Deriving Software Performance Models from Architectural Patterns by Graph Transformations

Dorina C. Petriu and Xin Wang

Carleton University
Ottawa, ON, Canada, K1S 5B6
{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, an extension of the well-known Queueing Network modelling technique [16, 17, 8]. 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]. The transformation result is an LQN model that can be analysed with existent solvers [5].

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 [15] 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 the 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

[0] Research partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by the Communications and Information Technology Ontario (CITO).

decisions. The *Layered Queueing Network (LQN)* is such a modelling method [16, 17, 8] and toolset [5] that was developed to capture the characteristics of concurrent and/or distributed software systems. 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, web servers, telecommunication systems [14]) 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. One is 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. Also, economical pressure for "shorter time 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 the description of the high-level 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 in average for a given software process to execute a given request" and "how many visits (in average) 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 high-level architectural patterns and the approach to translate them to LQN models is presented in section 3, the PROGRES graph schema and examples of rewriting rules are discussed in section 4, and conclusions in section 5.

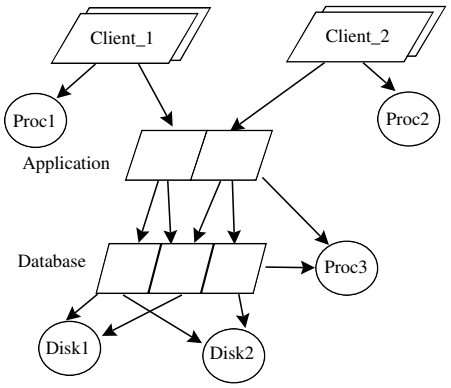

Fig. 1. Simple example of an LQN model

2 LQN Model

LQN was developed as an extension of the well-known Queueing Network (QN) model, at first independently in [16, 17] 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 incoming and outgoing arcs play both the role of client and that of server, and represent usually software servers. The leaf nodes are pure servers, and model usually hardware servers (such as processors, I/O devices, communication network, etc.) Fig. 1 shows a simple example of an LQN model for a three-tiered client/server system: at the top there are two classes with a given number of stochastic identical clients. Each client sends requests 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 by a so-called *entry*, drawn as a parallelogram "slice". An entry has its own execution times and demands for other services (given as model parameters). In this case, each Application entry requires two different services from the Database task. Each software task is running on a processor 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 Fig. 1. It is worth mentioning that the word layered in the name LQN does not imply a strict layering of the tasks (for example, a task may call each other tasks in the same layer, 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. Typical results of an LQN model are response times, throughput, utilization of servers on behalf of different types of requests, and 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 Approach for the Transformation from Software Architecture Descriptions into LQN models

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. In this paper, we are using the informal approach of architectural patterns from [13] and [4] to explain the principles for translating an architectural description into an LQN model. We will use then some of the background on architectural connections from [3] to present our graph rewriting approach.

A relatively small number of patterns that describe the high-level architecture of a large range of software systems are identified in literature. 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 (BB). These are frequently used to build distributed systems and present a variety of interactions between components.

PF divides the overall processing task into a number of sequential steps that 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 the left side of Fig. 3: in (*a*)

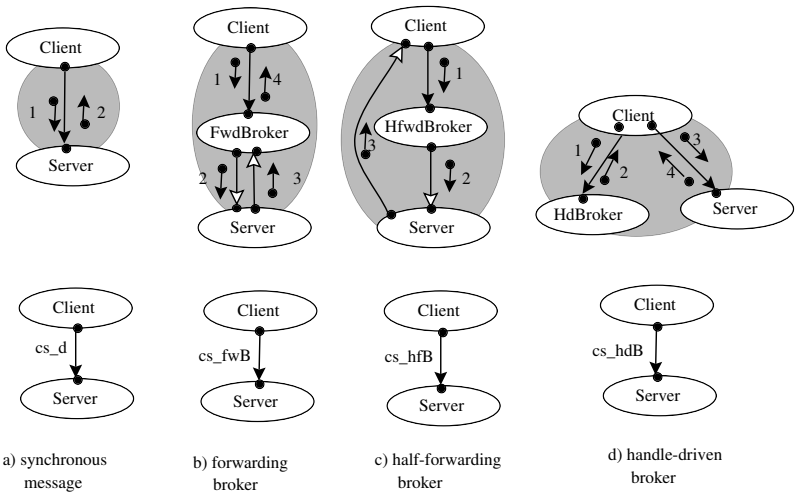

Fig. 2. Client/Server connection types

the filters communicate through asynchronous messages, whereas in (*b*) and (*c*) they communicate through a shared buffer (one pushes and the other pulls). The small arrows attached to the communication arcs indicate the data flow.

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, and a connection through a CORBA interface where the broker is the intermediary between clients and servers. In the first case shown in Fig. 2.a, the client 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 bi-directional (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 Fig. 2.b, 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 Fig. 2.c, 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 Fig. 2.d) 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 the left side of Fig. 5.) 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.

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 (see Fig. 2), 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 Fig. 2.b, 2.c and 2.d behave and interact differently with the client and the server parties, each one corresponds to a different connector type. In total, we have considered four client/server connector types: one direct (Fig. 2.a) and three using the services of a broker. Another example of connector type that contains three roles (two filters and a shared buffer) is shown in the left-hand side of Fig. 3.b and 3.c. 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 present work, we first identified the connector types associated to different architectural patterns, then defined graph transformation rules to translate each connector to an LQN submodel. The transformation approach for different connections is illustrated in Figures 3 to 5 in a more intuitive, higher-level graphical notation tailored to our problem domain, rather than in the more detailed PROGRES notation.

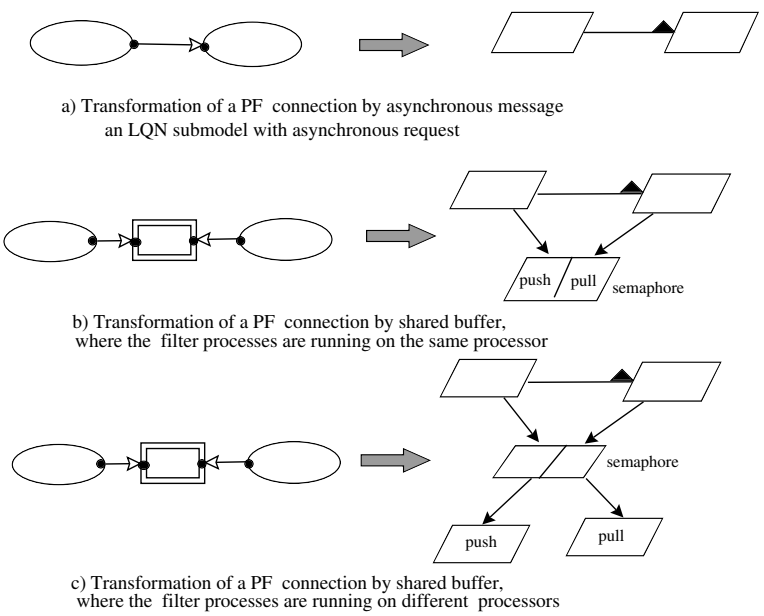

Fig. 3. Transformation of pipeline pattern

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

Fig. 3.a, 3.b and 3.c show the translation of the Pipe and Filters connection types to LQN. The first is using asynchronous messages and the other two a shared buffer. The translation is quite straightforward, only the way LQN models 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 different critical sections executed by the tasks accessing the buffer (two in this case, "push" and "pull"). It is interesting to note that, although the software architecture in Fig. 3.b and Fig. 3.c is exactly the same, the difference in the allocation of processes to processors leads to quite different LQN submodels.

Fig. 4.a, 4.b and 4.c illustrate the transformation of 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.

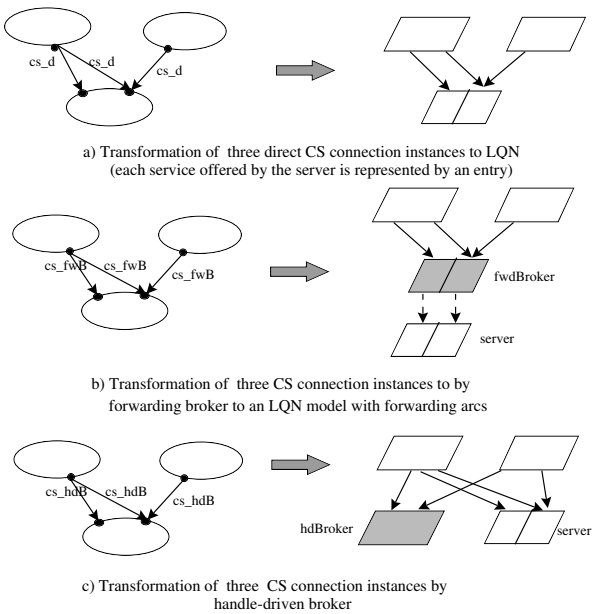

Fig. 4. Transformation of client-server pattern

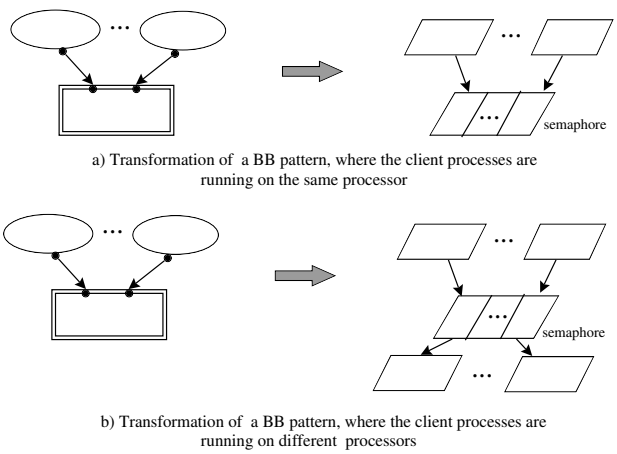

Fig. 5. Transformation of blackboard pattern

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 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 Fig. 4.b, 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 in Fig. 4.c sends two separate requests, one to the broker for getting the handle, then another to the desired server entry directly.

Fig. 5.a and 5.b show the transformation for the blackboard connection type. The performance model captures the serialization delays introduced by the constraint that the blackboard should be accessed by one client at a time. Similar to the pipeline with shared buffer, the same software architecture will generate different performance models depending on whether the clients are running on the same or on different processors. In the later case, the LQN tasks that represent the critical sections executed by the each client are co-allocated on the same processor as the respective client.

4 PROGRES Graph Transformations

The authors have followed the following principles when designing the PROGRES transformation program:

a. 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, 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.
b. 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.
c. The output ports do not have any correspondent in LQN. However, they play a role in the two-step translation process of server-to-server connections, as illustrated in Fig. 7.
d. Processors and devices, which are attributes in the architectural view, become full-fledged nodes in LQN (not illustrated in the paper due to space limitations). This happens because the issue of resource allocation is secondary to the software development process, but is central to performance analysis.

The graph schema according to the PROGRES language [9 to 11] is presented in Fig. 6. The upper part of the figure contains the *input schema* for architectural

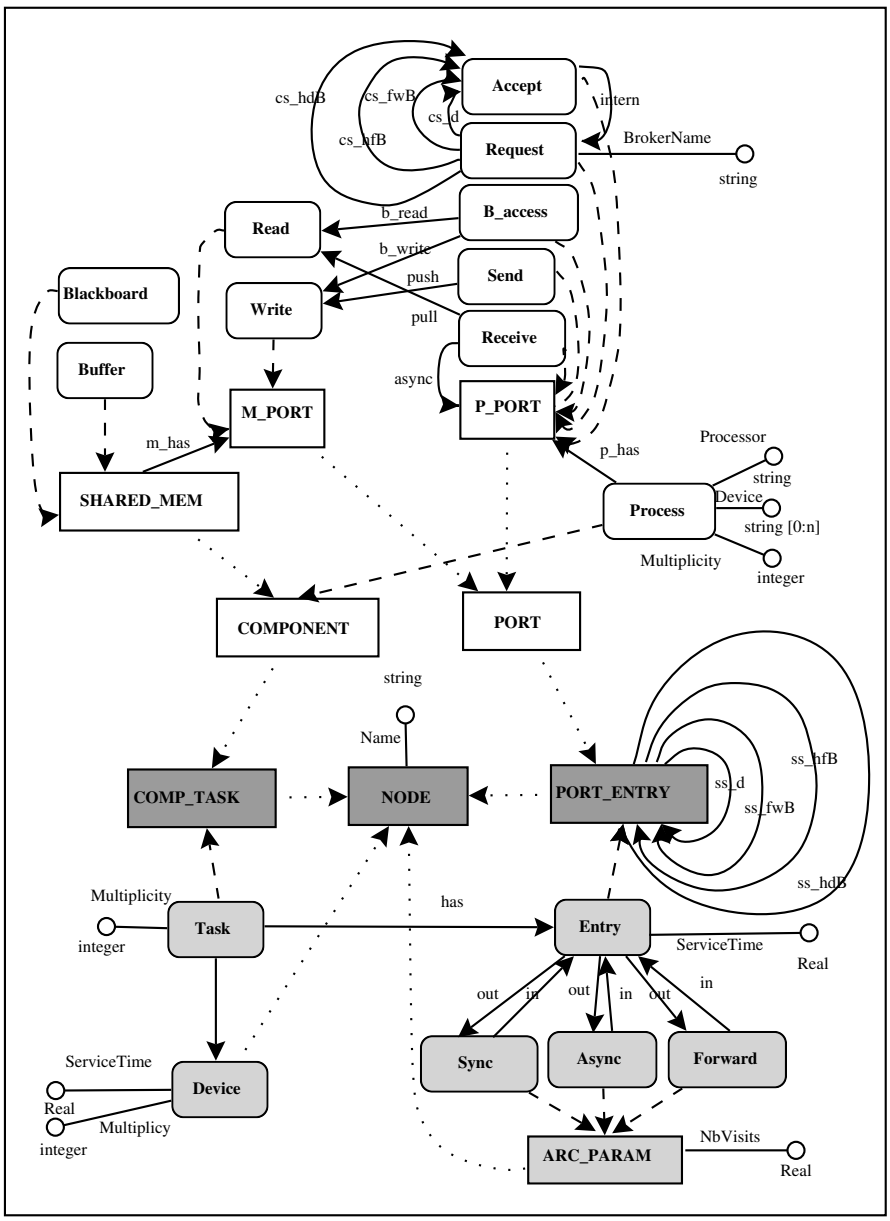

Fig. 6. The joint graph schema of the architectural description and the LQN model

descriptions and the lower part the *output schema*for LQN models (light-gray nodes). In order to accommodate graphs in intermediary translation stages, the two schemas are joined together by three nodes shown in dark-gray at the base of the node class hierarchy (*NODE, COMP_TASK, and PORT_ENTRY*). Also, some intermediary edge types (*ss_d, ss_fwB, ss_hfB, and ss_hdB*) were found to be necessary in the process of translating server-to-server CS connections, which appear in tiered client/server systems. Such edges are illegal in both input and output schemas; they are generated and then deleted during a two-step translation process, as shown later in this section.

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 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 edge type (*cs_d, cs_fwB, cs_hfB and cs_hdB*, respectively). 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, a broker has an important impact on the system performance, and it is explicitly modelled in LQN.

The LQN graph notation has "task" and "device" and "entry" nodes, which are described by the corresponding node types in the output schema. The LQN arcs may represent three types of requests (synchronous, asynchronous and forwarding); a parameter indicates the average number of visits associated with that request. Since PROGRES edges cannot have attributes, we represent an LQN arc by three elements: an incoming edge, a node carrying the parameter and an outgoing edge.

In the case of a layered (or "tiered") client/server system, some server nodes play a dual role of server to its own clients, and of client to other servers. Such a server component owns 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. Servers playing a dual role introduce an additional step in the translation of server-to-server connections, as shown in Fig.7.*a*. Firstly, the internal mapping between the input and output ports of the upper server is used to generate an appropriate number of request edges of an intermediary type (*ss_d, ss_fwB, ss_hfB, or ss_hdB*). Secondly, the translation process is applied to each intermediary edge as if it were a "normal" CS connection (*i.e.*, an *ss_d* edge is treated as a *cs_d* connection, an *ss_fwB* edge as *cs_fwB* connection, and so on.) Fig. 7.a illustrates the two-step translation process applied to a subsystem with two servers involved in several client/server connections of *cs_d* type: first the edges of *ss_d* type (represented by dotted lines) are created, then are translated into LQN like normal *cs_d* connections. Fig.7.b shows a sequence of PROGRES derivations for the first step. Each derivation deals with a two-edge link (shown in thick lines) from an input port of the upper

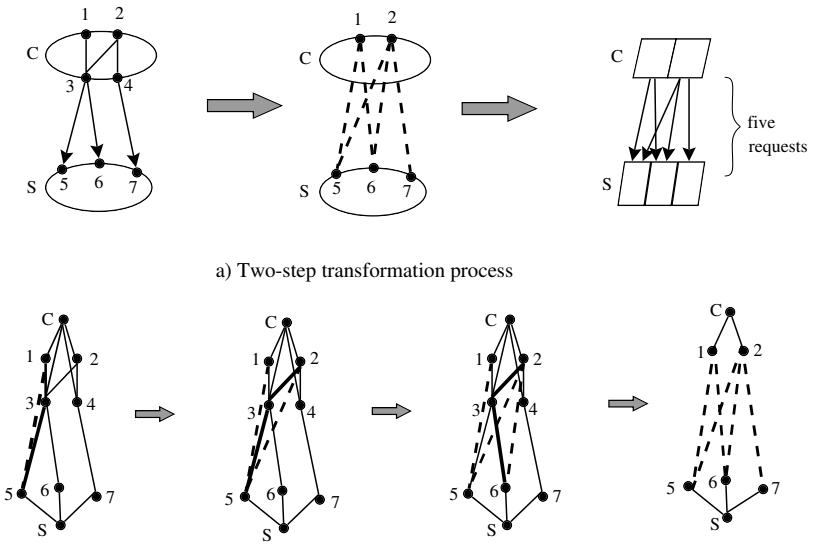

Fig. 7. Illustration of the two-step transformation of server-to-server CS connections for a simple case without brokers

server to an input port of the lower server. There are five such links in our example, each generating a connection edge of an intermediary type, as explained before.

The PROGRES transformation process is organized as follows: first all server-to-server CS connections are transformed into intermediary type edges as explained above, then transactions are executed for all connections found in the graph. These transactions are designed to transform any type of architectural connections into LQN sub-models. New nodes and edges of types described in the output schema are added to the graph, but not all the nodes and edges of types described in the input schema can be removed from the graph, as they may participate in more than one connection instance. Therefore, a general clean-up phase is required at the end, after all the architectural connections have been processed. For example, in the clean-up phase nodes representing output ports (which do not have an LQN equivalent) will be removed from the graph, and multiple broker tasks with the same name will be merged into a single task (in the case more than one was created by different transactions).

Fig.8 shows an example of production rule for the transformation of a CS connection with handle-driven broker. Nodes 1 and 2 represent architectural components in the left-hand side, and tasks in the right-hand side. The input port 4 is "transformed" into entry 6, and the output port 3 is kept in the graph until the clean-up stage. A broker task 8, its entry 10, client entry 7 and two request nodes 5 and 9 are also added to the graph.

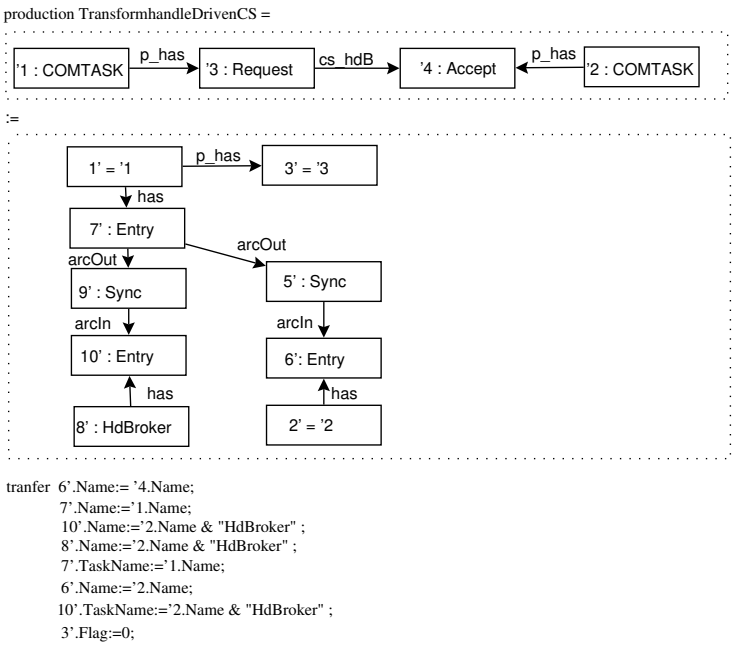

Fig. 8. Production rule for the transformation of a CS connection with handle-driven broker

5 Conclusions

The automatic generation of LQN performance models from software architecture descriptions raises a number of challenges due to the fact that each view has a different semantics, purpose and focus, which must be bridged by the translation process. The architectural view represents only the software components of the application under development, and may hide operating system and midware services. However, such services have a considerable impact on the overall system performance, and must be modelled in LQN explicitly. Another discrepancy between the two views arises from the different role played by the issue of resource allocation. Whereas the architectural view concentrates on the software under development, the performance model takes a *vertical view* of the whole system, modelling not only the application software, but also relevant characteristics of the underlying operating system, midware and hardware. The final result of the transformation process is an LQN model that can be written to a file according to a predefined format and can be solved with existing LQN solvers [5]. The analysis of an LQN model and the use of its results to identify performance trouble spots is outside of the scope of this paper, but is described in other work such as [5, 8, 14, 16, 17].

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. Shousha, D.C. Petriu, A. Jalnapurkar, K. Ngo, "Applying Performance Modelling to a Telecommunication System", Proc.of the First Int. Workshop on Software and Performance (WOSP'98), pp. 1-6, Santa Fe, New Mexico, October 1998.
15. C.U. Smith, *Performance Engineering of Software Systems*, Addison Wesley, 1990.
16. C.M. Woodside, "Throughput Calculation for Basic Stochastic Rendezvous Networks", *Performance Evaluation*, Vol. 9, Number 2, pp.143-160, April 1995.
17. 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.