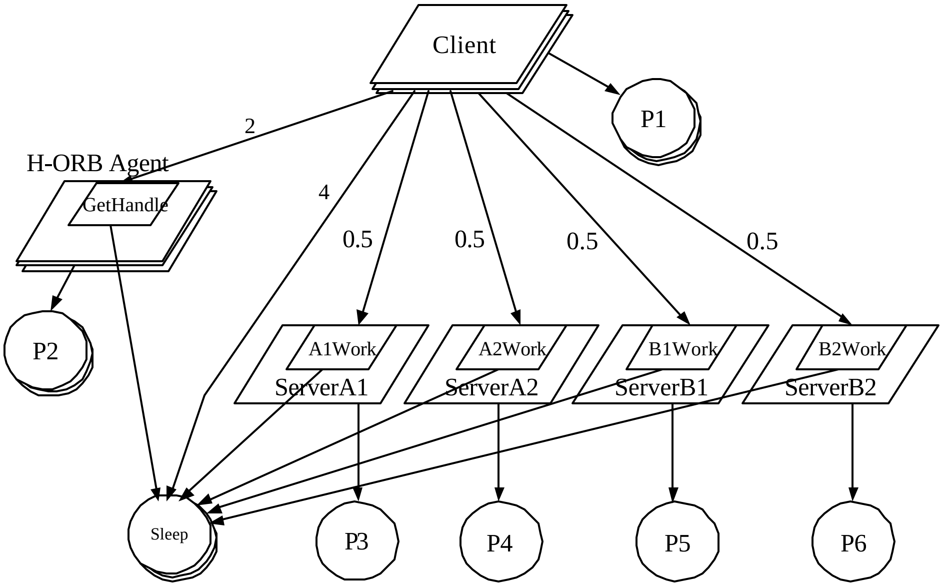


Figure 17. LQN model automatically generated for H-ORB

sleep operation, away from any of the existing processors. (Such a task is created every time the tagged value PAdemand is used). In the LQN model, Sleep will be implemented as a delay server (also known as infinite server), for which there is no queueing. Fig. 16 shows also how the different activity diagram states have been grouped into entry and phases. The object flow Arg1 that represents the content of the messages exchanged between the client and the servers was generated because the messages had arguments (not shown in the sequence diagram). What is important for the performance model is the size of the message, not its actual content. We have realized that, so far, the UML Performance Profile does not have a tagged value to define the message size. (However, we have added such an attribute to the PROGRES schema). As shown in Table 1, the size of the message was one of the experimental factors.

The LQN model automatically generated for F-ORB is given in Fig. 17.

The model was then given as input to the analytical LQN solver, obtaining 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 2: H-ORB Model Results VS Measured Values

A comparison between the model results and the measured values is depicted in Fig.18.

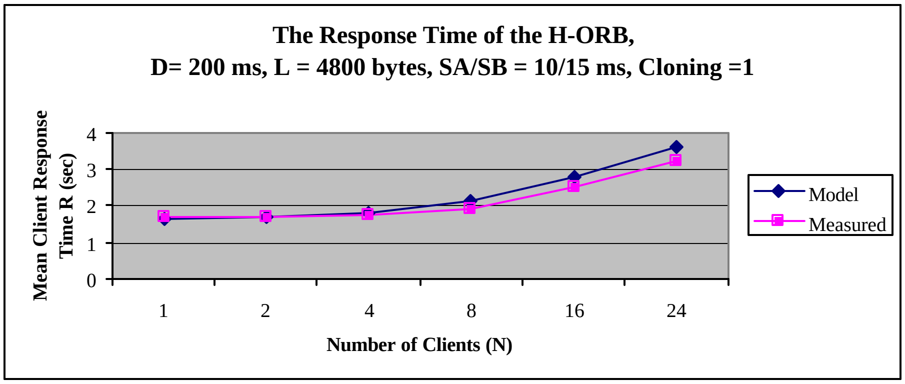


Figure 18: H-ORB Model results VS Measured values Graph

2.2. F-ORB results

As stated in (Abdul-Fatah and Majumdar, 1998), the Forwarding ORB (F -ORB) architecture differs from H-ORB in the sense that the F-agent forwards the reply to the desired server rather than returning the handle to the requesting client. During each experiment, a fixed number of F-agents 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.

Due to space limitations, we cannot show the diagrams for the UML input model and the generated LQN model, which are presented in (Amer, 2001). The LQN model was solved with the analytical solver, giving the following results for the Mean Client Response Time (in seconds) for different numbers 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 3: F -ORB Model Results VS Measured Values

The model results and the measured values for a cloning level of 1 are shown in Fig.19

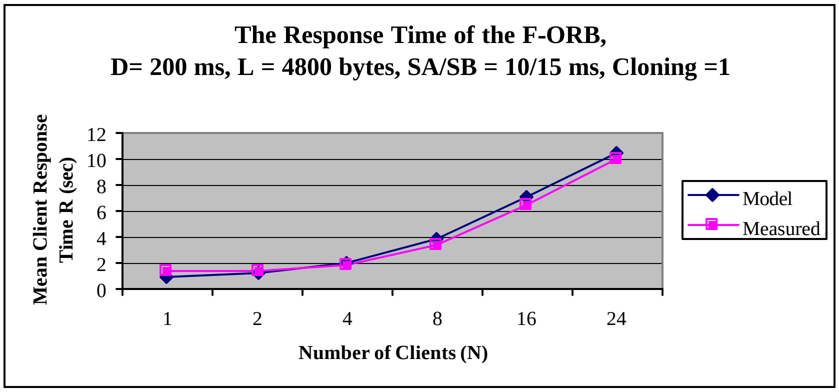


Figure 19: F-ORB Model Results VS Measured Values

5.3. P-ORB results

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 asynchronously since the design assumes no interdependencies between the two constituent requests. Both servers are invoked using a one-way send. 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. For more details, see (Abdul-Fatah and Majumdar, 1998).

Due to space limitations, we cannot show the diagrams for the UML input model and the generated LQN model, which are presented in (Amer, 2001). The LQN model was solved with the analytical solver, producing the results shown in Table 4 and Fig. 20 for the Mean Client Response Time (in seconds) for different numbers 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 2: P-ORB Model Results VS Measured Values

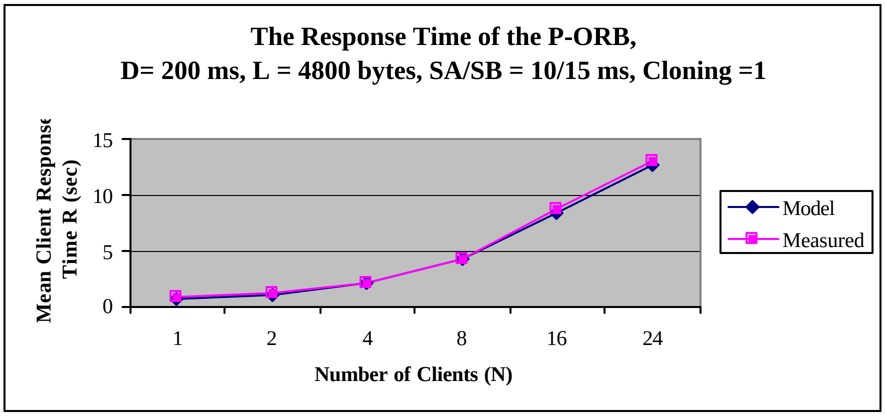


Figure 20: P-ORB Model Results VS Measured Values Graph

6. Conclusions

The paper presents a transformation algorithm from UML models (annotated with performance information) to LQN performance models. The algorithm was implemented with PROGRES, a known graph-rewriting tool. Automatically generated LQN models were solved analytically, then compared with measurements obtained from three CORBA-based systems. The model results are reasonably close to the measurements, which demonstrate that the approach is valid. This works contributes to bridging the gap between software design and performance analysis. It also offers promises that in the future, performance model generators could be integrated with UML tools, facilitating the quantitative analysis of software designs from the early stages throughout the software life cycle.

The successful use of a graph rewriting tool to convert UML models into LQN model represents a proof of concept that graph-grammar techniques work for this kind of problem. However, although PROGRES is a very powerful tool, it introduces additional steps both in the algorithm implementation and in the transformation process. One disadvantage is that we had to convert the UML metamodel into a PROGRES schema during the development of the algorithm. Another disadvantage is that every UML model has to be converted into a PROGRES input graph every time we want to generate its corresponding performance model. Therefore, by using the lessons learned from this experience, we started working on a graph transformation algorithm that works directly on the internal data structure of a UML model. This is, in fact, a graph described by a schema that is exactly the UML metamodel. The disadvantage of such an approach is that we will have to implement from scratch the graph transformation operations that are provided by the general-purpose graph rewriting tools such as PROGRES. The advantage is, however, that we will eliminate a lengthy intermediary step and obtain a faster transformation.

Another approach we are investigating is to perform the transformation at the XML level. The UML standard defines XMI, an interface from UML to XML. Every UML tool is supposed to implement this interface, and therefore to generate XML files that describe the UML models developed with the tool. There are new XML transformation languages, such as XSLT, and free tools to support them.

We started to investigate whether XSLT and its supporting tools are powerful enough for our application.

Other directions for future work include: the addition of new architectural patterns to the transformation, the application of the UML to LQN transformation to a wider class of systems, and the possibility of generating another kind of performance models besides LQN.

Acknowledgements

This work was partially supported by grants from the Natural Sciences and Engineering Research Council of Canada (NSERC) and Communications and Information Technology Ontario (CITO).

References

Abdul-Fatah, I., Majumdar, S., 1998. “Performance Comparison of Architectures for Client-Server Interactions in CORBA”, Proc. IEEE International Conference on Performance, Computing, and Communication, Tempe, Arizona.

Amer, H., 2001. "Automatic Transformation of UML Software Specifications into LQN Performance Models by using Graph Grammar Techniques", Master Thesis, Carleton University, Ottawa, Canada.

Balsamo, S., Simeoni, M., 2001. “On transforming UML models into performance models”, Workshop on Transformations in the Unified Modeling Language, Genova, Italy.

Booch, G., Rumbaugh, J., Jacobson, I., 1999. The Unified Modeling Language User Guide, Addison-Wesley.

Buchmann, F., Meunier, R., Rohnert, H., Sommerland, P., Stal, M., 1996. Pattern-Oriented Software Architecture: A System of Patterns, Wiley Computer Publishing.

Cortellessa, V. Mirandola, R., 2000. “Deriving a Queueing Network based Performance Model from UML Diagrams”, In Proc. of the 2nd International Workshop on Software and Performance, Ottawa, Canada, pp.58-70.

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

Franks, G., 2000. "Performance Analysis of Distributed Server Systems", Ph.D. Thesis, Carleton University, Ottawa, Canada.

Gomaa, H., Menasce, D.A., 2000. “Design and Performance Modeling of Component Interconnections Patterns for Distributed software architectures, Proceedings of 2nd ACM Workshop on Software and Performance, WOSP’2000, Ottawa, Canada, pp.117-126.

Kähkipuro, P., 2001. “UML-Based Performance Modeling Framework for Component-Based Distributed Systems”, in R.Dumke et al.(Eds): Performance Engineering, LNCS 2047, Springer, pp.167-184.

Neilson, J.E., Woodside, C.M., Petriu, D.C., and Majumdar, S., 1995. "Software bottlenecking in client-server systems and rendezvous networks", IEEE Trans. on Software Eng., vol. 21(9) pp.776-782.

Object Management Group, 1999. UML Specification Version 1.3, OMG Doc. ad/99-06-08.

Object Management Group, 2001. UML Profile for Scheduling, performance and Time, OMG Document ad/2001-06-14, http://www.omg.org/cgi-bin/doc?ad/2001-06-14.

Petriu, D.C., Amer, H., Majumdar, S., Abdul-Fatah, I., 2000 “Using Analytic Models for Predicting Middleware Performance”, Proceedings of 2nd International Workshop on Software and Performance, Ottawa, Canada, pp 189-194.

Petriu, D.C., Shousha, C., Jalnapurkar, A., 2000, "Architecture-Based Performance Analysis Applied to a Telecommunication System", IEEE Trans. on Software Eng., Vol.26, No.11, pp. 1049-1065.

Petriu, D.C., Wang, X., 2000. "From UML description of high-level software architecture to LQN performance models", in Applications of Graph Transformations with Industrial Relevance AGTIVE'99 (eds. M.Nagl, A. Schürr, M. Muench), LNCS 1779, pp. 47-62, Springer.

Petriu, D.C., Sun, Y., 2000 "Consistent Behaviour Representation in Activity and Sequence Diagrams", in *UML'2000 The Unified Modeling Language - Advancing the Standard*, LNCS 1939, pp.369-382, Springer.

Rolia, J.A., Sevcik, K.C., 1995. “The Method of Layers”, IEEE Trans. on Software Eng., Vol. 21, Nb. 8, pp 689-700.

Schürr, A., 1990. “Introduction to PROGRES, an attribute graph grammar based specification language”, in Graph-Theoretic Concepts in Computer Science, M. Nagl (ed), Vol. 411 of Lecture Notes in Computer Science, pp 151-165.

Schürr, A., 1994. “PROGRES: A Visual Language and Environment for PROgramming with Graph Rewrite Systems”, Technical Report AIB 94-11, RWTH Aachen, Germany.

Schürr, A., 1997. “Programmed Graph Replacement Systems”, in Handbook of Graph Grammars and Computing by Graph Transformation, G. Rozenberg (ed), pp 479-546.

Shaw, M., 1996. “Some Patterns for Software Architecture”, In Pattern Languages of Program Design 2 (J.Vlissides, J. Coplien, and N. Kerth eds.), pp.255-269, Addison Wesley.

Smith, C.U., 1990. Performance Engineering of Software Systems, Addison Wesley.

Smith, C.U., Williams, L.G., 2001. Performance Solutions: A Practical Guide to Creating responsive, Scalable Software, Addison Wesley.

Woodside, C.M., 1988. "Throughput Calculation for Basic Stochastic Rendezvous Networks", Performance Evaluation, Vol.9(2), pp. 143-160.

Woodside, C.M., Neilson, J.E., Petriu, D.C., Majumdar, S., 1995. “The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software”, IEEE Trans. on Computers, Vol.44(1), pp 20-34.