Deriving Software Performance Models from Architectural Patterns by Graph Transformations

Dorina C. Petriu and Xin Wang

Carleton University
Ottawa, Canada, K1B 4G2
e-mail: {petriu|xinw}@sce.carleton.ca

Abstract: The paper proposes a formal approach to building software performance models for distributed and/or concurrent software systems from a description of the system's architecture by using graph transformations. The performance model is based on the *Layered Queueing Network* (LQN) formalism, which is an extension of the well-known Queueing Network modelling technique. The transformation from the architectural description of a given system to its LQN model is based on PROGRES, a known visual language and environment for programming with graph rewriting systems [9-11].

1. Introduction

It is generally accepted that performance characteristics, such as response time and throughput, play an important role in defining the quality of software products. In order to meet the performance requirements of such systems, the software developers should be able to assess and understand the effect of various design decisions on system performance at an early stage, when changes can be made easily and effectively. *Software Performance Engineering* (SPE) is a technique introduced in [14] that strongly recommends the integration of performance analysis into the software development process, starting from the earliest stages and continuing throughout the whole life cycle. According to SPE, performance must be designed and built into a system from the beginning, as opposed to the more frequent practical approach that postpones the performance concerns until the system is completely implemented, then tries to "fix" its the performance problems at this late stage. Late fixes tend to be very expensive and inefficient; in some cases, the product will never meet its original performance requirements.

The application of SPE to software development requires predictive modelling methods and tools to support performance oriented design and implementation decisions. The *Layered Queueing Network (LQN)* is such a modelling method [15, 16, 8] and toolset [5] that was developed to capture the characteristics of concurrent and/or distributed software systems. Some LQN components represent software processes, others hardware devices. LQN determines the delays due to contention, synchronization and serialization at both the software and hardware levels (see section 2 for a more detailed description).

LQN was applied to a number of concrete industrial systems (such as database applications, telecommunication systems, web servers, etc.) and it was proven useful for providing insights into performance limitations at both software and hardware levels, for giving feedback for performance improvements to the software development team, and for supporting capacity planning under various workloads. However, the integration of performance modelling techniques, such as LQN, into the software development process as promoted by SPE is hampered by a number of factors:

- The existence of a cognitive gap between the software and the performance domains. Software developers are concerned with designing, implementing and testing the software, but they want to be spared from learning performance modelling methods and from building and solving performance models. The development teams depend usually on specialized performance groups to do the performance evaluation work, but this leads to additional communication delays, inconsistencies between design and model versions and late feedback.
- Economical pressure for "shorter times to market" leads to shorter software life cycles. There is no time left for SPE, which traditionally implies "manual" construction of the performance models.

The present paper attempts to close the gap between performance analysis and software development by proposing a formal technique based on graph transformations to build the performance model of a system from a description of the software architecture (*i.e.,* the system's high-level components and their interconnections).

By implementing the proposed technique in a tool, two goals can be achieved: "instant" building of the performance model shortens the time required for SPE and makes it possible to keep the model consistent with the changes in the architecture.

The process of building a LQN performance model has two stages: a) generating the *model structure* from the software architecture description and b) computing the model *parameters*. These are quantitative values representing resource demands and numbers of visits, as for example "how much CPU processing time is required for a given software process to execute a given request" and "how many visits to various servers are necessary to complete that request". The objective of this paper is to propose a method that automates the first stage of the modelling process. Research work on the second stage is currently under way, aiming to automate the derivation of model parameters from the internal structure and behaviour of the high-level architectural components.

The paper is organized as follows: a short description of the LQN model is given in section 2, a discussion of the software architecture is presented in section 3, the PROGRES graph schema for the architectural views and LQN models is defined in section 4, and the principles for graph transformations and rewriting productions are explained in section 5.

2. LQN Model

LQN was developed as an extension of the well-known Queueing Network (QN) model, at first independently in [15, 16] and [8], then as a joint effort [5]. The LQN toolset presented in [5] includes both simulation as well as analytical solvers that merge the best previous approaches. The main difference with respect to QN is that in LQN a server to which customer requests are arriving and queueing for service, may become a client to other servers from which it requires nested services while serving its own clients. An LQN model is represented as an acyclic graph whose nodes are software entities and hardware devices, and whose arcs denote service requests. The software entities (named also *tasks*) are drawn as parallelograms, and the hardware devices as circles. The nodes with outgoing and no incoming arcs play the role of *clients*, the intermediate nodes with both incoming and outgoing arcs are usually software servers and the leaf nodes are hardware servers (such as processors, I/O devices, communication network, etc.) Figure 1 shows a simple example of an LQN model of a three-tiered client/server system: at the top there are two classes with a given number of stochastic identical clients. Each clients sends demands for a specific service offered by a task named Application, which represents the business layer of the system. Each kind of service offered by an LQN task is modelled as a so-called *entry*, drawn as a parallelogram "slice". Each entry has its own execution times and demands for other services (given as model parameters). In this case, each Application entry requires services from two different entries of the Database task, which offers three kinds of services. Each software task is running on a processor that is shown as a circle; in the example, all clients of the same class share a processor, whereas Application and Database share another processor. Database uses also two disk devices as shown in Figure 1. It is worth mentioning that the word *layered* in the name LQN does not imply a strict layering of the 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 doesn't block after sending a request and the server doesn't send any reply back, as shown in the following sections. Although not explicitly illustrated in the LQN notation, each software and hardware server has an implicit message queue where the incoming requests are waiting their turn to be served. Servers with more then one entry still have a single input queue, where requests for different entries wait together. The default scheduling policy of the queue is FIFO, but other policies are also supported.

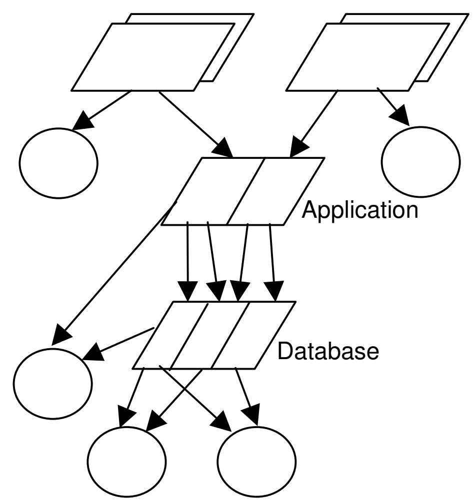

Figure 1. Simple example of an LQN model

Typical results of an LQN model are response times, throughputs, server utilization on behalf of different types of requests, queueing delays. The LQN results may be used to identify the software and/or hardware components that limit the system performance under different workloads and resource allocations. Understanding the cause for performance limitations helps the development team to come up with appropriate remedies.

3. Software Architecture Descriptions

The emerging discipline of software architectures is concerned with informal and formal ways of describing the overall system structure of complex software systems. In [12] a perspective on this new discipline is presented, in [13] and [4] a number of high-level architectural patterns frequently used in today's software systems are identified and described, and in [3] a formal foundation for software architectures based on architectural connections is introduced. Graph grammars were used in [6] to describe the evolution of dynamic architectures.

We are using in this paper the informal approach of architectural patterns from [13] and [4] to explain the principles proposed for translating an architectural description into an LQN performance model. We will use then some of the background on architectural connection from [3] to define the PROGRES schema and transformation rules for our graph rewriting approach.

There are a relatively small number of patterns identified in literature that are used to describe the high-level architecture of a large range of software systems. These patterns describe the collaboration between concurrent components, which can run on a single computer or in a distributed environment. We have selected three architectural patterns as a basis for our discussion: Pipe and Filters (PF), Client-Server (CS) and BlackBoard architecture (BB). These patterns are frequently used to build distributed systems, and they present a variety of interactions between components.

PF divides the overall processing task into a number of sequential steps which are implemented as filters, while the data between filters flows through unidirectional pipes. Interesting performance problems arise in the case of active filters [4] that are running concurrently. Each filter is implemented as a process or thread that loops through the following steps: "pulls" the data (if any) from the preceding pipe, processes it and then "pushes" the results down the pipeline. The way in which the push and pull operations are implemented may also have performance consequences. Two cases are shown in Figure 2: a) the filters communicate through asynchronous messages, and b) the filters communicate through a shared buffer (one pushes and the other pulls). The small arrows attached to the communication arcs indicate the data flow.

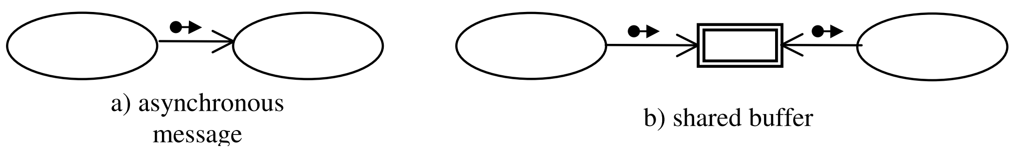

Figure 2. Communication in Pipeline and Filters Pattern

The Client-Server pattern is one of the most frequently used in today's distributed systems, especially since the introduction of new midware technology such as CORBA [7], which facilitates the connection between clients and servers running on heterogeneous platforms across local or wide-area networks. Since the communication between the architectural components has a crucial effect on performance, we will consider two different cases: direct client/server communication through a synchronous message or connection through a CORBA interface where the broker is the intermediary between clients and servers. In the first case shown in Figure 3a, the client

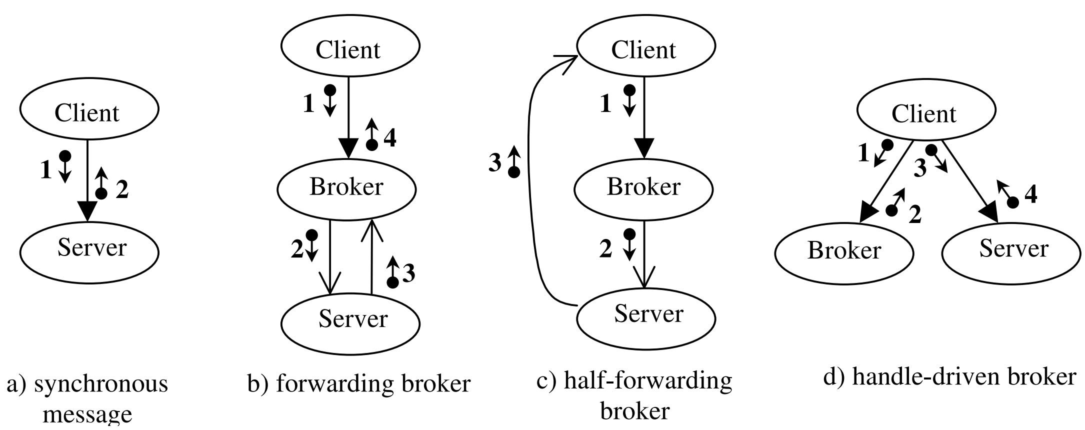

Figure 3. Communication styles in the Client Server pattern

sends a synchronous request to the server, then blocks and waits for the server's reply. Although the direction of the synchronous message is from the client to the server, the data flow is bidirectional (the request goes one way and the reply comes the other way). In the case of a CORBA interface, we distinguish several types of client/server connections [2]. In the *forwarding broker* pattern from Figure 3b, the broker relays a client's request to the relevant server, retrieves the response from the server and relays it back to the client. The forwarding broker is at the center of all communication paths between clients and servers, and can provide load balancing or restart centrally any failed transactions. However, there is a price to pay in terms of performance: an interaction between a client and a server requires four messages, which leads to an excessive network traffic when the client, broker and server reside on different nodes. An alternative that reduces the excessive network traffic of the forwarding broker is the *half-forwarding broker* from Figure 3c, where the server returns the reply directly to the client. This reduces the number of messages for a client/server interaction to three, while it retains the main advantages of the forwarding broker (load balancing and centralized recovery from failure). A *handle-driven* broker (as in Figure 3d) returns to the client a handle containing all the information required to communicate directly with the server. The client may use this handle to talk directly to the server many times, thus reducing the potential for performance degradation. However, the client takes on additional responsibilities, such as checking if the handle is still valid after a while, and recovering from failures. Load balancing is also more difficult

The Blackboard pattern is composed of a number of processes that share a common knowledge base stored in shared memory (see Figure 4.) In order to insure the correctness of the common data, the access must be controlled by semaphores, locks or other similar mechanisms. The serialization brings performance effects, and must be captured in a performance model.

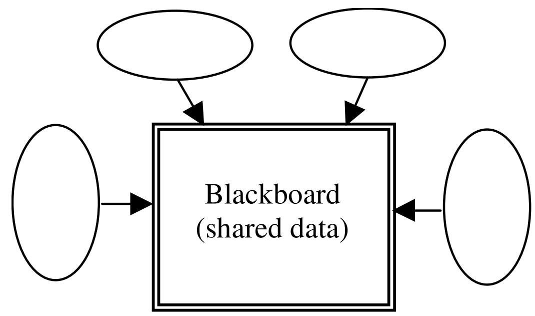

Figure 4. Blackboard pattern

After the informal presentation of the chosen architectural patterns and of their performance implications, we will review briefly the formal approach to architectural connections introduced in [3], which is the basis for the graph grammar representation of software architectures proposed in the next section. According to [3], software architecture can be defined as a collection of computational *components* together with a collection of *connectors*, which describe the interactions between components. A *component type* is described as a set of *ports* and a *specification* that describes its function. Each port defines a logical point of interaction between the component and its environment. A *connector type* is defined by a set of *roles* and a *glue* specification. The roles describe the expected behaviour of the interacting parties, and the glue shows how the interactions are coordinated. The connector specification is formally described in [3] with a subset of Hoare's process algebra.

For example, in a CS pattern with CORBA interface, the connector type is defined by three roles (client, server and broker) and by the glue that shows what kind of interactions take place between participants, and in which order. Since the three kinds of brokers shown in Figures 3b, 3c and 3d behave and interact differently with the client and the server parties, each one corresponds to a different connector type. Figure 2b illustrates another example of connector type that contains three roles (two filters and a shared buffer). Its glue describes the "push" and "pull" operations and the constraints for correct behaviour (as for example "cannot pull data from an empty buffer", "cannot read and write to the buffer at the same time", etc.)

In our performance-oriented work, we identified first the connector types associated to the architectural patterns, then defined graph transformation rules to translate each connector to an LQN model.

4. Graph Schema

The graph schema defined according to the PROGRES language [9 to 11] is presented in Figure 5. The upper part of the figure contains the *input schema* for architectural descriptions and the lower part the *output schema* for LQN models (light-gray nodes). The two schemas are joined together by three nodes (shown in dark-gray) at the base of the node class hierarchy in order to accommodate graphs in intermediary translation stages.

The input schema describes two kinds of software components and their connections: "process" (active component with its own thread of control) and "shared-memory" (passive component of either "buffer" or "blackboard" types). Each type of component has different types of ports corresponding to the roles played in various architectural connections. The edge types in the graph also correspond to different connection types. An interesting example is that of the four Client Server connection types, which are differentiated in the architectural view only by their different edge type (*cs_d*, *cs_fwB, cs_hfB* and *cs_hdB*, respectively).

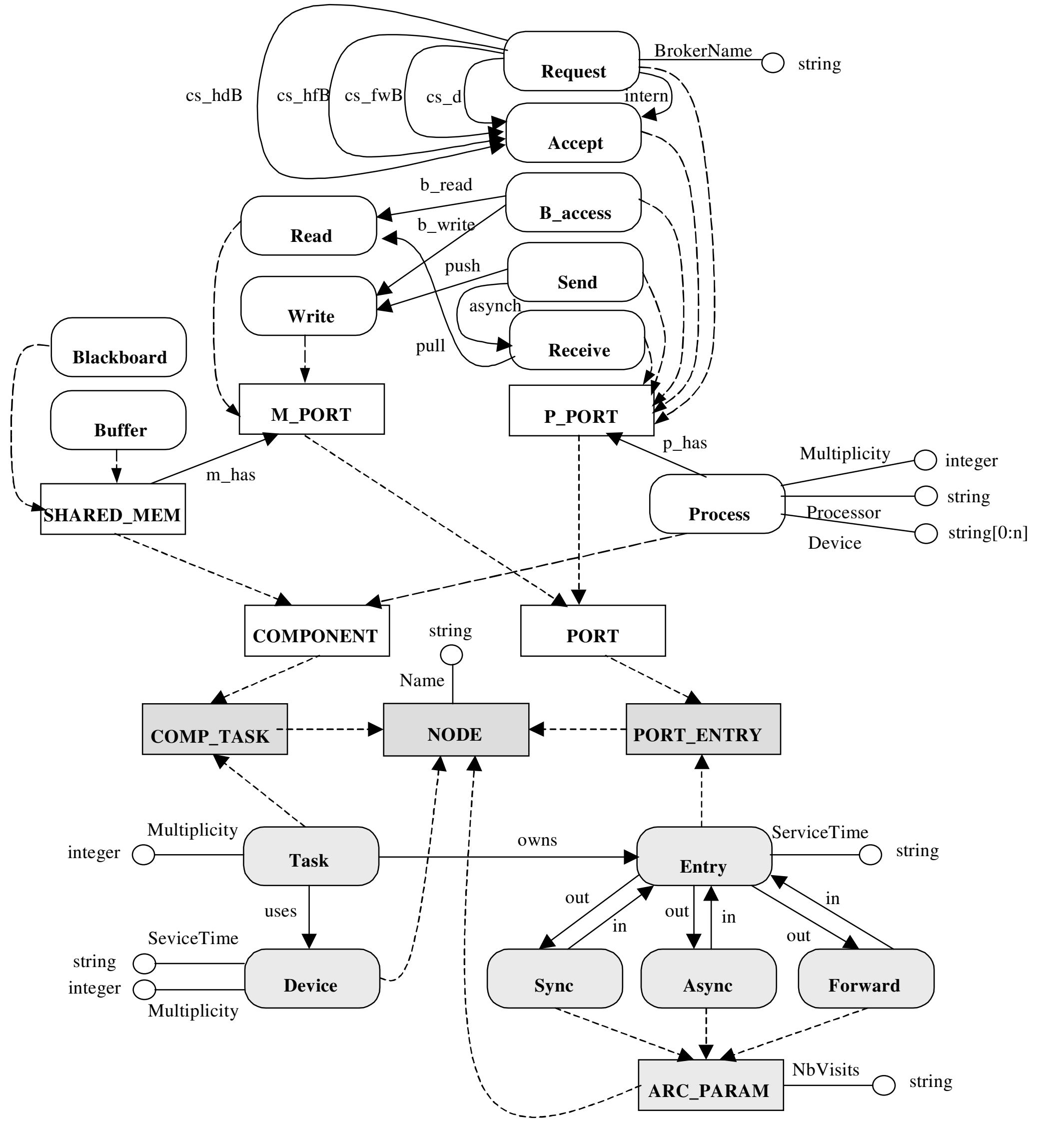

Figure 5. The joint graph schema of the architectural description and the LQN model

It is interesting to note that the "broker" component is not explicitly shown in the architectural view (as the broker is not actually part of the software application, but is provided by the underlying midware). However, as the broker has an important impact on system performance, it is explicitly represented in the LQN model.

The LQN graph notation presented in section 2 and illustrated in Figure 1 has task and device nodes, which are described by corresponding node types in the output schema. As the LQN tasks contain entries, an entry type was also added to the schema. The LQN arcs may represent three types of requests (synchronous, asynchronous and forwarding) and each has a parameter indicating the number of visits associated with that request. As the PROGRES edges cannot have attributes, we represented an LQN arc by three elements: an incoming edge, a node carrying the parameter and an outgoing edge.

5. Graph Transformations

We defined transformations for each architectural connection type into an LQN model, as illustrated in Figures 6 to 10 for a few selected cases. In order to convey the principle of these transformations, we used here a more intuitive graph notation for architectures and LQN models, rather than the more detailed PROGRES notation.

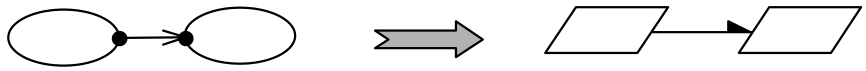

Figure 6. Translation of a PF connection by asynchronous message to an LQN model with asynchronous request

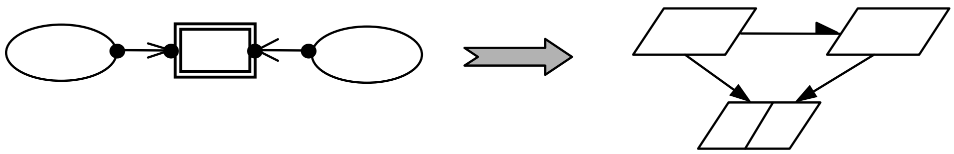

Figure 7. Translation of a PF connection by shared buffer to an LQN model with asynchronous request and semaphore task

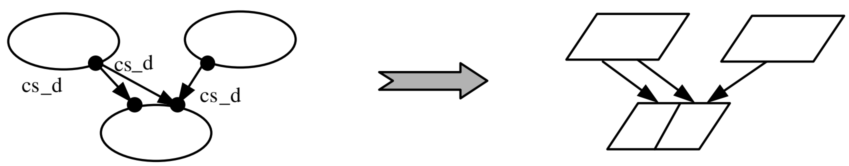

Figure 8. Translation of three direct CS connection instances to an LQN model (each service offered by the server is represented by an entry)

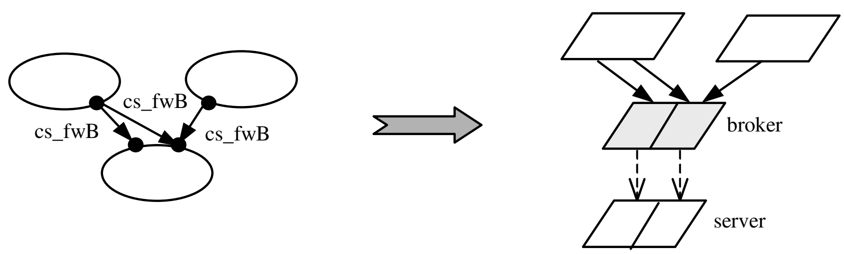

Figure 9. Translation of three CS connection instances by forwarding broker to an LQN model with forwarding arcs

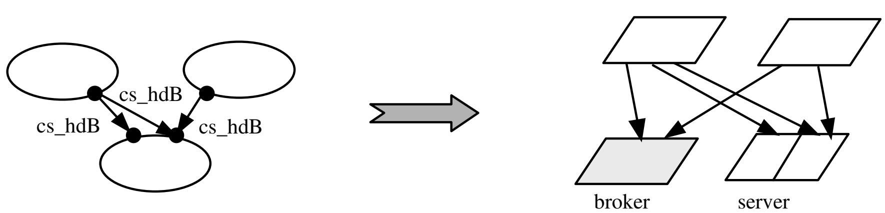

Figure 10. Translation of three CS connection instances by handle-driven broker to an LQN model

A software system contains many components involved in various architectural connection instances, and a component (a process) may play different roles in connections of various types. Such a process must own an appropriate port for each of the roles it plays, whereas a port can participate in more than one connection instance of the same type.

The PROGRES transformation process executes a transaction for every connection instance found in the input architectural description graph. The translation process ends when all architectural connections have been processed. The final result is an LQN model that can be written to a file according to a predefined LQN model format [5].

The following translation approach was followed:

- Each architectural component is converted to an LQN task, for which reason a common base class COMP_TASK was defined in the graph schema for components and tasks. However, the correspondence between components and tasks is not bijective as it may seem at first, due to processes implemented in the underlying operating system or midware (such as brokers) which are not represented explicitly in the architectural view, but are explicit in the LQN view.
- Each input port of a component is converted into an LQN entry. The correspondence between input ports and entries is not bijective either, due to broker entries.
- The output ports do not have any correspondent in LQN. However, they play a role in the translation process, as illustrated in Figure 11.
- Processes and devices, which are attributes in the architectural view, become full-fledged nodes in LQN (not illustrated in the paper). This happens because the issue of resource allocation is not central to the software development process, but has to be specified before doing performance analysis.

Figure 6 and 7 show the translation of two Pipe and Filters connection types, one using asynchronous messages and the other a shared buffer. The translation is quite straightforward, only the way LQN deals with a passive shared buffer warrants a little discussion. The pipeline connection is represented by an asynchronous LQN arc, but this does not take into account the serialization delay due to the constraint that buffer operations must be mutually exclusive. A third task is introduced, with as many entries as the number of critical sections in the tasks accessing the buffer. A further complication appears when these tasks execute on different processors, leading to a different transformation, which is not discussed here due to space limitations.

Figures 8, 9 and 10 illustrate the transformation for three Client Server connections that have similar architectural descriptions, differentiated only by the edge type. However, their LQN models are quite different, as the connections have very different operating modes and performance characteristics. The LQN model for the direct client-server connection is quite straightforward, but those for broker connections are more interesting. The forwarding broker model uses LQN forwarding requests (drawn with dotted lines) with a special semantic. After accepting a request from a client, the acceptor task will do some processing, then may decide to forward the request to another task. The forwarder is free to continue its activity, while the client remains blocked, waiting for a reply. The second task that continues to serve the request may eventually complete it and send the reply to the client, or may decide to forward the request to another task. The LQN model implies that a reply

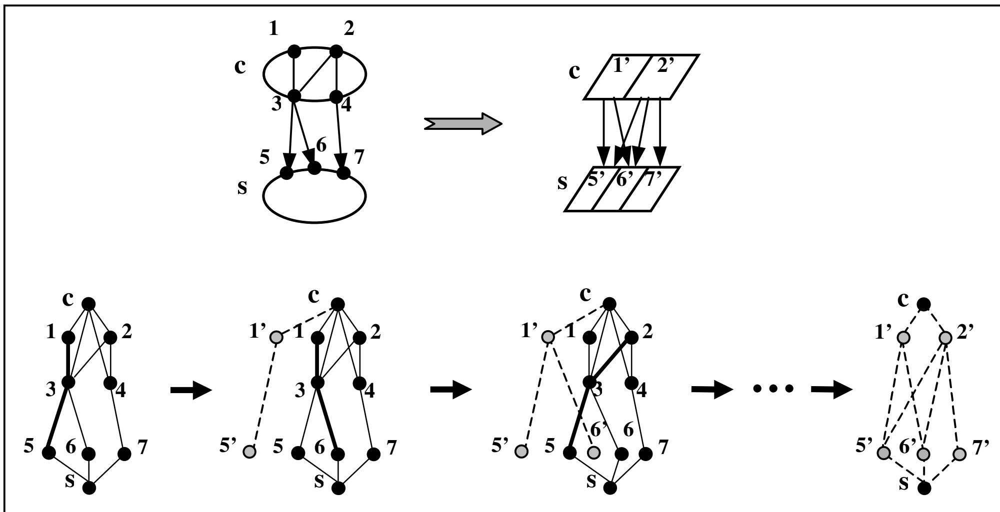

Figure 11. Sequence of graph transformations for the translation of layered servers from the architectural to the LQN view

will be sent to the client by the last task in the forwarding chain, but it does not represent this reply by an arrow. In Figure 9, the broker is the task that receives the requests from the clients and forwards them to the appropriate entry of the server. The broker must have a separate entry for each entry it forwards to, otherwise the clients would be unable to choose the server entry they need. The LQN model for the handle-driven broker sends two separate requests, one to the broker for getting the handle, then another to the desired server entry directly.

Figure 11 illustrates the handling of servers in layered (also known as "tiered") client/server architectures, where a server may require the support of another server while serving its own clients (the second server's work gets nested in the first one's service). Such a server component is involved in a number of client/server connections, playing the role of server in ones and the role of client in others. It owns both input ports (corresponding to the server role) and output ports (corresponding to the client role). The input and output ports are linked through edges of "intern" type, representing the association between client and server roles (as shown in Figure 11). The upper part of the figure shows the translation from architectural description to LQN for a subsystem with two servers involved in several client/server connections, and the lower part gives a sequence of PROGRES derivations for the same subsystem. In the later, the black nodes and full edges have types defined in the input schema, while the gray nodes and dotted arcs correspond to types from the output schema. (It is easy to see that the PROGRES notation is more detailed and less intuitive than the other one). Each derivation deals with a two-edge link (shown in thick lines) from an input port of the upper server to an input port of the lower server. There are five such links in our example, each one generating an LQN request arc.

6. Conclusion

The automatic generation of LQN performance models from software architecture descriptions raises a number of challenges due to the fact the two views have different semantic, purpose and focus that must be bridged by the translation process. Whereas the architectural view is mostly concerned with representing the components of the application under development, it uses any number of operating system and midware services that may have a strong impact on performance, and must be explicitly modelled in LQN. Another discrepancy between the two views arises from the different role played by the issue of resource allocation. The graph rewriting formalisms has proven very useful in dealing with these challenges.

References

[1] O. Adebayo, J. Neilson, D. Petriu, "A Performance Study of Client-Broker-Server Systems", in Proceedings of CASCON'97, pp 116-130, Toronto, Canada, November 1997.

[2] R. Adler, "Distributed Coordination Models for Client/Server Computing", IEEE Computer, pp 14-22, April 1995.

[3] R.Allen, D. Garlan, "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering Methodology*, Vol.6, No.3, pp 213-249, July 1997.

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

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

[6] D. Le Metayer, "Software Architecture Styles as Graph Grammars", in Proceedings of the 4-th ACM SIGSOFT Symposium on the Foundations of Software Engineering, San Francisco, USA, pp 15-23, Oct. 1996.

[7] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Object Management Group and X/Open, Framingham, MA and Reading Berkshire UK, 1992.

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

[9] A. Schuerr, "PROGRES: A Visual Language and Environment for PROgramming with Graph Rewrite Systems", Technical Report AIB 94-11, RWTH Aachen, Germany, 1994.

[10] A. Schuerr, "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, 1990.

[11] A. Schuerr, "Programmed Graph Replacement Systems", in Handbook of Graph Grammars and Computing by Graph Transformation, G. Rozenberg (ed), pp 479-546, 1997.

[12] M. Shaw, D. Garlan, *Software Architectures: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[13] M. Shaw, "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, 1996.

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

[15] C.M. Woodside, "Throughput Calculation for Basic Stochastic Rendezvous Networks", Performance Evaluation, Vol. 9, Number 2, pp.143-160, April 1995.

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