Applying the UML Performance Profile: Graph Grammar-based Derivation of LQN Models from UML Specifications [*]

Dorina C. Petriu and Hui Shen
Carleton University,
Systems and Computer Engineering
Ottawa ON Canada, K1S 5B6
petriu@sce.carleton.ca

February 2002

Abstract

The Object Management Group (OMG) is in the process of defining a UML Profile for Schedulability, Performance and Time that will enable the construction of models for making quantitative predictions regarding these characteristics. The paper proposes a graph-grammar based method for transforming automatically a UML model annotated with performance information into a Layered Queueing Network (LQN) performance model. The input to our transformation algorithm is an XML file that contains the UML model in XML format according to the standard XMI interface. The output is the corresponding LQN model description file, which can be read directly by existing LQN solvers. The LQN model structure is generated from the high-level software architecture and from deployment diagrams indicating the allocation of software components to hardware devices. The LQN model parameters are obtained from detailed models of key performance scenarios, represented as UML interaction or activity diagrams.

1 Introduction

The Object Management Group (OMG) is in the process of defining a UML Profile for Schedulability, Performance and Time [10] that would enable the construction of models that can be used for making quantitative predictions regarding these characteristics. The original RFP issued in March 1999 was followed by the first "Response to RFP" submission in August 2000, and by a revised submission in June 2001 [10]. The later includes

[*]To be presented at Performance TOOLS'2002, London, UK, April 2002

some additional aspects not covered in the former, among which is a section dealing with performance analysis. The proposed performance profile extends the UML metamodel with stereotypes, tagged values and constraints, which make it possible to attach performance annotations (such as resource demands and visit ratios) to a UML model. In order to conduct quantitative performance analysis of an annotated UML model, one must first translate it into a performance model, use an existing performance analysis tool for solving the performance model and then import the performance analysis results back in the UML model. The focus of this paper is the first step of the process. The paper proposes a graph-grammar based method for transforming a UML model annotated with performance information into a Layered Queueing Network (LQN) performance model [20], [21]. The input to our transformation algorithm is an XML file that contains an annotated UML model in XML format according to the standard XMI interface [9], and the output is the corresponding LQN model description file, which can be read directly by existing LQN solvers [5]. The transformation algorithm described in the paper was completely implemented in Java on top of a rather large open source Java library that implements and manipulates the UML metamodel [23]. More work is necessary for the present implementation to become a tool prototype, especially the addition of a suitable GUI.

The transformation approach is as follows: the LQN model structure is generated from the high-level software architecture, more exactly from the architectural patterns showing high-level software components and their relationships, 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.

The paper continues previous work of the authors from [12], [11], [13], [1]. In [12] the system's architectural patterns were used to build LQN performance models, but the model derivation was not automated. In [11] an automated graph transformation was proposed to derive the LQN model structure from the high-level architecture and the architectural patterns. The transformation was implemented by using an existing graph rewriting tool named PROGRES [17]. The PROGRES-based transformation was extended in [1] to obtain the whole LQN model (including its parameters) from a UML model represented as a PROGRES input graph. The merit of the PROGRES-based approach is that it represents a proof of concept that graph-transformation techniques can be applied successfully to the derivation of performance models from UML models. Although PROGRES had the advantage of freeing us from low-level operations of graph matching and manipulation, it also brought the disadvantage of introducing additional steps in the process. For example, we had to express the UML metamodel as a PROGRES schema, and to convert each UML model into a PROGRES input-graph format. The solution presented in this paper eliminates the general-purpose graph rewriting tool from the loop by performing the manipulation and transformation of the UML model directly at the metamodel level. This solution is more efficient than the PROGRES-based one, as it has fewer steps; also, the ad-hoc transformation is faster, as it is tailored to the problem at hand. Another advantage of working directly at the UML metamodel level is that it becomes possible to merge the performance model builder with a UML tool.

Since the introduction of the Software Performance Engineering technique in [18], there

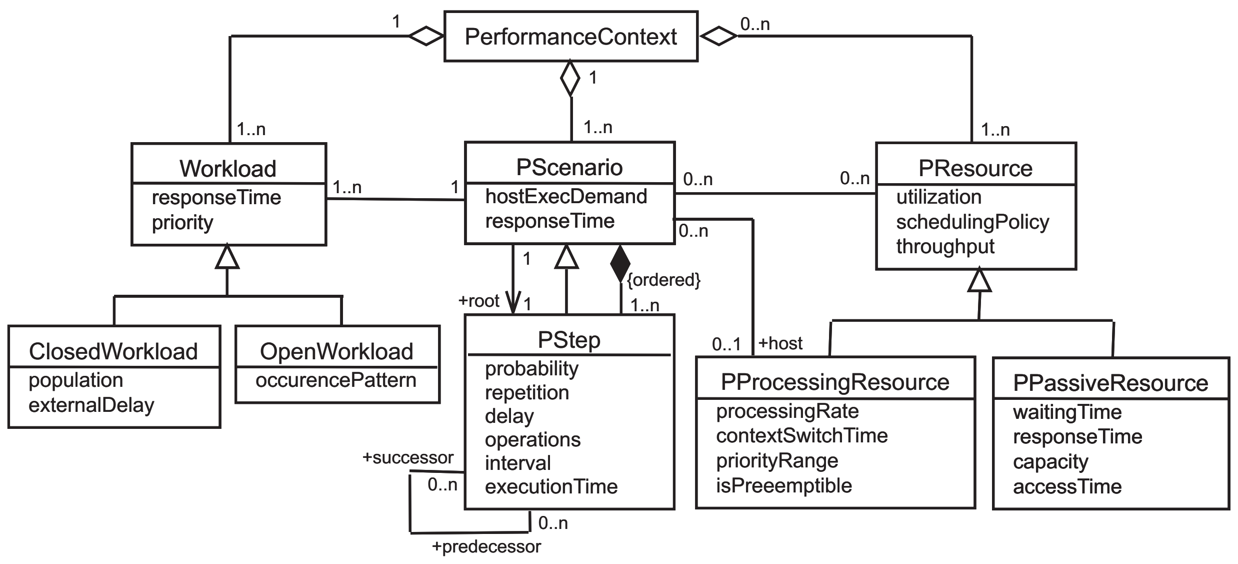


Figure 1: Domain model in the UML Performance Profile

has been a significant effort to integrate performance analysis into the software development process throughout all lifecycle phases. This requires the ability to derive 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 [2]. Among the techniques surveyed, the one from [3] follows the SPE methodology very closely. 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 [18], [19]. The work presented in [8] introduces an UML-based notation and framework for describing performance models, and a set of special techniques for modelling component-based distributed systems. In [4] the idea of patterns is used to investigate the design and performance modelling of interconnection patterns for client/server systems.

Compared to the existing research, our paper is a unique combination of the following characteristics: a) it accepts as input XML files produced by UML tools, b) it generates LQN performance models by applying graph transformation techniques to graphs of metaobjects that represent different UML diagrams of the input model, and c) it uses the UML performance profile for adding performance annotations to the input model.

2 Background

2.1 UML Performance Profile

According to [10], 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. 1, 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 profile maps the classes from Fig. 1 to a stereotype that can be applied to a number of UML model elements, and each class attribute to a tagged value. For example, the basic abstraction PStep is mapped to the stereotype `<<PAstep>>` that can be applied to the following UML model elements:`Message` and `Stimulus` (when the scenario is represented by an interaction diagram) or `ActionState` and `SubactivityState` (when the scenario is represented by an activity diagram).

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

Fig. 2 illustrates the inter-operability of the different tools involved: a UML tool (such as Rational Rose or ArgoUML [22]), a performance model solver (LQN analytical solver or simulator) and our UML to LQN transformation implementation. So far we have made progress on the forward path (represented with black arrows) but have not attempted the backward path yet (represented with gray arrows).

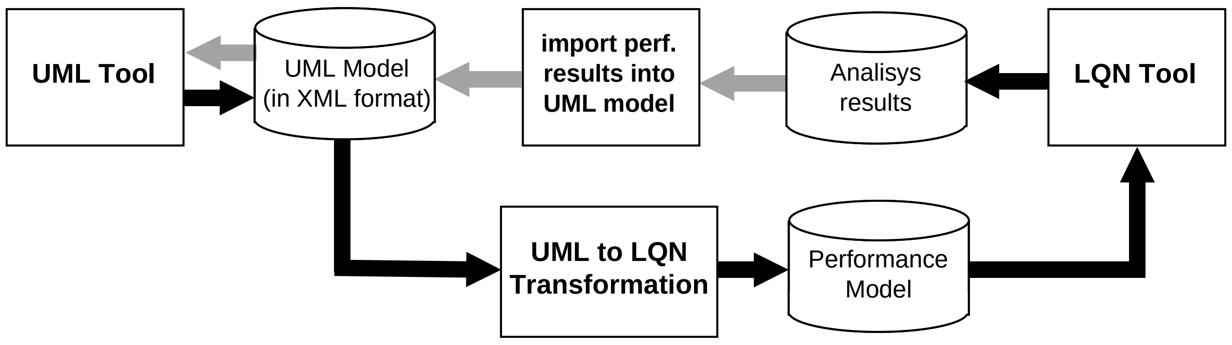

UML Tool
UML Model (in XML format)
import perf. results into UML model
Analisys results
LQN Tool
UML to LQN Transformation
Performance Model

Figure 2: Tool inter-operability

2.2 The LQN model

LQN was developed as an extension of the well-known QN model, at first independently in [20], [21] and [15], then as a joint effort [5]. The LQN toolset presented in [5] includes both simulation and analytical solvers. The main difference with respect to QN is that LQN can easily represent nested services: a server which receives and serves client requests, may become in turn a client to other servers from which it requires nested services while serving its own clients.

A LQN model is represented as an acyclic graph, whose nodes represent software entities and hardware devices, and arcs denote service requests. The software entities (also known as tasks) are drawn as parallelograms, and the hardware devices as circles. The nodes with outgoing but no incoming arcs play the role of clients, the intermediate nodes with both incoming and outgoing arcs are usually software servers and the leaf nodes are hardware servers (such as processors, I/O devices, communication network, etc.) A software or hardware server node can be either a single-server or a multi-server. Fig. 3 shows a simple example of a LQN model of a web server: at the top there is a customer class with a given number of stochastical identical clients. Each client sends demands for different services of the WebServer. Each kind of service offered by a LQN task is modelled as a so-called entry, drawn as a parallelogram "slice" in the figure. Every entry has its own execution times and demands for other services, given as model parameters. In this case, the WebServer entries require services from different entries of the Database task. Each software task is running on a processor shown as a circle. Also as circles are shown the communication network delays and the disk device used by the Database.

It is worth mentioning that the word layered in the LQN name does not imply a strict layering of tasks (for example, tasks in a layer may call each other or skip over layers). All the arcs used in this example represent synchronous requests, where the sender of a request message is blocked until it receives a reply from the provider of service. It is possible to have also asynchronous request messages, where the sender does not block after sending a request and the server does not send any reply back. Another communication style in LQN allows for a client request to be processed by a chain of servers instead of a single server, as shown in section 4. The first server in the chain will forward the request to the second, etc., and the last server will reply to the client. Although not explicitly illustrated in the LQN notation, every server, be it software or hardware, has an implicit message queue where incoming requests are waiting their turn to be served. Servers with more then one entry have a single input queue, where requests for different entries wait together.

A server entry may be decomposed in two or more sequential phases of service. Phase 1 is the portion of service during which the client is blocked, waiting for a reply from the server (it is assumed that the client has made a synchronous request). At the end of phase 1, the server will reply to the client, which will unblock and continue its execution. The remaining phases, if any, will be executed in parallel with the client. A recent extension to LQN [6] allows for an entry to be further decomposed into activities if more details are required to describe its execution. The activities are connected together to form a directed graph, which may branch into parallel threads of control, or may choose randomly between different branches. Just like phases, activities have execution time demands, and can make

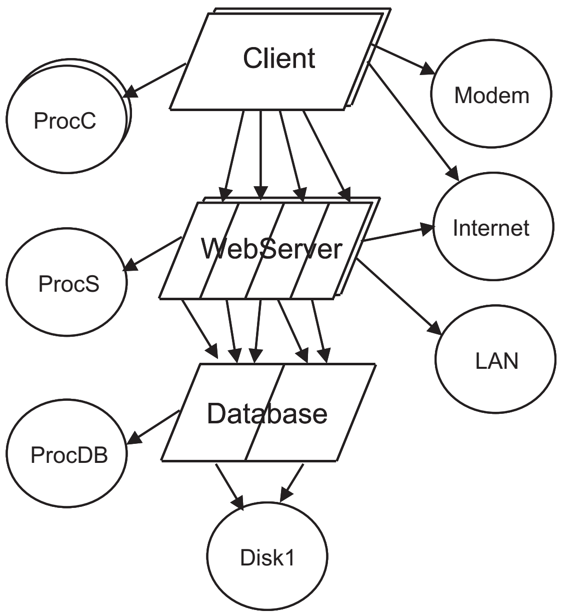


Figure 3: LQN model example

service requests to other tasks. Examples of LQN with activities are given in Fig. 8 and Fig. 10.

The parameters of a 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,

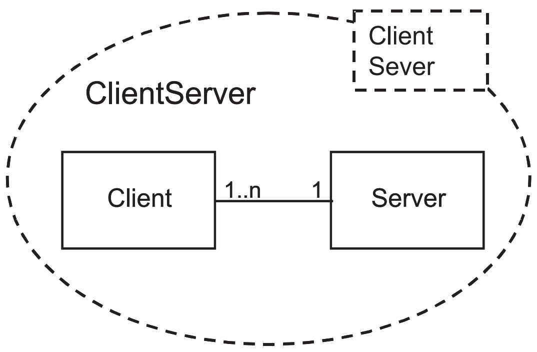
• for each phase (activity) making a request to another task entry: average number of visits,

• for each request arc: average communication delay,

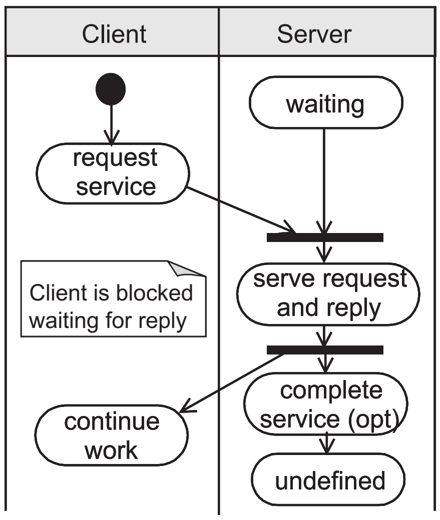
• for each software and hardware server: scheduling discipline.

2.3 Architectural Patterns

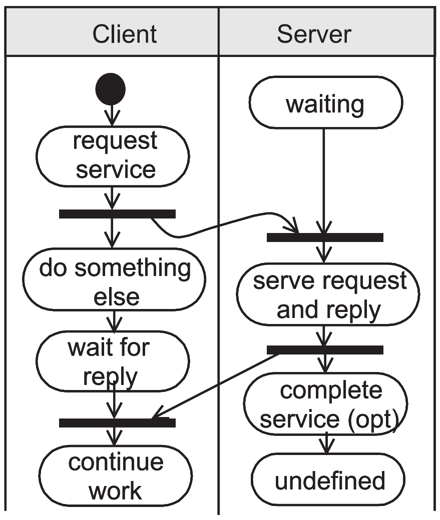
In our approach, the structure of the LQN model is generated from the high-level architecture, and more exactly from the architectural patterns used in the system. Frequently used architectural solutions are identified in literature as architectural patterns (such as pipeline and filters, client/server, broker, layers, master-slave, blackboard, etc.) A pattern introduces a higher-level of abstraction design artifact by describing a specific type of collaboration between a set of prototypical components playing well-defined roles, and helps our understanding of complex systems. Each architectural pattern describes two inter-related aspects: its structure (what are the components) and behaviour (how they interact). In the case of high-level architectural patterns, the components are usually concurrent entities that are executed in different threads of control, compete for resources, and their interaction may require some synchronization. The patterns are represented as UML collaborations [9].



a) ClientSever collaboration



b) ClientSever with a synchronous message

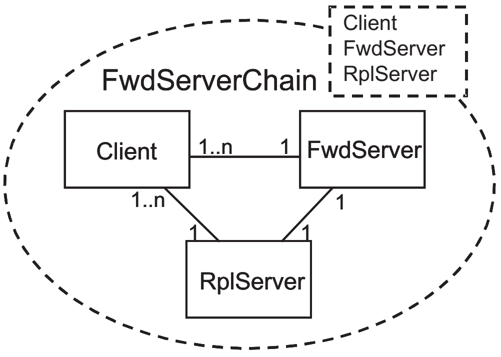


c) ClientSever with two asynchronous messages

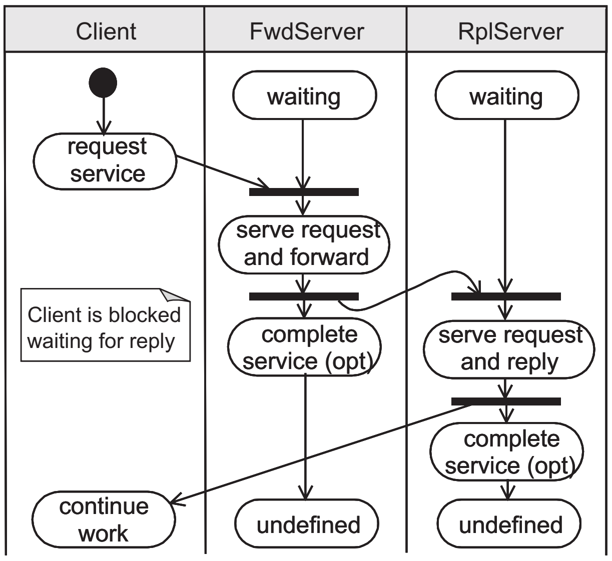
Figure 4: Client Server architectural pattern

The symbol for a collaboration is an ellipse with dashed lines that may have an "embedded" square showing the roles played by different pattern participants.

In Fig. 4 and Fig. 5 are shown the structure and behaviour of two patterns used in our case study: Client Server and Forwarding Server Chain. The Client Server pattern has two alternatives: the one shown in Fig. 4.b is using a synchronous communication style (where the client sends the request then remains blocked until the sender replies), whereas the one from Fig. 4.c is using an asynchronous communication style (where the client continues its work after sending the request, and will accept the server's replay later). The Forwarding Server Chain, shown in Fig. 5, is an extension of the Client Server pattern, where the client's request is served by a series of servers instead of a single one. There may be more than two servers in the chain (only two are shown in Fig. 5) . The servers in the middle play the role of ForwardingServer, as each one forwards the request to the next server in the chain after doing their part of service. The last server in the chain plays the role of ReplyingServer,



a) Forwarding Server Chain collaboration



b) Behaviour of the FwdServerChain pattern

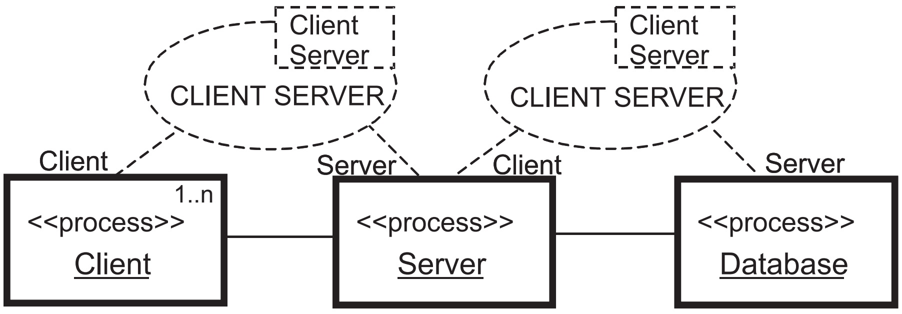
Figure 5: Forwarding Server Chain architectural pattern

as it sends the reply back to the client. In this paper we show how these two patterns are converted into LQN models. More architectural patterns and the corresponding rules for translating them into LQN are described by the authors of the present paper in [10, 11].

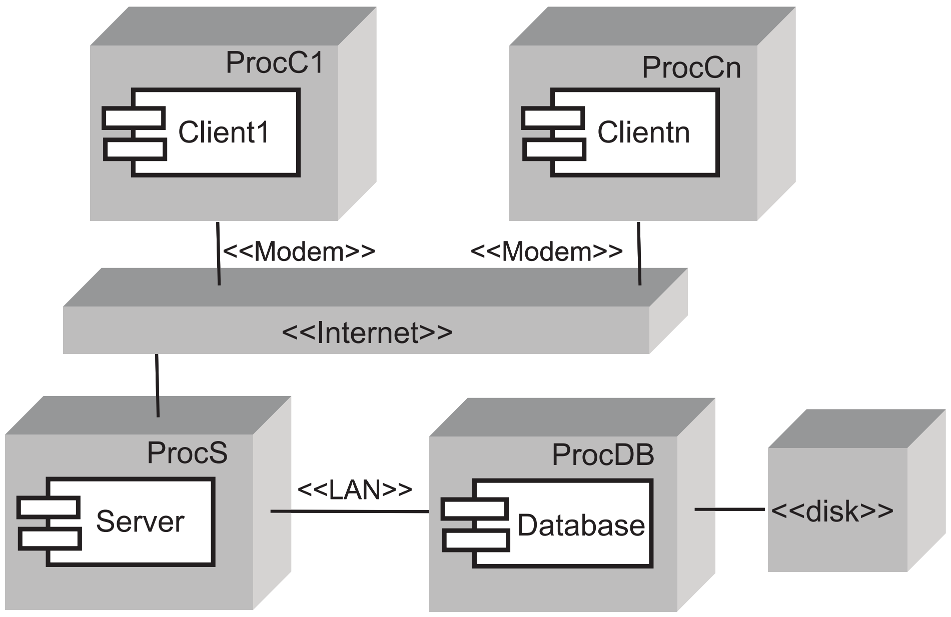
3 Transformation from UML to LQN

Similar to the SPE methodology from [16, 17], the starting point for our algo rithm is a set of key performance scenarios annotated with performance inform ation. For each scenario we derive a LQN submodel, then merge the submodels together. The approach for merging is similar with the one used in [6], where LQN submodels were derived from execution traces. The derivation of each LQN submodel is done in two big steps:

a) The submodel structure (i.e., the software and hardware tasks and their connecting arcs) is obtained from the high-level architecture of the UML model and from the deployment of software components to hardware devices. Two kinds of UML diagrams are taken into account in this step: a high-level collaboration diagram that shows the concurrent/distributed high-level component instances and the patterns in which they participate, and the deployment diagram. Fig. 6 shows these two diagrams for the web server model that was used to derive the LQN model from Fig. 3 (devices and tasks without entries.) Each high-level software component is mapped to a LQN software task, and each hardware device (processor, disk, communication network, etc.) is mapped to a LQN hardware task. The 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



a) High-level software architecture



b) Deployment diagram

Figure 6: UML diagrams used to generate the structure of the LQN model from Fig.3

of the architectural patterns; their behavioural aspect will be considered in the next step.

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

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

In the proposed approach, we decided to use activity diagrams for translation to LQN. The main reason is that sequence (collaboration) diagrams are not well defined in UML yet, as they are lacking convenient features for representing loops, branches and fork/join structures. Other authors who are building performance models from UML designs have pointed out this deficiency of the present UML standard, and are using instead extended sequence diagrams that look like the Message Sequence Chart standard (see [19] for a well known example). We did not take the approach of extending the sequence diagrams with the missing features because our algorithm takes in XML files generated by UML tools, and parses graphs of UML metaobjects. Our implementation is consistent with the present UML metamodel and XMI interface; moreover, it uses an open-source library, named Novosoft Metadata Framework and UML Library [23], which implements the standard UML metamodel as defined in [9]. Therefore, our choice was to use activity diagrams that are able to represent branch/merge, fork/join and activity composition without any extensions.

However, the activity diagrams have a disadvantage with respect to sequence (collaboration) diagrams: they do not show what objects are responsible for different actions. An attempt to counterbalance this weakness was the introduction of "swimlanes" (or partitions) in the UML standard. A swimlane contains actions that are performed by a certain instance or set of instances (for example, a swimlane can be associated to a whole department when modeling a work flow problem). In our approach, we associate a swimlane with each concurrent (distributed) component, which will be translated into a LQN task (see Figures 4, 5 and 9.c). Since many UML modellers prefer sequence diagrams for expressing the cooperation between objects, we proposed in [13] an algorithm based on graph transformations for converting automatically sequence diagrams into activity diagrams. The transformation associates a swimlane to all the objects that belong to a concurrent (distributed) component, and therefore adjusts the level of abstraction of the model to our needs.

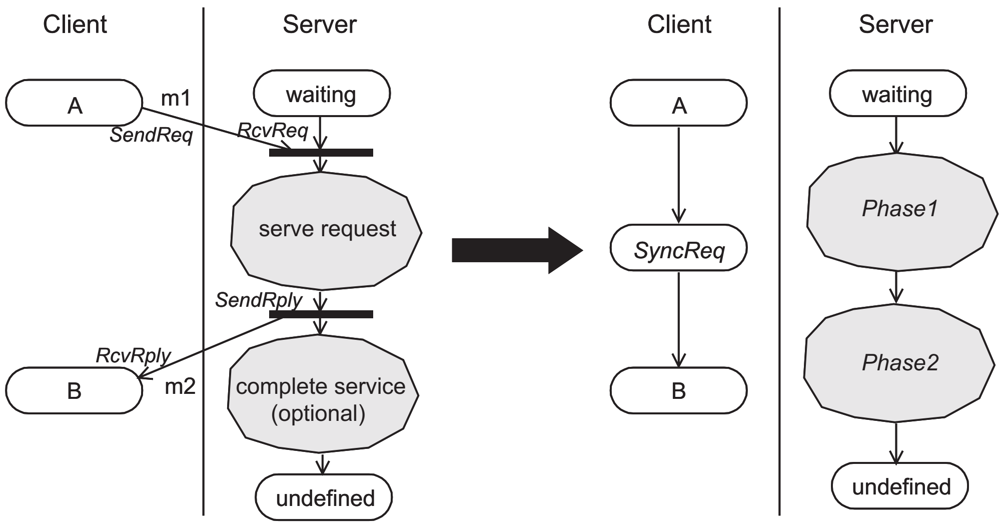
4 From Activity Diagrams to LQN entries, phases and activities

This section presents the graph grammar-based transformation of activity diagrams into LQN detailed features (i.e., the realization of step (b) from the previous section). The graph-grammar formalism is appropriate in this case because both UML and LQN models are described by graphs. The essential idea of all graph grammars or graph rewriting systems is that they are generalization of the string grammars that are used in compilers. The terms "graph grammars" and "graph rewriting systems" are often considered synonymous. However, a graph grammar is a set of production rules that generates a language of terminal graphs and produces nonterminal graphs as intermediate results, whereas 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 between terminals and nonterminals graphs. The main component of a graph grammar is a finite set of production rules. A production is a triple (L, R, E), where L and R are graphs (the left-hand side and right-hand side, respectively) and E is some embedding mechanism. Such a production rule can be applied to a host graph H as follows: when an occurrence of L is found in H, it is removed end replaced with a copy of R; finally, the embedding mechanism E is applied to attach R to the remainder of H [14].

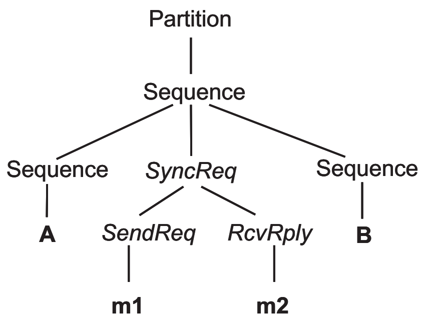
In our case, the initial host graph is the set of metaobjects that represents a given activity diagram (the metaobjects are the nodes and the links between them are the edges of the graph). According to the UML metamodel, the nodes (i.e., metaobjects) are of type `StateVertex` and `Transition`. A `StateVertex` type has a number of subtypes, among which `State` and `Pseudostate` are the most important. `State` is associated eventually with the actions represented in the diagram, whereas the special diagram blocks such as "fork", "join", "choice", etc., are `Pseudostates` (see [9] for more details).

Our purpose is to parse the activity diagram graph to check first whether it is correct, then to divide it into subgraphs that correspond to various LQN elements (entries, phases, etc.). In general, it is quite difficult to parse graph grammars, and in some cases even impossible [14]. In the case discussed in this paper we have found a shortcut by decomposing the original host graph into a set of subgraphs, each corresponding to a swimlane from the activity diagram. Each subgraph is described by a simpler graph-grammar, very similar to the string grammars used in compilers. This is not so surprising, since a swimlane describes the behaviour of a single component, dealing with sequences of scenario steps and nested structures such as loops, alternative branches and fork/joins. After disconecting the swimlanes as shown below, each swimlane subgraph has a structure known as an AND/OR graph. We defined a context-free grammar describing these subgraphs and implemented a top-down parser with recursive methods for parsing them. Our algorithm for the step (b) from the previous section contains two substeps:

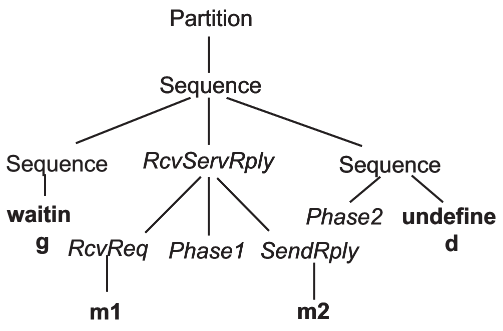
b.1) Overlay the behaviour of the architectural patterns extracted in step (a) from the high-level collaboration diagram over the activity diagram, and verify whether the communication between concurrent components is consistent with the pattern. This is done by traversing the graph, by identifying the cross-transitions between swimlanes,



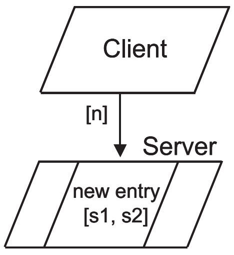
a) Graph transformation rule for the Client Server pattern with synchronous communication



b) Parse tree for the Client's partition



c) Parse tree for the Client's partition



d) Generated LQN elements

Figure 7: Transformation rule and parsing trees for the ClientServer pattern with synchronous communication

and by checking if they follow the protocol defined by the pattern. Then, apply a graph transformation rule corresponding to the respective pattern, and attach appropriate nonterminal symbols to the subgraphs identified (see Fig. 7 and Fig. 8). This disconnects practically the subgraphs corresponding to each swimlane from its neighbors. Repeat this step for all inter-component communication (we assume that all are covered by some architectural pattern).

b.2) Parse separately the subgraph within each swimlane. This will deal with sequences, loops, alternative branches and fork/join structures. When parsing a subgraph corresponding to a certain LQN element (phase or activity) identified in the previous step, compute its execution time S from the execution times of the contained scenario steps as follows: $S = \sum_{i=1}^{n} r_i s_i$, where r_i is the number of repetitions and s_i the host execution time of scenario step i.

The graph transformation rule and parsing tree for the ClientServer pattern with synchronous communication are illustrated in Fig. 7, whereas those for the variant with asynchronous communication are illustrated in Fig. 8. The right-hand side R of the rule from Fig. 7.a contains some nodes and edges carried over from the left-hand side L (which will be kept in the graph), and some new ones (which will be added). Also, some nodes and edges from L do not appear in R, meaning that they will be deleted from the graph. The embedding mechanism assumes that (i) all the nodes that are kept in R from L will keep also their links with the remaining of the graph, (ii) the links of the deleted nodes with the remaining of the graph will be deleted as well, and (iii) the new nodes are explicitly connected only to nodes from R as shown in R.

According to Fig. 7.a, two subgraphs are identified on the server side. One corresponds to the phase 1 of service and is found between the "join" Pseudostate marking the receiving of the client request and the "fork" Pseudostate marking the sending of the reply. The other subgraph corresponds to the second phase of service, is optional, and can found between the "fork" marking the sending of the reply and either a "waiting" state for a new request or the "undefined" state used by default to mark the end of the respective component behaviour on behalf of the current scenario. The subgraphs will be labeled with the nonterminal symbols *Phase1* and *Phase2*, respectively, and will be parsed after step (b.1). On the client side a new node `SyncRec`, which symbolizes the making of the request, will replace the transitions `m1` and `m2`, which represent the sending of the request and the receiving of the reply, respectively. The partial parsing trees for the client and server side are shown in Fig. 7.b and 7.c. It is important to emphasize that the semantic of the nonterminals in italic is related to the abstraction level represented by the high-level pattern. These nonteminals are used to identify and label the subgraphs that correspond to smaller LQN elements, such as phases and activities.

Fig. 7.d illustrates the LQN elements generated in this case. A new entry of the Server task is generated for each new type of request accepted by the server. In the case shown in the figure, the entry has two phases, and their service times are computed as explained in step (b.2). A LQN request arc is generated between the Clients phase (or activity) containing the new `SyncRec` node inserted by the rule from Fig. 7.a and the new server entry. Its visit ratio n is identical with the number of repetitions of state `A` that originates the synchronous request.

Fig. 8.a shows a similar transformation rule for the ClientServer pattern with asynchronous communication. The difference here is that the client does not block immediately after sending the reply to the server. A fork/join structure is inserted in the client side, as seen in Fig. 8.a and 8.b. The LQN elements generated are shown in Fig. 8.c. The new entry contains LQN activities and a similar fork/join structure. The transformation rules for other patterns are not given here due to space limitations.

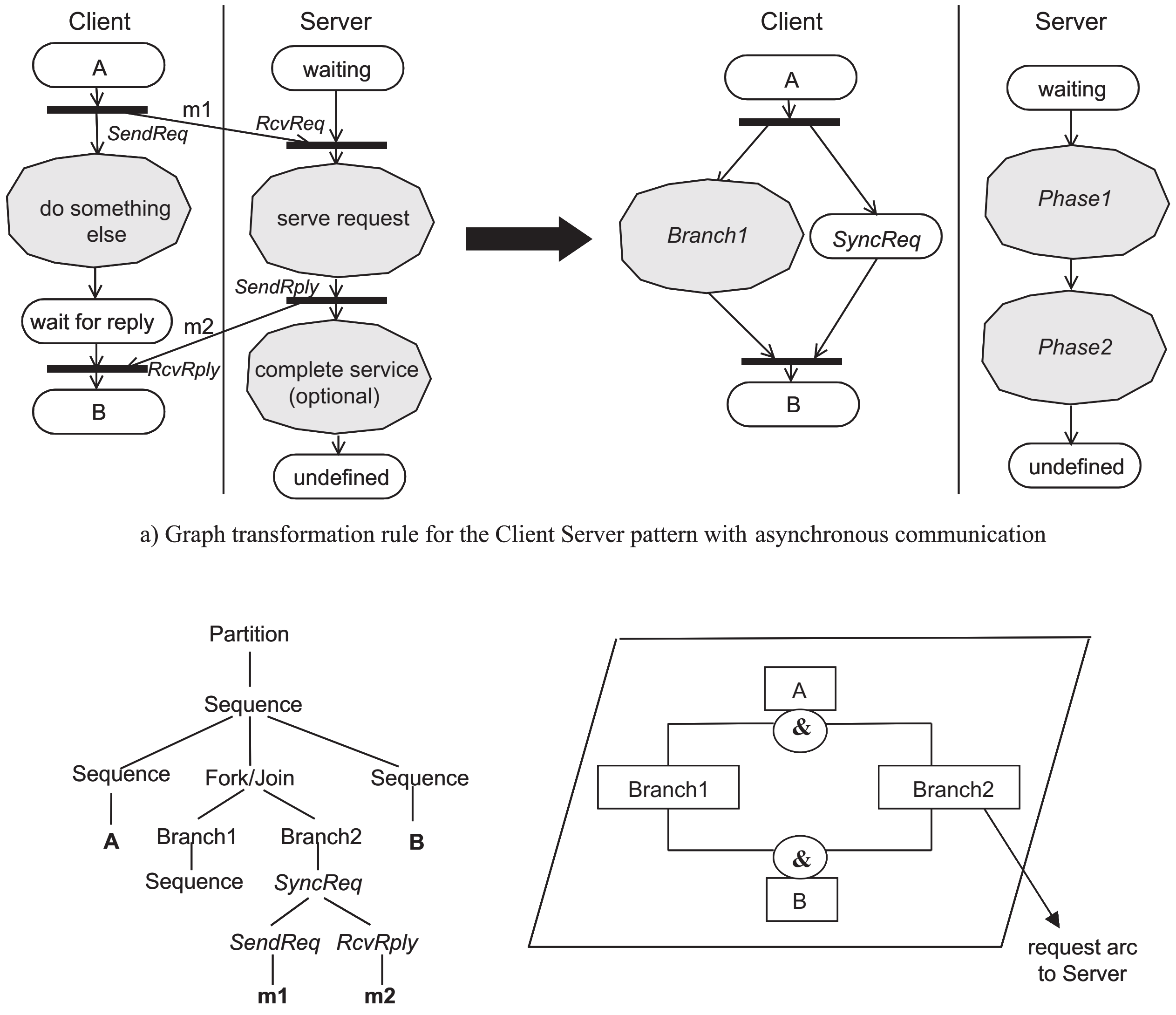
5 Case study: group communication server

In this section is presented the application of the proposed UML to LQN transformation algorithm to a case study [16]. A group communication server accepts two classes of documents from its subscribers, private and public, each with its own access rights. The documents are kept in two different files: the private documents on disk1 and the public ones on disk2. Five use cases were analyzed for the system: subscribe, unsubscribe, submit document, retrieve document and update document. However, only one scenario, namely Retrieve, is presented here. Fig. 9.a shows the architectural patterns in which are participating the components involved in this scenario, and Fig. 9.b gives the deployment diagram. The activity diagram with performance annotations describing the scenario is given in Fig. 9.c. A User process sends a request for a specific document to the Main process of the server, which determines the request type and forwards it to another process named RetrieveProc. This process is rctually esponsible for retrieving the documents. Half of all the requests will refer to documents found in the buffer, so no disk access is necessary. The other half is split as follows: 20% of the requests will refer to private documents, and 30% to public documents. In each case, RetrievProc delegates the responsibility of reading the document to the corresponding disk process.

Disk1Proc reads private documents from a sequential file, so it makes a number of F accesses to an external operation named "readDisk1" in order to find the desired document. On the other hand, Disk2Proc reads public documents from an indexed file, so it will make only one access to the external operation "readDisk2". The performance characteristics and the resources required for the external operations are described elsewhere. The external operations were specially provided in the UML performance profile to allow the UML modeler to describe the system at the right level of abstraction.

After getting the desired document either from file or from memory, RetrieveProc sends it back to the user. The scenario steps that send messages over the Internet invoke an external operation "network" once for every packet transmitted . This will allow to introduce in the performance model communication network delays that are not fully represented in the UML model.

Every scenario step in Fig. 9.c has a `PAdemand` tagged value indicating its estimated mean execution time on the host processor. The workload for the Retrieve scenario is closed, with a number of \$Nusers clients. A user "thinks" for a mean delay of 15s. (The identifiers starting with '\$' indicate variables that must be assigned concrete values before doing the

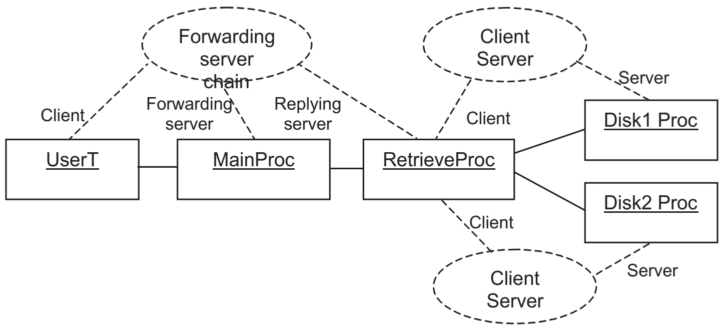


a) Graph transformation rule for the Client Server pattern with asynchronous communication

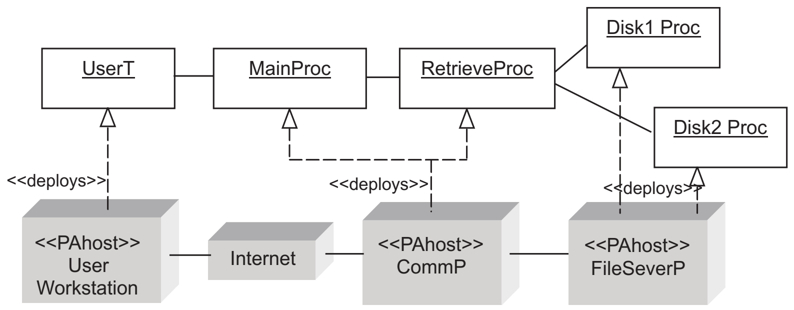
b) Parse tree for the Client's partition

c) LQN elements generated for the Client

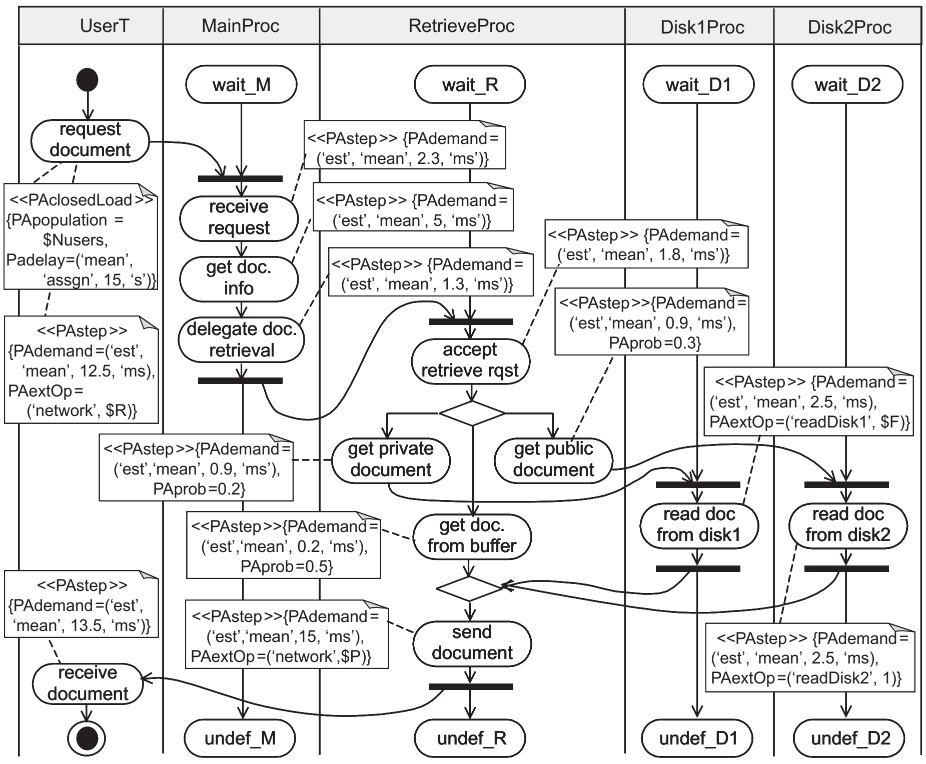
Figure 8: Transformation rule and parsing trees for the ClientServer pattern with asynchronous communication



a) High-level architecture for "retrieve" scenario



b) Deployment diagram for the components involved in "Retrieve" scenario


UserT
MainProc
RetrieveProc
Disk1Proc
Disk2Proc
wait_M
wait_R
wait_D1
wait_D2
request document
<<PAclosedLoad>> {PApopulation = $Nusers, Padelay=('mean', 'assgn', 15, 's')}
<<PAstep>> {PAdemand=('est', 'mean', 12.5, 'ms'), PAextOp= ('network', $R)}
receive request
get doc. info
delegate doc. retrieval
<<PAstep>> {PAdemand= ('est', 'mean', 2.3, 'ms')}
<<PAstep>> {PAdemand= ('est', 'mean', 5, 'ms')}
<<PAstep>> {PAdemand= ('est', 'mean', 1.3, 'ms')}
<<PAstep>> {PAdemand= ('est', 'mean', 1.8, 'ms')}
<<PAstep>>{PAdemand= ('est','mean', 0.9, 'ms'), PAprob=0.3}
accept retrieve rqst
get private document
get public document
<<PAstep>>{PAdemand= ('est','mean', 0.9, 'ms'), PAprob=0.2}
<<PAstep>> {PAdemand= ('est', 'mean', 2.5, 'ms), PAextOp=('readDisk1', $F)}
read doc from disk1
read doc from disk2
get doc. from buffer
<<PAstep>>{PAdemand= ('est','mean', 0.2, 'ms'), PAprob=0.5}
<<PAstep>> {PAdemand=('est', 'mean', 13.5, 'ms')}
<<PAstep>>{PAdemand= ('est','mean',15, 'ms'), PAextOp=('network',$P)}
send document
<<PAstep>> {PAdemand= ('est', 'mean', 2.5, 'ms'), PAextOp=('readDisk2', 1)}
receive document
undef_M
undef_R
undef_D1
undef_D2

c) Activity Diagram with performance annotations for "Retrieve" scenario

Figure 9: Group communication server: high-level architecture, deployment and activity diagram for "Retrieve" scenario

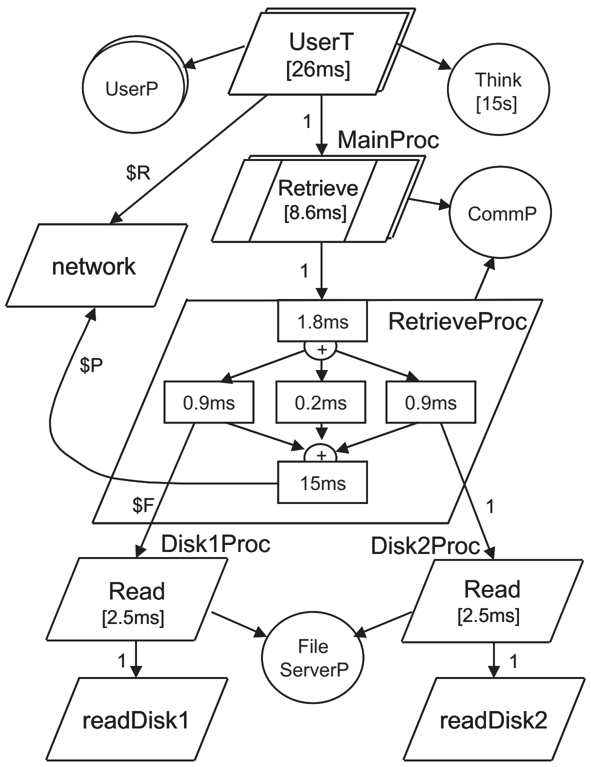


Figure 10: LQN submodel generated for the scenario "Retrieve"

actual performance analysis).

The LQN model obtained by applying our method is shown in Fig. 10. A LQN task was generated for each of the five software components from Fig. 9. Additional tasks were generated for the external operations `readDisk1`, `readDisk2` and `network`. The task `MainProc` has many entries, one for each type of requests it accepts. However, only one of its entries is determined from this scenario. This entry forwards the client request to `RetrieveProc` (forwarding is represented by a dotted request arc). `RetrieveProc` has one entry with internal branching, represented as a LQN activity graph that mirrors the scenario steps from the activity diagram from fig.9.c.

The purpose of this paper is to present the proposed UML to LQN transformation, so no performance analysis results are presented here. Our experience with the UML Performance Profile shows that it is relatively easy to understand, and that it provides enough performance annotations for generating working LQN models. However, we would like to add a few items on the "wish list" for the Performance Profile. Firstly, an additional tagged value is needed for expressing the size of messages, which is necessary for calculating the network delays. Secondly, it would be very useful to use parameterized expressions instead of concrete numbers for the tagged values. Thirdly, sometime it's useful to be able to define a workload over a set of scenarios that are composed in a certain way (sequentially, random choice, etc.)

6 Conclusions

Our experience with the graph-grammar formalism shows that it is very powerful and modularized by nature. The rule-based transformation approach lends itself rather easily to extensions. We are working right now on adding new transformation rules for other architectural patterns, such as pipeline and filters, critical section, blackboard, master-slave, etc. Another kind of extension we are planning on doing is the addition of a suitable GUI. An area that is completely uncovered is the backward path to bring back into the UML model the results from the performance analysis solver (represented with gray arrows in Fig. 2).

Regarding the inter-operability with UML tools, we have met with some problems due to the fact that the present UML tools do not support entirely the current UML standard. For example, Rational Rose does not support yet the following features: collaborations (i.e., the dashed ellipse symbol), object flow in activity diagrams and tagged values. Another tool we have been using, ArgoUML[22], does not support swimlanes and object flow in activity diagrams, etc. In order to test our algorithm, we have obtained XML files from the existing UML tools, but had to change them by hand in order to add the missing features. We hope that this problem will disappear with time, so that tool inter-operability will become a true reality.

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

[1] Amer, H., Petriu, D.C.: Software Performance Evaluation: Graph Grammar-based Transformation of UML Design Models into Performance Models. submitted for publication, 33 pages (2002)

[2] Balsamo, S., Simeoni, M.: On transforming UML models into performance models. In: Proc. of Workshop on Transformations in the Unified Modeling Language, Genova, Italy (2001)

[3] Cortellessa, V., Mirandola, R.: Deriving a Queueing Network based Performance Model from UML Diagrams. In: Proc. of 2nd ACM Workshop on Software and Performance, Ottawa, Canada (2000) 58–70

[4] Gomaa, H., Menasce, D.A.: Design and Performance Modeling of Component Interconnections Patterns for Distributed Software Architectures. In: Proc. of 2nd ACM Workshop on Software and Performance, Ottawa, Canada (2000) 117–126

[5] Franks, G., Hubbard, A., Majumdar, S., Petriu, D.C., Rolia, J., Woodside, C.M: A toolset for Performance Engineering and Software Design of Client-Server Systems. Performance Evaluation, Vol. 24, Nb. 1-2 (1995) 117–135

[6] Franks, G.: Performance Analysis of Distributed Server Systems. Report OCIEE-00-01, Ph.D. Thesis, Carleton University, Ottawa, Canada (2000)

[7] Hrischuk, C.E., Woodside, C.M., Rolia, J.A.: Trace-Based Load Characterization for Generating Software Performance Models. IEEE Trans. on Software Eng., V.25, No.1 (1999) 122–135

[8] Kähkipuro, P.: UML-Based Performance Modeling Framework for Component-Based Distributed Systems. In: R.Dumke et al.(eds): Performance Engineering, Lecture Notes in Computer Science, Vol. 2047. Springer-Verlag, Berlin Heidelberg New York (2001) 167–184

[9] Object Management Group: UML Specification Version 1.3. OMG Doc. ad/99-06-08 (1999)

[10] Object Management Group: UML Profile for Schedulability, Performance and Time. OMG Document ad/2001-06-14, `http://www.omg.org/cgi-bin/doc?ad/2001-06-14` (2001)

[11] Petriu, D.C., Wang, X.: From UML Description of High-Level Software Architecture to LQN performance models. In: Nagl, M., et al.(eds): Applications of Graph Transformations with Industrial Relevance AGTIVE'99. Lecture Notes in Computer Science, Vol. 1779. Springer-Verlag, Berlin Heidelberg New York (2000) 47–62

[12] Petriu, D.C., Shousha, C., Jalnapurkar, A.: Architecture-Based Performance Analysis Applied to a Telecommunication System. In: IEEE Transactions on Software Eng., Vol.26, No.11 (2000) 1049–1065

[13] Petriu, D.C., Sun, Y.: Consistent Behaviour Representation in Activity and Sequence Diagrams. In: Evans, A., et al.(eds): UML'2000 The Unified Modeling Language - Advancing the Standard. Lecture Notes in Computer Science, Vol. 1939 (2000) 369–382

[14] G.Rozenberg (ed): Hanbook of Graph Grammars and Computing by Graph Transformation, Vol.1. World Scientific (1997)

[15] Rolia, J.A., Sevcik, K.C.: The Method of Layers. EEE Trans. on Software Engineering, Vol. 21, Nb. 8 (1995) 689–700

[16] Scratchley, W.C.: Evaluation and Diagnosis of Concurrency Architectures. Ph.D Thesis, Carleton University, Dept. of Systems and Computer Eng. (2000)

[17] Schürr, A.: Programmed Graph Replacement Systems. In G.Rozenberg (ed): Handbook of Graph Grammars and Computing by Graph Transformation. (1997) 479–546

[18] Smith, C.U.: Performance Engineering of Software Systems. Addison Wesley (1990)

[19] Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison Wesley (2001)

[20] Woodside, C.M.: Throughput Calculation for Basic Stochastic Rendezvous Networks. In: Performance Evaluation, Vol.9, No.2 (1998) 143–160

[21] Woodside, C.M., Neilson, J.E., Petriu, D.C., Majumdar, S.: The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software. In: IEEE Transactions on Computers, Vol.44, Nb.1 (1995) 20-34

[22] ArgoUML tool. to be found at `http://argouml.tigris.org/`

[23] Novosoft Metadata Framework and UML Library, open source library to be found at `http://nsuml.sourceforge.net/`