Deriving Software Performance Models from Architectural Patterns by Graph Transformations

by

Xin Wang

A thesis submitted to the Faculty of Graduate Studies
in partial fulfilment of the requirements for the degree of
Master of Science
Information and System Science

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
Ottawa, Canada
September, 1999

To my husband Lizhong

Abstract

Performance characteristics play an important role in defining the quality of software products, especially in the case of real-time and distributed systems. The developers of such systems should be able to assess and understand the performance effects of various architectural decisions, starting at an early stage, when changes are easy and less expensive, and continuing throughout the software life cycle. This can be achieved by constructing and analyzing quantitative performance models that capture the interactions between the main system components and point to the system's performance trouble spots. This thesis contributes toward bridging the gap between software architecture and performance analysis. It proposes a systematic approach, and implemented two versions of the proposed approach based on graph transformations, by using PROGRES (Programmed Graph Rewriting System) to build LQN (Layered Queueing Network) performance models from descriptions of high-level software architecture of a system and more exactly from the architectural patterns used in the system.

Acknowledgments

This thesis would not have been possible without the encouragement, support of many people. In particular, I am very much in debt to my supervisor Professor Dorina C. Petriu, for motivation when I needed it most, and many hours of interesting discussion. Without her guidance and helpful suggestions, working on this thesis would have been much more difficult. Her inspiration and motivation will have a long-lasting impact on my future endeavor too.

During my studying at Carleton, I was able to benefit from the knowledge of many professors, staff, former graduate students, research engineers. I would like to thank all of them, especially the members of the RADS lab, for their full support, friendship and a wonderful working environment.

The financial assistance of the Communications and Information Technology Ontario (CITO) is gratefully acknowledged.

Last but not least, I would like to give my special thanks to my husband Lizhong. Without his consistent support and encouragement, I could have never gone this far.

Contents

List of Figures

List of Abbreviations

CCB	Component/Connector Based
LHS	Left-Hand Side
LQN	Layered Queueing Network
PROGRES	Programmed Graph Rewriting System
RHS	Right-Hand Side
SPE	Software Performance Engineering
UML	Unified Modelling Language

CHAPTER 1

Introduction

1.1 Motivation

Performance is a key criterion in defining the quality of software products, especially in the case of real-time and distributed systems. The goal of computer systems engineers is to get the highest performance for a given cost. As the field of computer design matures, the computer industry is becoming more competitive, and it is more important than ever to ensure that the alternative selected provides the best performance trade-off.

Performance evaluation is required at every stage in the life cycle of a computer system. In order to meet the performance requirements of software 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 that proposes to use quantitative

methods and performance models in order to assess the performance effects of different design and implementation alternatives during the development of a system [Smith90]. SPE promotes the idea that the integration of performance analysis into the software development process, from the earliest stages to the end, can insure that the system will meet its performance objectives. This would eliminate the need for "*late-fixing*" of performance problems, a frequent practical approach that postpones any performance concerns until the system is completely implemented. Late fixes tend to be very expensive and inefficient, and the product may never reach its original requirements.

Although the need for SPE is generally recognized by industry, there is still a cognitive gap between the software and the performance domains. Software developers are concerned with designing, implementing and testing the software, but they are not trained in performance modelling and analysis techniques. The software development teams usually depend on specialized performance groups to do the performance evaluation work, which 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 development cycles. There is no time left for SPE, which traditionally implies "*manual*" construction of the performance models.

This thesis contributes toward bridging the gap between software architecture and performance analysis. It proposes a systematic approach, based on graph transformations, by using PROGRES (Programmed Graph Rewriting System) to build LQN (Layered Queueing Network) performance models from descriptions of high-level software architecture of

a system.

By automating the construction of the performance models from software architectures, the time and effort required for SPE will be considerably reduced, and the consistency between the model and the system under development more easily maintained. Such a model will be solved with existing performance analysis tools, producing much faster feedback for the software development team.

1.2 Contributions

The research contributions of this thesis are as follows:

- Developed a formal approach for generating of LQN (Layered Queueing Network) performance models from the high-level software architecture of a system, and more exactly from the architectural patterns used in the system.
- Implemented two versions of the proposed approach by using an existing graph grammar tool named PROGRES (Programmed Graph Rewriting System)[Schuerr90]. The first version takes as input the high-level architectures of a system described in an ad-hoc language based on component/connector relationships[Allen97]. The second version accepts architectural descriptions expressed in UML collaborations [Booch99].

The results of the thesis are presented in two papers:

- D.C. Petriu, X. Wang "Deriving Software Performance Models from Architectural Patterns by Graph Transformations", Proc.of the 6th International Workshop on The-

ory and Applications of Graph Transformations TAGT'98, Paderborn, Germany, November 1998.

- D.C. Petriu, X. Wang "From UML Descriptions of High-level Software Architecture to LQN Performance Models", Accepted by *Applications of Graph Transformation with Industrial Relevance*, Monastery Rolduc, Kerkrade, The Netherlands, September 1999.

1.3 Thesis Outline

The thesis is organized as follows:

Chapter 2 provides an overview of the background information related to this thesis, such as Software Performance Engineering (SPE), software architectural patterns, Layered Queueing Network (LQN) model, Unified Modelling Language (UML) and Programmed Graph Rewriting System (PROGRES).

Chapter 3 describes the graph transformations used to build software performance models (LQN models) for distributed and/or concurrent software systems from an architectural descriptions based on architectural components and connectors which introduced by Allen and Garlan in [Allen97]. We named this Component/Connector Based(CCB) description throughout the thesis.

Chapter 4 describes the graph transformations used to build software performance models (LQN models) from UML (Unified Modelling Language) descriptions of the high-

level architecture of a system, and more exactly from the architectural patterns used in the system.

Chapter 5 concludes the thesis research, summarizes contributions of the thesis and identifies directions for future research.

CHAPTER 2 Background Literature

This chapter presents an overview of the background information related to the thesis, such as Software Performance Engineering (SPE), software architectural patterns, Layered Queueing Network (LQN) model, Unified Modelling Language (UML) and Programmed Graph Rewriting System (PROGRES).

2.1 Software Performance Engineering

End to end performance of a system refers to the response time or throughput as seen by the users. Performance characteristics play an important role in defining the quality of software products, especially in the case of real-time and distributed systems.

Most of today's software development is still heavily influenced by the "*fix-it-later*"

performance tuning approach, meaning to postpone the performance concerns until the system is completely implemented, then trying to "*fix*" its performance problems at the late stage. Late fixes tend to be very expensive and inefficient. Performance tuning will require changes to the program itself, retesting, and in some cases even serious modification of the design which drives up the costs [Smith90]. In some cases, the product will never meet its original performance requirements.

Software Performance Engineering (SPE) is a technique introduced in [Smith90] that proposes to use quantitative methods and performance models in order to assess the performance effects of different design and implementation alternatives during the development of a system. SPE promotes the idea that the integration of performance analysis into the software development process, from the earliest stages to the end, can insure that the system will meet its performance objectives.

The benefits of SPE are as follows [Smith90]:

- Increased productivity, due to the fact that the developers' level of efforts are not spent for an implementation that will later be discarded and that the testing can be focused on critical parts of the software.
- Improved quality and usefulness of the resulting software product by selecting suitable design and implementation alternatives, thus avoiding late tuning modifications.
- Controlled costs of the supporting hardware and software by identifying early what equipment is needed and allowing sufficient time for competitive procurement.

- Enhanced productivity during the implementation, testing, and early operational stages by ensuring that sufficient computing power is available.

The sequential steps for SPE performance assessment concerns as detailed in [Smith90] are:

1. Build the Software Execution Model, a flow-graph that follows the execution of the software.
2. Gather resource requirements for every software module, then aggregate the resource requirements.
3. Map the Software Execution Model to a System Model that contains both the hardware and the software of the system.
4. Solve the model analytically or with simulation. Both best and worst case analysis are need to be conducted.

- If the worst-case performance is satisfactory, go into the software development cycle.
- If the best-case performance is unsatisfactory, seek feasible alternatives.

Software Performance Engineering (SPE) is proven to be able to insure that the system will meet its performance objectives by the means of assessing the performance effects of different design and implementation alternatives during the development of a system. The need for SPE is generally recognized by industry, but is not widely applied.

2.2 Software Architectural Pattern

In order to build large systems, software developers have to scope out the system ahead of time and break it up into manageable pieces. They need ways of specifying what the pieces do and how they communicate with other pieces. Software architecture is the abstraction that give an insight into the system. It is a notion of overall design apart from implementation.

The high-level architecture describes the main system components and their interactions at a level of abstraction that captures certain characteristics relevant to performance, such as concurrency, parallelism, contention for software resources (as software servers and critical sections), synchronization, serialization, etc. A architectural 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.

According to [Allen97], a software architecture represents a collection of computational components that perform certain functions, together with a collection of connectors that describe the interactions between components. A component type is described by a specification defining its functions, and a set of ports representing logical points of interaction between the component and its environment. A connector type is defined by a set of roles explaining the expected behavior of the interacting parties, and a glue specification showing how the interactions are coordinated.

A similar, even though less formal, view of a software architecture is described in the form of architectural patterns [Buschmann96], [Shaw96b] which identify frequently used architectural solutions, such as pipeline and filters, client/server, client/broker/server, layers, critical section, etc.

Architectural patterns express fundamental structure organization schemas for software systems [Buschmann96]. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them. Architectural patterns are high-level patterns in a pattern system. They help software developers to specify the fundamental structure of an application. Every development activity that follows is governed by this structure, e.g. the detailed design of subsystems, the communication and collaboration between different parts of the system, and its later extension.

Software architectural patterns are distinct from design patterns and idioms that predate them. Idioms are language-specific, and design patterns capture relationships at the class and object level. Architectural patterns are a further step up in granularity, capture relationships at the subsystem level [Shaw96b].

Each architectural pattern describes two inter-related aspects: its structure (what are the components) and behavior (how they interact). In the case of high-level architectural patterns, the components are usually concurrent entities that execute in different threads of control, compete for resources, and their interaction may require some synchronization.

This kind of issues contribute to the performance characteristics of the system, and therefore must be captured in a performance model.

The thesis defines graph transformations from a number of frequently used architectural patterns (such as pipeline and filters, client/server, client/broker/server, layered client/server, critical section, etc.) into LQN models.

2.3 Layered Queueing Network Model

Layered Queueing Network (LQN) was developed as an extension of the well-known Queueing Network (QN) model. LQN was first independently developed under the name of Stochastic Rendezvous Networks (SRVN or SRN) in [Woodside89] [Petriu91b] [Petriu94] [Woodside95a] [Woodside98] and under the name Method of Layers(MOL) in[Rolia87] [Rolia92] [Rolia95]. Many features of the two approaches were joined under the name of LQN [Frank95] [Woodside95b].

The Layered Queueing Network is a model of a network of tasks running on processors and communicating via a send-receive-reply pattern, in which the sender of a message waits for a reply pattern we call a *rendezvous*, a RPC, or synchronous messaging. The tasks may also send messages without reply, known as asynchronous messaging. By modelling a system before it is implemented, performance bottlenecks are revealed and it may be possible to improve the decision of functions between tasks and/or the allocation of tasks to processors[Woodside95b].

The main difference with respect to QN is that in LQN a server may become a client to other servers while serving its own clients. High-level servers become clients to lower level servers. Therefore, the solution process will expand the services time at higher level servers due to the inclusion of queueing delays at lower level servers.

The advantages of the layered approach to software analysis are mentioned in [Woodside95b]:

- It determines software queueing effects (requests queued at servers) and competition between applications.
- It identifies software bottlenecks.
- It identifies the contribution of different high-level software components to performance.

The LQN toolset presented in [Frank95] includes both simulation as well as analytical solvers that merge the best previous approaches[Woodside95b].

2.3.1 LQN Model Components

There are two basic building blocks in a LQN model: *tasks* and *requests*.

Task

A task is an entity that models a software process execution demand and executes some work if its processor is available. A task may be either a *client* task or a *server* task.

A client task sends requests to other tasks. A server task performs work on behalf of the requests from its clients. A server itself may also be a client to its lower level servers by making requests to those as part of fulfilling their own work to the higher level client. Each task may have different classes of workloads on the processor by representing it with several entries. Each entry provides a different service pattern and a different workload. However, all entries of one task share a common task queue. The task queue scheduling disciplines supported by LQN, that controls the order in which requests are processed, is First In First Out (FIFO) and Head of Line (HOL).

A server may be a single server, a multiserver or an infinite server. A single server is modelled as a single task, which handles only one request at a time. Concurrency in LQN is modelled by multi-servers and replicated servers. A multiserver contains a multitude of copies of a task, yet all copies share one common queue for incoming requests. A replicated server, however, is similar to the multiserver, except that each copy task has its own request queue. An infinite server is modelled as an infinite number of tasks on an infinite number of processors that can handle an infinite number of requests. For example, Network delays are often modelled as infinite servers.

Entry

An LQN server may offer more than one service, each one with it own service time and visit ratio to other severs. Each service is modelled as an entry of the task. It is assumed that all the requests for all entries of a task are queued in a common task queue.

The execution of a server entry following the reception of a message by an entry may-be broken into two parts, the first part named first phase ends when the reply is sent back, and the second part is the subsequent phases after the reply.

Request

A communication request from a node playing the role of client to a node playing the role of server can be *synchronous, asynchronous* or *forwarding.*

- *A synchronous request* blocks the client until the server sends back the reply.
- *An asynchronous request* is a request that the client continues its work in parallel with the server.
- *A forwarding request* is similar to a synchronous request from the client's point of view. The difference is that more than one servers are involved. The first server forwards the request to the next server, and itself is free to do other work, after the second server finishes the request, the second server send back the reply to the original client (instead of sending to the first server which forwards the request). The original client is blocked until it receives the reply. There can be more than two servers in the forwarding chain.

2.3.2 Graphical Representation of an LQN Model

An LQN model is represented as an acyclic graph which includes the follows:

- *Parallelograms* represents software entities (named also tasks).
- *Ellipses* represents hardware devices.

- Every kind of service offered by an LQN task is modelled by a task entry, drawn as a parallelogram "slice".
- Arcs represents service requests.
    - *Synchronous requests* are represented by full-head arrows.
    - *Asynchronous requests* are represented by half-head arrows.
    - *Forwarding requests* are represented by full-head arrows with a dashed line.

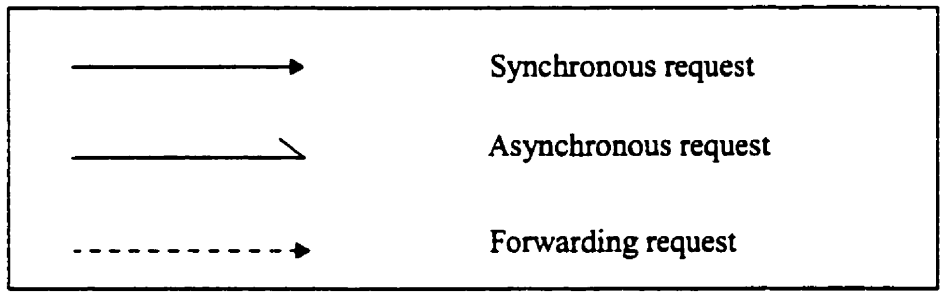


Figure 2.1: Graph representations for requests in LQN

Figure 2.2 illustrates a simple example of an LQN model for a three-tiered client/server system: at the top there are two groups of stochastic identical clients. Each clients send requests for a certain service offered by *Application* task, which represents the business layer of the system. Every kind of service offered by an LQN task is modelled by a task 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 services from two different *Database* entries. Every software task is running on a given processor shown as a ellipse; more than one tasks can share the same processor. The word "*layered*" in the name LQN does not imply a strict layering: a task may call 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 re-

ply from the provider of service. Although not explicitly illustrated in the LQN notation, each server has an implicit message queue where the incoming requests are waiting their turn to be served. Servers with more than one entries still have a single input queue, where requests for different entries wait together.

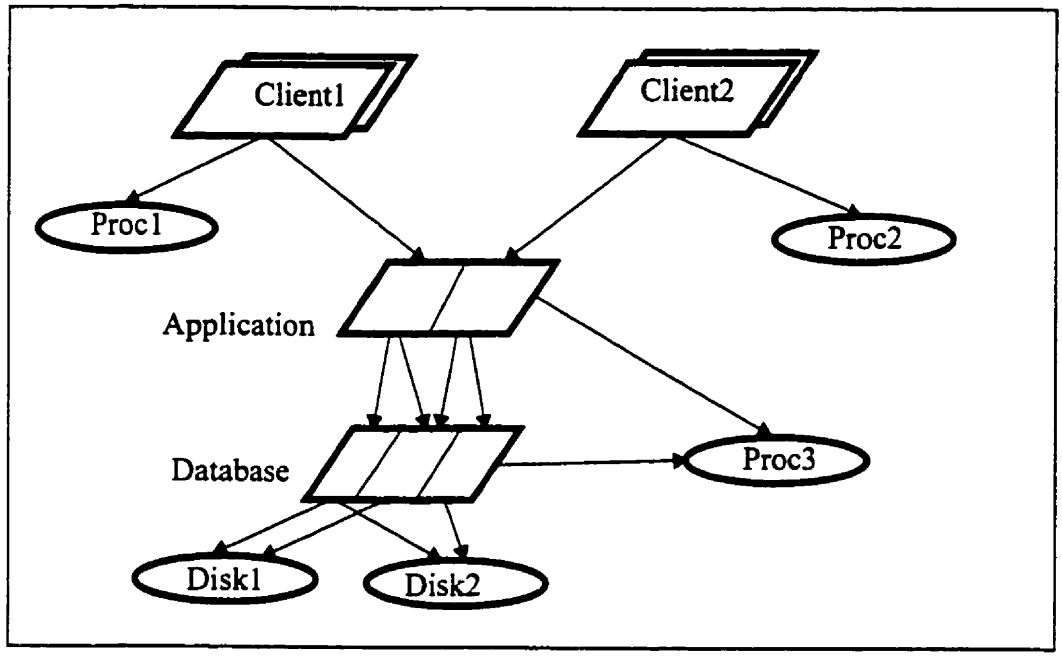


Figure 2.2: Sample LQN model of a database application

2.3.3 LQN Parameters

The most basic parameters needed to build the complete LQN model are:

- s_{iep} = mean execution time of task *i*, during phase *p* of entry *e*.
- y_{edp} = mean or expected number of synchronous (rendezvous) messages sent from entry *e* to entry *d* during phase *p* of entry *e*.
- z_{edp} = mean or expected number of asynchronous (send-no-reply) messages sent from entry *e* to entry *d* during phase *p* of entry *e*.

2.3.4 Solving LQN Models

LQN model can be solved with the solving tools provided in the toolset [Franks95]. The same input file can be solved by both analytical tools such as *lqns* and simulation tools such as *ParaSRVN*.

The *ParaSRVN* simulation solver mimics the behavior of the system by conducting a discrete event simulation, which is expected to be considerably slower than the analytical solver. However, the simulation is more powerful, in the sense that it allows for more details models of the system, whereas the modelling power of the analytical model is limited by mathematical assumptions.

The *lqns* analytical solver technique consists of decomposing the LQN model into submodels and solving individual sub-models with Mean Value Analysis. Outputs of some sub-models are fed into other sub-models. Iteration among the sub-models until continues waiting time results converge for all layers. *Lqns* is also a simulation solver.

2.3.5 Results of LQN Models

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.

Overall, LQN was developed especially for modelling concurrent and/or distributed

software systems. LQN determines the delays due to contention, synchronization and serialization at both software and hardware levels. LQN was applied to a number of concrete industrial systems and was proven useful for providing insights into performance limitations at software and hardware levels.

2.4 Unified Modelling Language

The Unified Modelling Language (UML) is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. It simplifies the complex process of software design, making a "blueprint" for construction. [UML]

The Unified Modelling Language (UML) is created by the joint efforts of Grady Booch, Ivar Jacobson, and Jim Rumbaugh as a response to the Object Modelling Group's (OMG) request for a proposal for a standard object-oriented methodology. It is the successor to the wave of object-oriented analysis and design (OOA &D) methods that appeared in the late 80s and 90s.

The UML is a graphical modelling language, not a methodology. It is a language for expressing the constructs and relationships of complex systems. UML is more complete than other methods in its support for modelling complex systems and is particularly suited for including real-time systems.

2.4.1 UML Goals

According to [UML], the primary goals in the design of the UML are as follows:

- Provide users a ready-to-use, expressive visual modelling language so they can develop and exchange meaningful models.
- Provide extensibility and specialization mechanisms to extend the core concepts.
- Be independent of particular programming languages and development processes.
- Provide a formal basis for understanding the modelling language.
- Encourage the growth of the OO tools market.
- Support higher-level development concepts such as collaborations, frameworks, patterns, and components.
- Integrate best practices.

2.4.2 UML Major Features

The major features of UML include [Douglass98]:

- Object model
- Use case and scenarios
- Behavioural modelling with state charts
- Packaging of various kinds of entities
- Representation of tasking and task synchronization
- Models of physical topology

- Models of source code organization
- Support for object-oriented pattern

In terms of the views of a model, the UML defines the following graphical diagrams [UML]:

- Use case diagram
- Class diagram
- Behaviour diagrams:
    - State chart diagram
    - Activity diagram
    - Interaction diagram
    - Sequence diagram
    - Collaboration diagram
- Implementation diagrams:
    - Component diagram
    - Deployment diagram

These diagrams provide multiple perspectives of the system under analysis or development. The underlying model integrates these perspectives so that a self-consistent system can be analyzed and built.

2.4.3 UML Collaboration

UML is very rich when it comes to format. It has a number of different models and diagrams that fit the different purpose of the designers. Some of them are on the object and class level, like class diagram and behaviour diagrams. But as we know architectural patterns are a further step up in granularity, capture relationships at the subsystem level [Shaw96b]. So, the UML feature better suited to describe the software architecture patterns for the purpose of the thesis are UML *collaborations*.

According to the authors of UML [Booch99], a *collaboration* is a notation for describing a mechanism or pattern, which represents "a society of classes, interface, and other elements that work together to provide some cooperative behavior that is bigger than the sum of all of its parts." A collaboration has two aspects: structural (usually represented by a class/object diagram) and behavioral (an interaction diagram). Collaborations can be used to hide details that are irrelevant at a certain level of abstraction; these details can be observed by "zooming" into the collaboration. The symbol for collaboration is an ellipse with dashed lines, and may have an "embedded" square showing template classes. Another special UML notation employed in this section is that of an active class (object) which has its own thread of control, represented by a square with thick lines. An active object may be implemented either as a process (identified by the stereotype <<process>>), or as a thread. We will discuss more about UML collaborations in chapter 4.

In this thesis we use two descriptions of software architecture, one of the two descriptions of software architecture is using UML collaborations. UML is attractive because it is

a standard, and is rapidly gaining acceptance in the software industry. However, UML is a very rich, sometimes informal language, which raises a number of yet unresolved issues.

2.5 Programmed Graph Rewriting System

Graphs play an important role within many areas of applied computer science, and there exists an abundance of visual languages and environments which have graphs as their underlying data model. Furthermore, rule-based languages and systems have proven to be well-suited for the description of complex transformation or inference processes on complex data structures. A graph grammar is a set of productions that generates a language of terminal graphs and produces nonterminal graphs as intermediate results. 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[Schuerr97].

Programmed Graph Rewriting System (PROGRES) was developed by Andy Schuerr [Schuerr90]. It is a toolset which is available as a free software for university researches. PROGRES contains a visual programming language in the sense that it has a graph-oriented data model and a graphical syntax for its most important language constructs. This system and its programming language PROGRES were already used in the following areas[Schuerr97]:

- For specifying tools and data structures of integrated software engineering environments for describing process modelling and version control.
- As the underlying fundament of a new approach to diagram parsing.

- For defining the semantics of a visual database query language.

2.5.1 Components of PROGRES Graph

A PROGRES graph consists of labelled *nodes* and directed labelled *edges*. The nodes represent different objects, and there can be different node types. The edges represent relationships between two nodes, and there can be different edge types too. *Attributes* may be attached to nodes only.

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

- *Node* types, which determine the static properties of their nodes instances. The declaration of a node can be either *positive node* (node for short) or *negative node*.
    - A *positive node* is represented by a solid rectangle. It matches a single node of the regarded graph, which fulfils all required conditions.

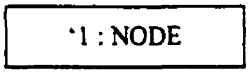


Figure 2.3: Graph representation of a positive node

    - A *negative node* is represented by a crossed-out solid rectangle. A negative node which matches a node of the regarded graph leads to failure of the pattern matching processes, a negative node without a match is simply ignored afterwards.

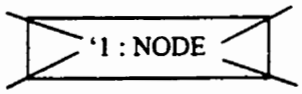


Figure 2.4: Graph representation of a negative node

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

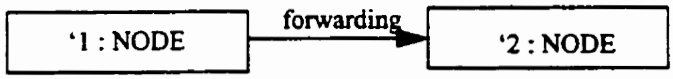


Figure 2.5: Graph representation of an edge

- Derived relationship, which model often needed paths of a given graph, which defined by the means of *path* or *restriction* expression.
    - A *path* is a constraint that the two nodes it is attached to must meet, it is a more complicated relationship than edge types. A *Path* is represented by double arrow pointed from one node to the other node.

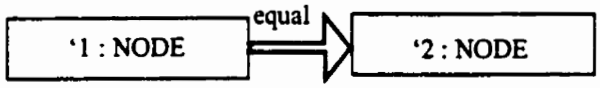


Figure 2.6: Graph representation of a path

    - A *restriction* is a constraint that the one node it is attached to must meet, it is a more complicated relationship than edge types. A *restriction* is represented by double arrow pointed to the node.

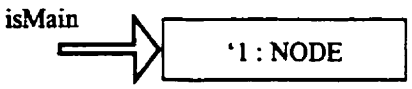


Figure 2.7: Graph representation of a restriction

Attribute

Attributes are needed to store additional information that is not necessary to be represented in the graph structure. Attributes represent the information that is local to a particular node and which has an unimportant structure from the current point of view.

An intrinsic attribute is stored data of a node type that is explicitly manipulated. *A derived attribute* is stored data of a node that is automatically generated from the intrinsic attributes of a node, or from programmed relationships amongst nodes.

Standard attribute domains like integer, string and boolean together with their functions are a built-in part of the language PROGRES. PROGRES can be extended by adding external attribute types, as well as new, external function. External function written in C or Modula-2 can be called within a PROGRES specification. The function are written and compiles into object modules. Several mandatory functions must be provided for an external attribute type. The external attribute types and functions are imported into the PROGRES in the import part of any declaration list. The PROGRES interpreter uses a dynamic linker (dll) for binding object-files to the PROGRES executable.

2.5.2 Definition of Graph Schema

The language feature provided by PROGRES to define a static properties of graphs take in the form of a graph schema which is conceptually similar to a database schema. The graph schema definition part of a PROGRES specification enables us to specify static properties of any class of directed, attributed, node and edge labelled graphs.

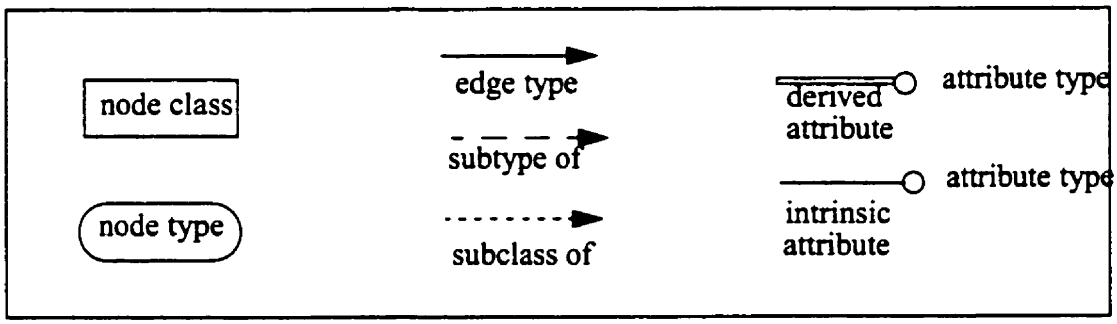


Figure 2.8: PROGRES graph schema notation

Figure 2.8 illustrates the notation of graph schema.

- Rectangle boxes represent *node classes* which are connect to their superclasses by means of dashed edges representing "*is-a*" relationships.
- Boxes with round corner represent *node types* which are connected to their uniquely defined classes by means of dashed edges representing "type is instance of class" relationships.
- Solid edges between node classes represent *edge type* definitions.
- Circles attached to node classes represent *attributes* with their names above or below the connection line segment and their type definition nearby the circle.

Node classes can have common attributes that will be inherited by node types. A node types is a classification of a node and represents a concrete object. The instance of node types are the objects in the graph which are manipulated. A node type can inherit from several node classes (multiple inheritance). PROGRES also enable us to build hierarchies of node classes by exploiting multiple inheritance as the relation between node classes. As usual in object-oriented languages, PROGRES uses the "*is-a*" notion to express the inheritance relation. Multiple inheritance may be used to cut down the size of graph schema definitions considerably.

2.5.3 Definition of Graph Transformation

The graph schema definition part of a PROGRES specification enables us to specify static properties of any class of directed, attributed, node and edge labelled graphs. The next step is graph transformation which can be done using productions, tests, functions and transactions.

Production Rule (Rewriting Rule)

A graph *production rule* performs a basic graph transformation, by selecting first a subgraph that matches its left-hand side (subgraph matching step) and replacing it by its right-hand side (subgraph replacement step).

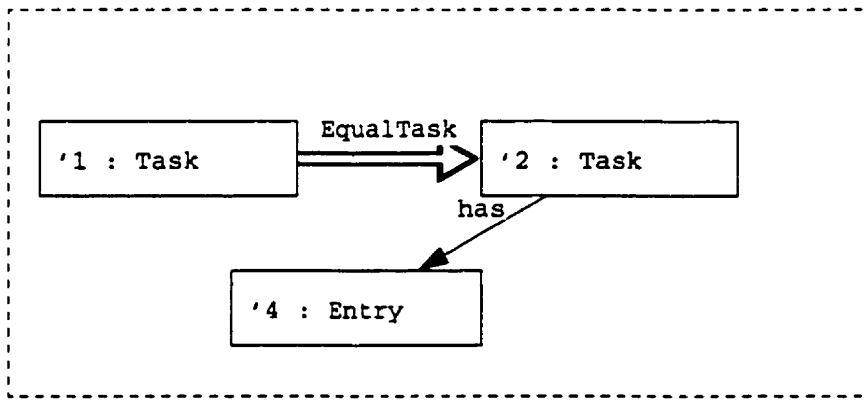
The *left hand side (LHS)* of the production rule describes a sub-graph that must exist in the graph being manipulated. Constraints on a LHS match can take the form of node attributes value conditions, restrictions on the edges associated with the node(s), or the ex-

istence of paths of edges between nodes. If a subgraph cannot be selected or the constraints are not satisfied then no operations are performed.

The *right hand side (RHS)* of the production rule defines the node transformations that take place, which may involve node(edge) additions, node(edge) deletions, node(edge) type changes, etc. Attribute values of nodes can also be altered by the RHS. The adjacent edges and attributes of a node are preserved unless explicitly altered by the RHS.

A production rule can take parameters as input and provide output values as well. Edge types cannot be parameter values although a node type can be a parameter. Output parameters must have the *out* keyword when the production (or test) is declared and whenever the production is used. This makes it easy to distinguish which parameters are input and output values.

```
production MergeTask =
```



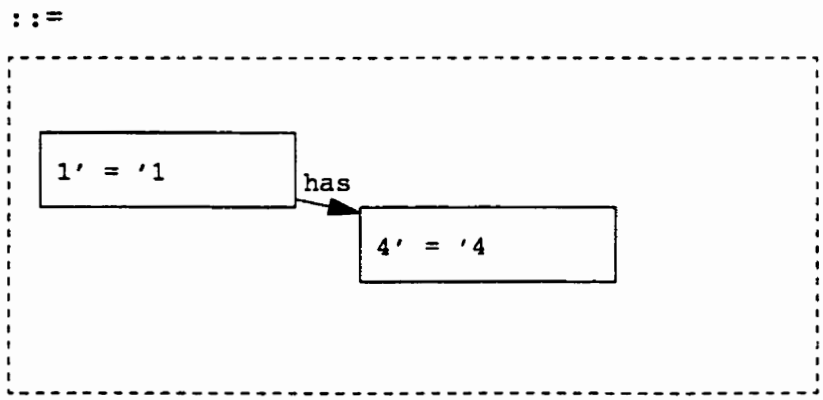


Figure 2.9: An example of an production rule

Figure 2.9 illustrates a simple production rule to merge two tasks together. In LHS, if a *EqualTask* path is found and the *task* (node 2) has an *entry*, then in RHS, one *task* (node2) is deleted and the other two nodes remain the same and the *task* has the *entry*.

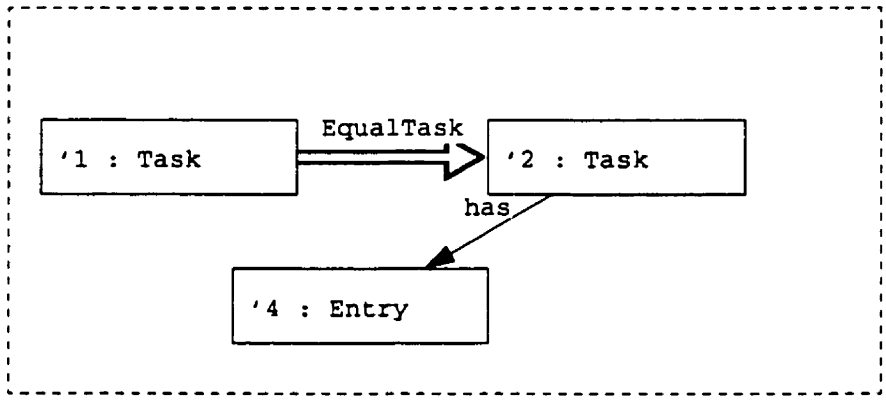
Overall, the general approach of graph transformation using a production rule is as followed:

- The matching succeeds if all *positive nodes* and *edges* patterns are found and the *negative node* pattern do not exist.
- All *positive nodes* and *edges* of the left-hand side which have no counterparts in the right-hand side have to be deleted, including all incident edges of deleted nodes.
- All *nodes and edges* of the right-hand side with no counterparts in the left-hand side are added to the host graph.
- Finally, new *attribute* values are computed by evaluating values (these expressions are computed before any modification of the host graph is performed).

Test

A *test* looks like the left-hand side of a production and only does not have a right-hand side. It does not perform the replacement of the LHS, instead it returns true or false according to whether the LHS is found or not.

```
test DuplicatedTask = (task1,task2:string)
```




```
end;
```

Figure 2.10: An example of a test

Figure 2.10 illustrates a simple *test* to find a duplicated task. If found, the test itself will return Boolean value 1, and otherwise 0.

Function

A *function* in PROGRES is like a procedure in other languages. It does a piece of work which does not need a production. It can have parameters and a return value.

```
function addSize( Val:integer; node:NODE) -> integer=
  Val+ node.Size
end;
```

Figure 2.11: An example of a function

Figure 2.10 illustrates an example of a simple function. It just increases the *Size* of a *NODE* by *Val* and returns back the new value.

Transaction

A *transaction* is very much like a procedure in other languages but it either succeeds in all of its graph operations or it leaves the graph in its initial state. Transactions can have input parameters, output parameters, local variables and recursively execute.

```
transaction MergeAllTasks =
    loop
      MergeTask
    end
  end;
```

Figure 2.12: An example of an transaction

Figure 2.11 illustrates an example of a simple transaction. In the transaction, production *MergeallTask* is called in a loop until LHS of the production is not found in the graph.

2.5.4 Components of a PROGRES Program

A program written in PROGRES may have the following components:

- *Schema Section* contains all the *node types*, *node class* and *edge types*, e.g.:

```
section DataScheme
node class NODE
    intrinsic
      index name : string := "node";
  end;
edge type has : OBJ_TASK -> OP_ENTRY;
```

- A *transaction* named *MAIN* indicates where the program starts from. A PROGRES

program must have one and only one *MAIN transaction*, e.g.:

```
transaction MAIN =
  UMLtoLQN
    & writeToFile
  end;
```

- A list of *transactions*, *tests* and *functions* like procedures doing pieces of work.
- A set of *production rules* specify subgraph transformation.
- A list of *imported types* and *function* doing external works like, e.g.:

```
from RealNumbers import

    types
    Real;

    functions
    BQPlusQuote     : ( Real, Real) -> Real,
    BQMinusQuote    : ( Real, Real) -> Real,
    Real_StringToValue
                    : ( string) -> Real,
    Real_ValueToString
                    : ( Real) -> string;

  end;
```

Overall, PROGRES is a useful visual programming language and graph rewriting system. Related work with generating software performance model by graph transformation includes generating LQN model from Trace-Based Load Characterization (TLC). [Hrischuk99]

The above are brief introduction to some background information related to the thesis, such as Software Performance Engineering (SPE), software architectural patterns, Layered Queueing Network (LQN) model, Unified Modelling Language (UML) and Pro-

grammed Graph Rewriting System (PROGRES). In this thesis we developed and implemented a formal approach for generating of LQN (Layered Queueing Network) performance models from the high-level software architecture of a system, and more exactly from the architectural patterns used in the systems by using PROGRES.

CHAPTER 3

From Component/Connector Based (CCB) Descriptions of Software Architecture to LQN Models

This chapter 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 PROGRES graph transformations. The descriptions of high level software architecture we use in this chapter is based on [Allen97] and we named it Component/ Connector Based (CCB) description. The performance model is based on the Layered Queueing Network (LQN) formalism, an extension of the well-known Queueing Network modelling technique [Woodside89] [Petriu91b] [Petriu94] [Woodside95a] [Woodside98] [Rolia87] [Rolia92] [Rolia95] [Frank95]. 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[Schuerr90].

The CCB to LQN transformation is the first version developed chronologically. Its

main disadvantage is that the language for architectural descriptions is non-standard while its advantage is that the notation is simple, easy to understand and contains only required information. The next version presented in chapter 4 uses UML to describe software architecture.

3.1 Component/Connector Based (CCB) Descriptions

As mentioned in chapter2, the high-level architecture describes the main system components and their interactions at a level of abstraction that captures certain characteristics relevant to performance, such as concurrency, parallelism, contention for software resources (as software servers and critical sections), synchronization, serialization, etc. The emerging discipline of software architectures is concerned with informal and formal ways of describing the overall system structure of complex software systems. In [Shaw96a] a perspective on this new discipline is presented, in [Shaw96b] and [Buschmann96] a number of high-level architectural patterns frequently used in today's software systems are identified and described, and in [Allen97] a formal foundation for software architectures based on architectural connections is introduced.

According to [Allen97], a software architecture represents a collection of computational components that perform certain functions, together with a collection of connectors that describe the interactions between components. A *component type* is described by a specification defining its functions, and a set of ports representing logical points of inter action between the component and its environment. *A connector type* is defined by a set of roles explaining the expected behavior of the interacting parties, and a glue specification

showing how the interactions are coordinated. On the other hand, an architectural pattern describes two inter-related aspects: its structure (what are the components) and behavior (how they interact). In the case of high-level architectural patterns, the components are usually concurrent entities that execute in different threads of control, compete for resources, and their interaction may require some synchronization. This kind of issues contribute to the performance characteristics of the system, and therefore must be captured in a performance model.

We defined an ad-hoc notation (Figure 3.1) for the description of high level software architecture based on the type of Component/Connector relationship, as in [Allen97] and the architectural patterns from [Shaw96b] and [Buschmann96]. We refer it as Component/ Connector Based (CCB) descriptions.

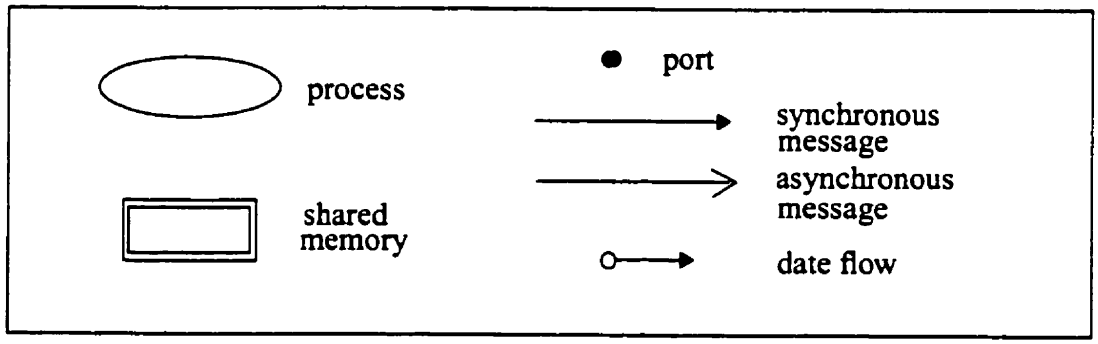


Figure 3.1: Notation for CCB description

In the notation, a *process* is represented by an ellipse, it can be any active component with its own thread of control. A *shared-memory* is represented by a double-bordered rectangle, it can be any passive component, e.g. a buffer. A port is represented by a solid dot, it defines a logical point of interaction between the component and its environment. Ar-

rows with different heads represents synchronous and asynchronous message respectively. A small arrow with a small circle attached indicates the data flow.

3.2 Some Frequently Used Architectural Patterns

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, Client-Server and Critical Section architecture. These patterns are frequently used to build distributed systems, and they present a variety of interactions between components. We will discuss each pattern more in details in the next several subsections.

3.2.1 Pipeline and Filter Pattern

The pipeline and filter pattern 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 [Buschmann96] 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. Both cases are shown in Figure 3.2: a) the filters communicate through an asynchronous mes-

sages, and b) the filters communicate through a shared buffer (one pushes and the other pulls).

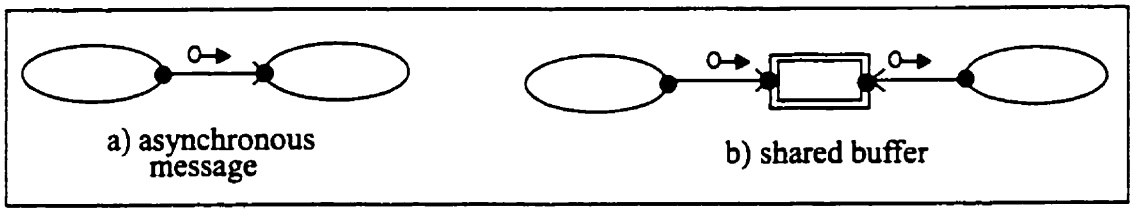


Figure 3.2: Communication styles in pipeline and filter pattern

3.2.2 Client-Server 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 [OMG92], 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, different alternatives are considered in the paper: direct client/server communication through a synchronous message and three types of connections mediated by brokers.

In the first case shown in Figure 3.3a, 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 [Adler95]. In the forwarding broker pattern from Figure 3.3b, 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 3.3c, 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 3.3d) 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 in this case.

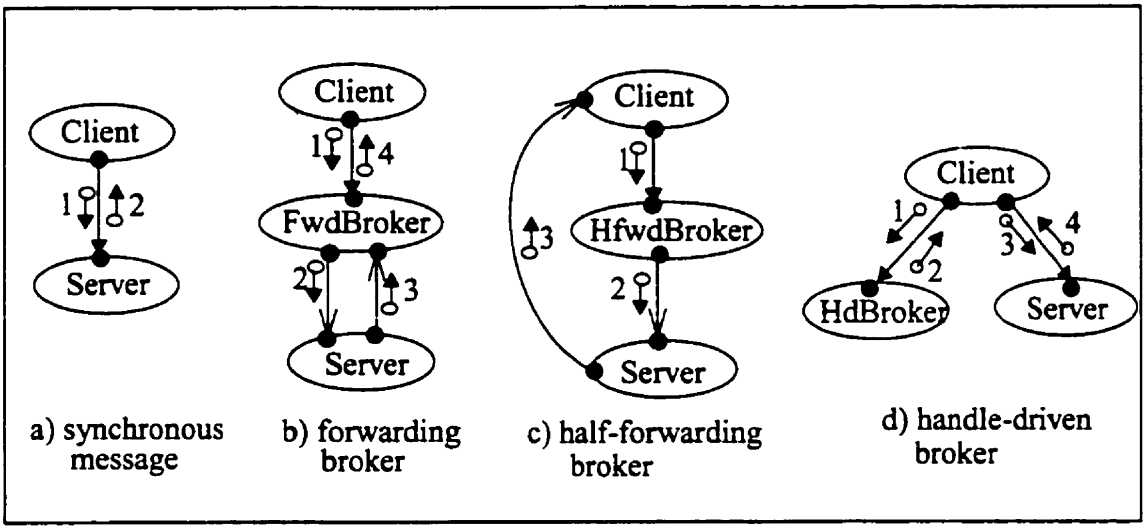


Figure 3.3: Communication styles in client-server pattern

3.2.3 Critical Section Pattern

The Critical Section pattern is composed of a number of processes that share a common data stored in shared memory (see Figure 3.4) 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.

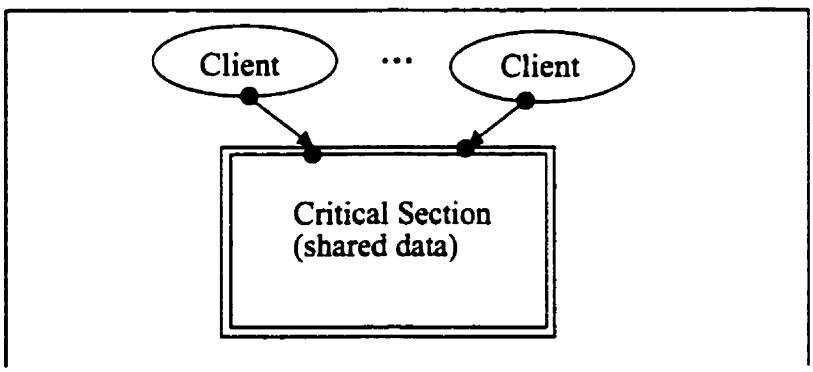


Figure 3.4: Critical section pattern

3.3 More On Connector Types

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 [Allen97], which is the basis for the graph grammar representation of software architectures proposed in the next section.

According to [Allen97], 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 behavior of the interacting parties, and the glue shows how the interactions are coordinated. The connector specification is formally described in [Allen97] with a subset of Hoare's process algebra.

For example, in a CS pattern with CORBA interface (see Figure 3.5) 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 3.3b, 3.3c and 3.3d 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 and three using the services of a broker. Figure 3.2b illustrates another example of connector type that con-

tains three roles (two filters and a shared buffer). Its glue describes the "push" and "pull" operations and the constraints for correct behavior (as for example "cannot pull data from an empty buffer", "cannot read and write to the buffer at the same time", etc.)

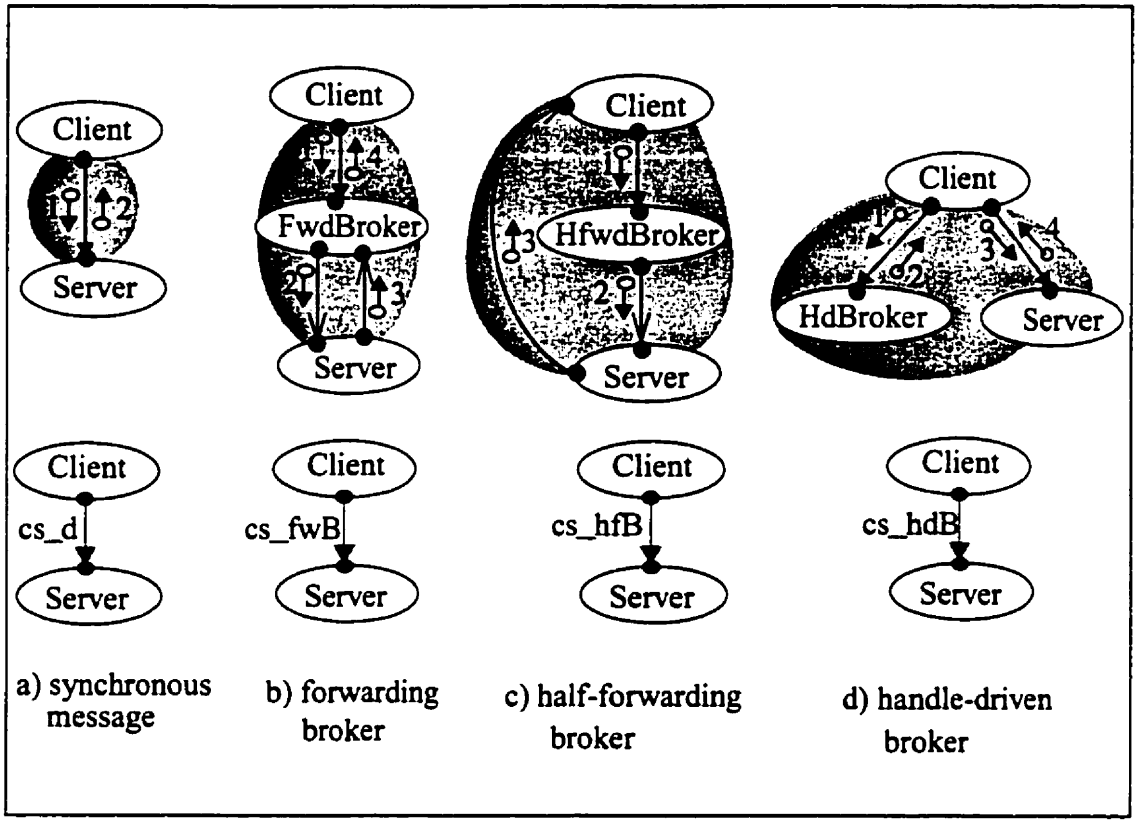


Figure 3.5: Client-server connector types

In our work, we first identified the connector types associated to different architectural patterns, then defined graph transformation rules to transform each connector to an LQN submodel.

3.4 PROGRES Graph Schema

As mentioned in chapter 2, a feature of the PROGRES language is to define the static properties of graphs in the form of a *graph schema* which is conceptually similar to a database schema.

The graph schema for the transformation from CCB to LQN defines the types of *nodes* and *edges* allowed in an input graph (CCB description), an output graph (LQN model) and an intermediary graph (a combination of both input and output graphs).(see Figure 3.6) 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). In order to accommodate graphs in intermediary transformation 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 transforming server-to-server CS connections which appear in tiered client/server systems. Such edges are illegal in both the input and output schemas; they are generated and then deleted during a two-step transformation process that is presented later in the paper.

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 "*criticalSection*" 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 different edge type (*cs_d, cs_fwB, cs_hfB and cs_hdB*, respectively).

A note-worthy fact is 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, a broker has an important impact on the system performance, so it is explicitly modelled in LQN.

The LQN graph notation presented in chapter 2 and illustrated in Figure 2.2 has *"task"* nodes, which are described by the 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); 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.

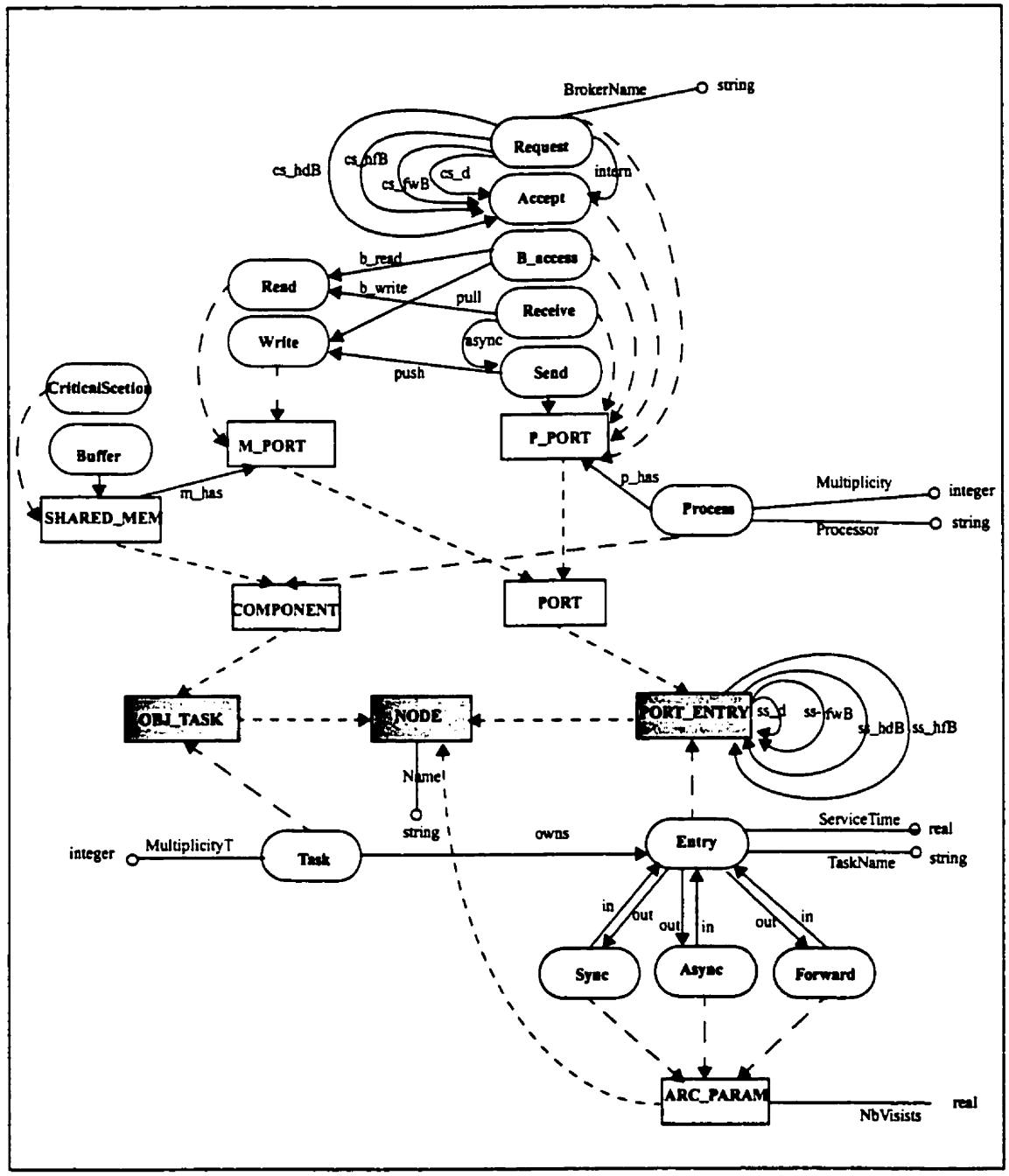


Figure 3.6: Joint graph schema for the CCB description and LQN model

3.5 Transformations from Architectural Patterns to LQN

We have defined transformation rules for each architectural connection type, as illustrated in Figures 3.7 to 3.15. In order to convey the principle of these transformations, they are given in a more intuitive, higher-level graphical notation tailored to our problem domain, rather than in the more detailed PROGRES notation. The PROGRES transformation process executes a transaction for every connection instance found in the input architectural description graph. The transformation process ends when all architectural connections have been processed. As expected, the performance of the system depends on the performance attributes of its components and on their interaction. Performance attributes are not central to the software architecture itself, but must be specified by the user in order to transform the architecture into a performance model. Such attributes describe the demands for hardware resources by the software components: allocation of processes to processors, average execution time for each software component, average demands for other resources such as I/O devices, communication networks, etc.The final result is an LQN model that can be written to a file according to a predefined LQN model format [Franks95]. We will discuss more about the control structure of the whole transformation process in section 3.6. Some general transformation approaches are also given in the next subsection.

3.5.1 General Transformation Principles

The general principles of the transformation from CCB description to LQN model are as follows:

- 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 seems 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 two-step transformation process of server-to-server connections, as illustrated in Figure3.15.

3.5.2 Pipeline and Filter Pattern

Figures 3.7, 3.8 and 3.9 show the transformation of the pipeline and filters connection types, the first using an asynchronous messages and the other two a shared buffer. A regular arrow with a solid line in the figures represents a synchronous request in an LQN model. A half arrow represents an asynchronous request and an arrow with a dotted line represents a forwarding request. We use these arrows in the LQN models throughout the thesis.The transformation 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 Figures 3.8 and 3.9 is exactly the same, the difference in the allocation of processes to processors leads to quite different LQN submodels.

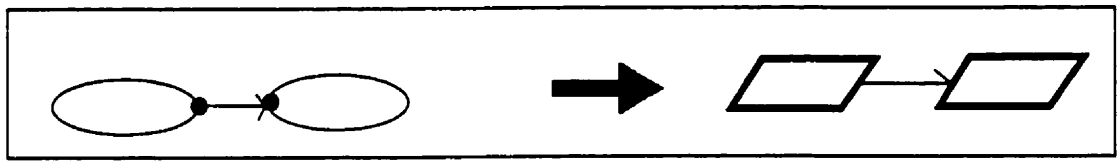


Figure 3.7: Transformation of a PF connection by an asynchronous message to a LQN model with asynchronous request

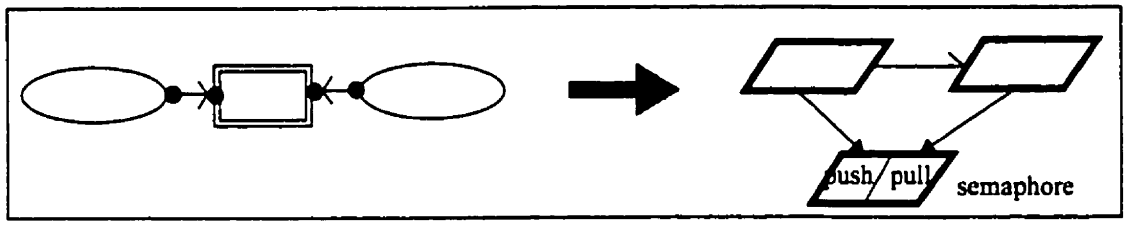


Figure 3.8: Transformation of a PF connection by a shared buffer, where the filter processes are running on the same processor

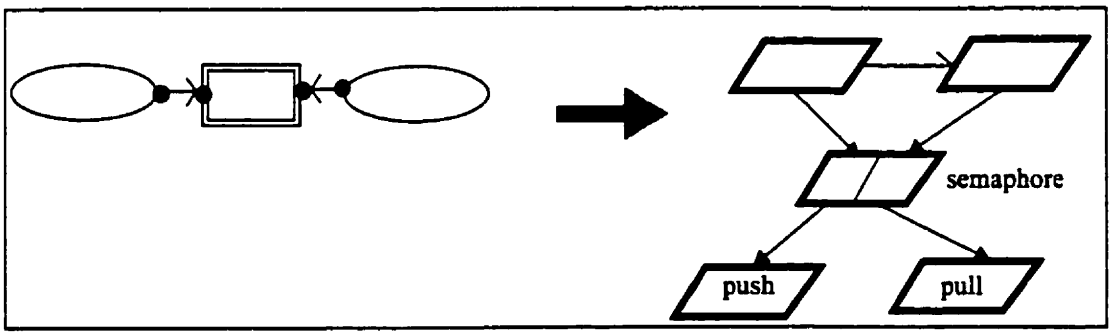


Figure 3.9: Transformation of a PF connection by shared buffer, where the filter processes are running on different processors

3.5.3 Client-Server Pattern

Figures 3.10, 3.11 and 3.12 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 half 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 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 Figure3.11, 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.

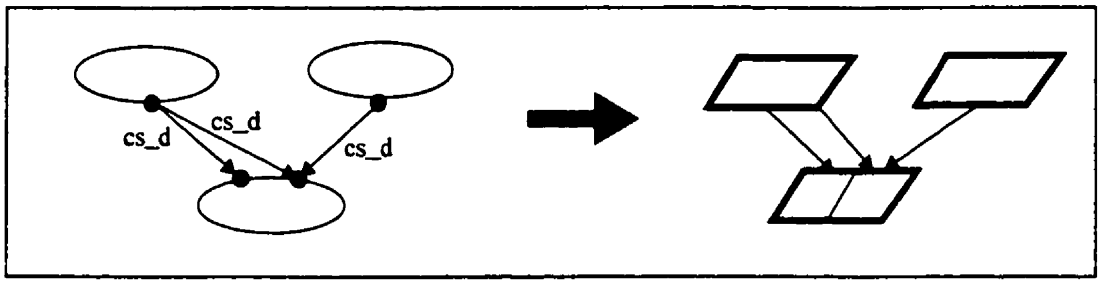


Figure 3.10: Transformation of three direct CS connection instances to LQN (each service offered by the server is represented by an entry)

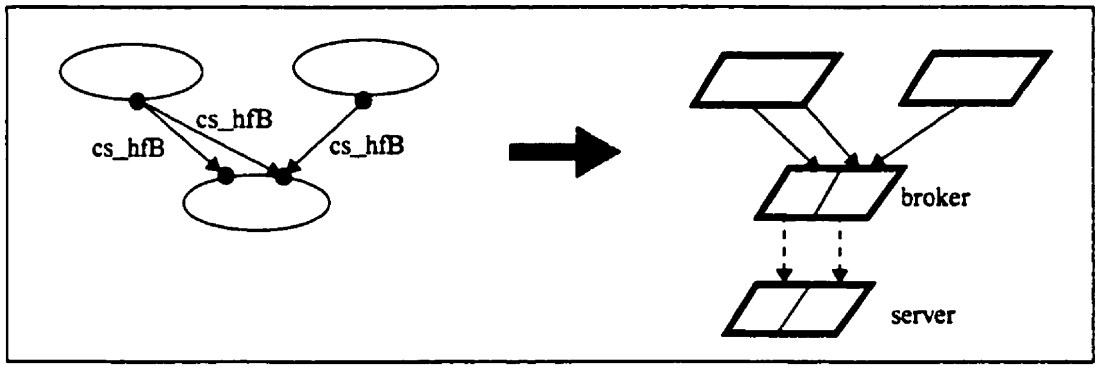


Figure 3.11: Transformation of three CS connection instances by half forwarding broker to an LQN model with forwarding arcs

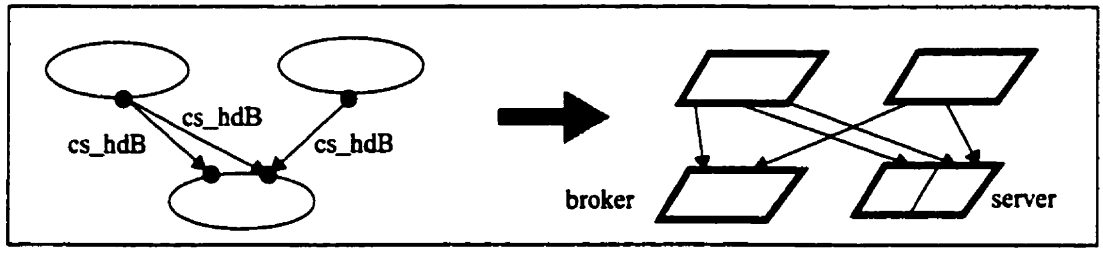


Figure 3.12: Transformation of three CS connection instances by handle-driven broker to LQN

3.5.4 Critical Section Pattern

Figures 3.13 and 3.14 show the transformation for the Critical Section connection type. The performance model capture the serialization delays introduced by the constraint that the common data should be accessed by one client at a time. Similar to the pipeline by 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.

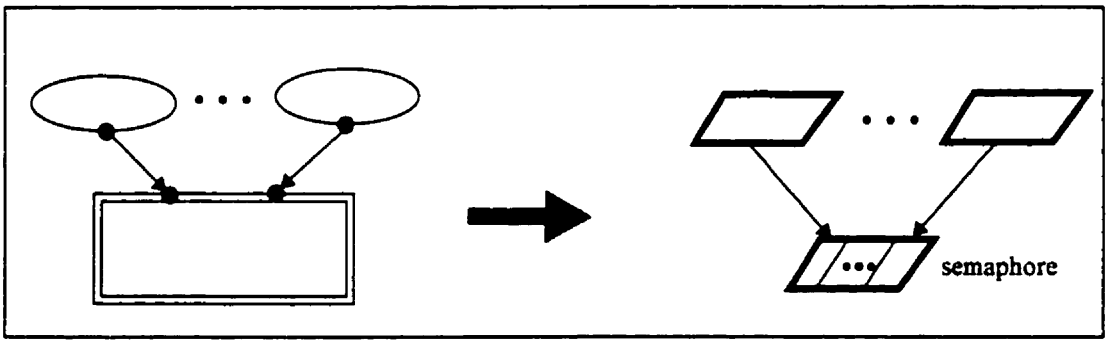


Figure 3.13: Transformation of a critical section pattern, where the client processes are running on the same processor

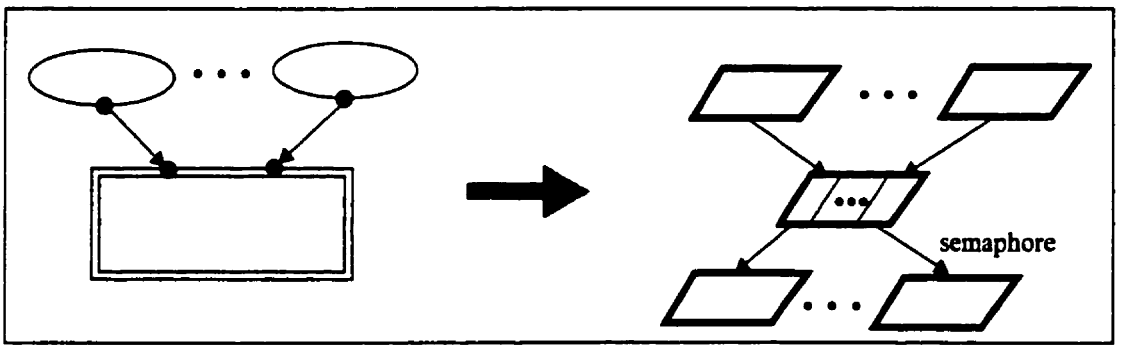


Figure 3.14: Transformation of a critical section pattern, where the client processes are running on different processors

3.5.5 Layered Client-Server

Figure 3.15 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 layered 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 its client and server roles (as shown in Figure 3.15). The upper part of the figure shows the two-step transformation 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 first step. 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 generating a connection edge of an intermediary type, as explained before. In the second transformation step, each CS connection between the two server is completely transformed to LQN, depending on the connection type (direct or through a broker).

Also in this case as a layered client/server system, some server nodes play both the role of server to its own clients, and that of a client to the servers below. This introduces an additional step in the transformation of server-to-server CS connections (as opposed to client-to-server connection). Firstly, the internal mapping between the input and output ports

of the upper server is used to generate an appropriate number of CS request edges (see Figure3.15.). These edges are of an intermediary type (ss_d, ss_fwB, ss_hfB, or ss_hdB) and remember the original CS connection type (direct or through a broker). Secondly, the transformation process is completed for each connection represented by an intermediary edge as it were a normal client-to-server connection.

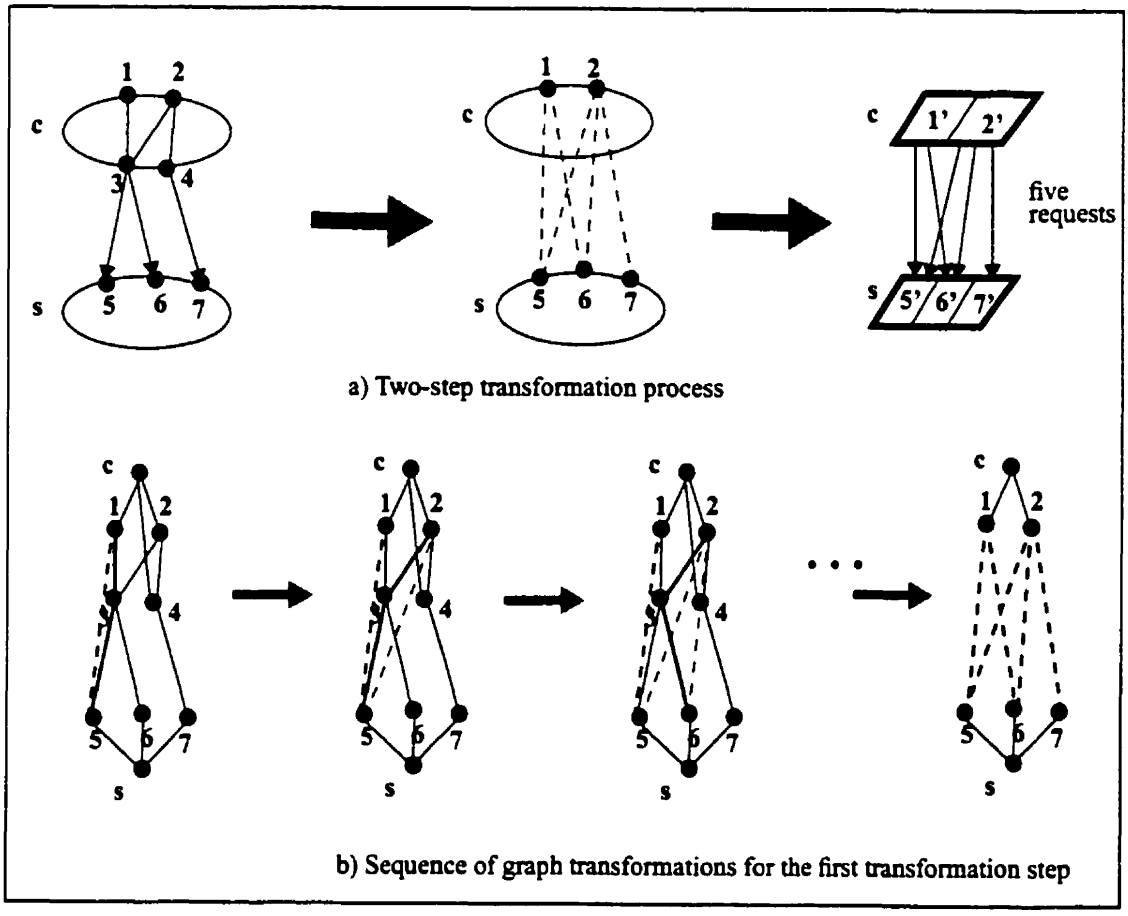


Figure 3.15: Two-step transformation process of server-to server CS direct connections

3.6 Control Structure for Graph Transformation

A software system contains many components involved in various architectural connection instances, and a component (such as 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. PROGRES searches for subgraphs in the underlying input graph that has an one-to-one correspondence with the given pattern. The transformation 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 the predefined LQN model format [Franks95].

The following shows some code from the first version of PROGRES program as an example of transformation from CCB descriptions to LQN model.

```
transaction MAIN=
    use swp1, swp2: Process
    do
        Create3AsyncPipeline ("aaa", "bbb", "ccc", "ddd")
      & AddAsyncPipeline ("ddd", "eee")
      & InsertBufferToAsyncPipeline ("aaa", "bbb", "buf")
      & AddForwardCS ("ccc", "Server", "service1")
      & AddRequestForwardCS ("ccc", "Server", "service2")
      & AddExistingClientForwardCS ("ddd", "Server",
        "service2")
      & TransformAllAsyncPipeline
      & TransformAllAsyncPipelineWithBuffer
      & TransformAllLayeredDirectCS
      & TransformAllLayeredForwardCS
      & TransformAllLayeredHandleDrivenCS
      & TransformAllDirectCS
      & TransformAllForwardCS
      & TransformAllHandleDrivenCS
      & TransformAllCriticalSection
```

```
      & Clean
    end
  end;

transaction TransformAllAsyncPipeline =
    loop
      TransformAsyncPipeline
    end
  end;

......
```

Figure 3.16: An example of a whole transformation process

The piece of program showed in Figure 3.16 does the followings:

- First of all, the PROGRES program creates a input graph from scratch according to CCB description (as in Figure 3.1 to 3.4) using *productions*.
- And then it performs transformation of each pattern in a loop like the second transaction *TransformAllasyncPipeline* in Figure 3.16. Because there maybe more than one match for the particular pattern.
- Then it repeats the transformation pattern by pattern until there is no match of any pattern is found.
- At the last stage of the program will clean all the intermediate nodes and merge all duplicated nodes.

Intermediate nodes like *COMTASK* are used in the process of graph transformation and all of them are transformed to LQN Tasks at the last stage of transformation.

Duplicated nodes can be generated by partial transformation. (e.g. in the transformation of brokers in client-server connection) Actually the transformation of brokers in client-server connection is worth a little discussion. One approach of the transformation is that generates a new broker task and service entry every time a client-server connection with brokers is found. This will lead to a result of duplicated broker tasks and also some duplicated entries. Duplicated tasks and entries can be recognized if they have the same name (and also taskname for entries) and they can be merged together at the last stage of the transformation.

Another approach is that during the transformation of client-server connection with brokers, always check the graph to see if there is already a broker task. If not, add the broker task, otherwise check to see if the desired service entry exists. If not, add the service entry and the request arc, otherwise, only add the request arc. Since a lot of checking (e.g. *tests and queries* (more complicated tests)) must be done in this case, the transformation is more complicated. We choose to use the first approach in the transformation in the thesis.

The order of patterns transformed in the transformation does matter in some cases when the transformations are related. For example, in the case of layered client-server systems, we have to deal first with the layered client-server transformation (as in Figure 3.15), and then the client-server transformation. However the other patterns which are not related to layer structures can be done in any order.

The whole transformation process can be illustrated in the following flow chart.

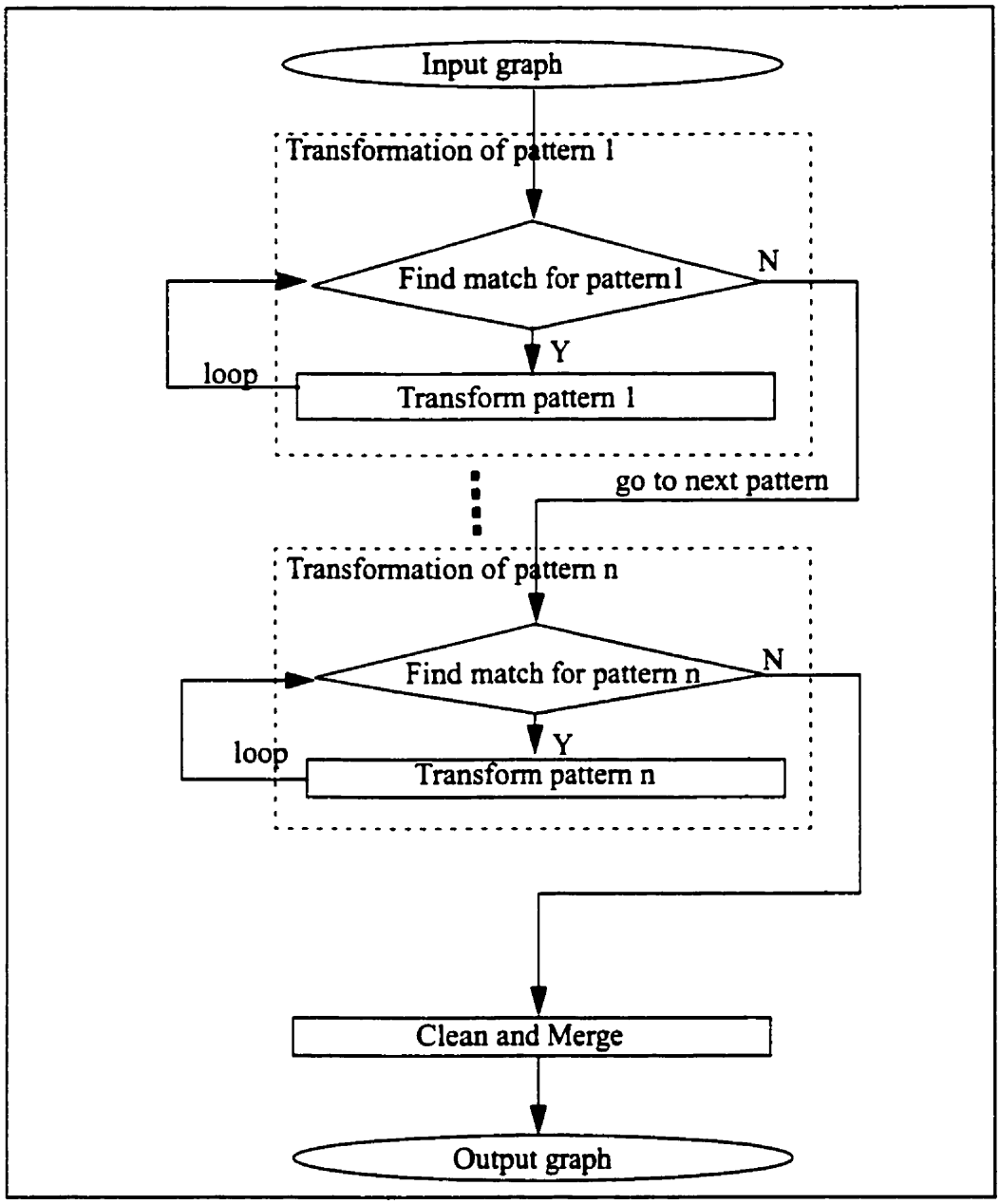


Figure 3.17: Flow chart of transformation control structure

Due to space limitation, we will not discuss in details the graph *production rules* for the CCB description to LQN model transformation. However, such a detailed discussion

will be presented in the next chapter for the UML description of software architecture to LQN performance model transformation.

CHAPTER 4

From UML Descriptions of Software Architecture to LQN Models

This chapter proposes a formal approach to build Layered Queueing Network (LQN) performance models from UML descriptions of the high-level architecture of a system, and more exactly from the architectural patterns used in the system. It is the second version of the implementation. The difference between this version and the first version in chapter 2 is that we are using UML descriptions of the high-level architecture of a system in this version. The main advantage of using Unified Modelling Language (UML) is the fact that UML is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. The disadvantage is that UML is a rich notation with many types of diagrams which contains much more information than that required for developing a performance model, which means that a selection process is necessary to decide what to retain and what to ignore. On the other hand some performance specific informations (such as resource demands) is missing in UML and has to be pro-

vided by the user in order to build the performance model. The transformation from UML architectural description of a given system to its LQN model is implemented by using the PROGRES tool [Schuerr90].

4.1 Architectural Patterns and UML Collaborations

As mentioned in chapter2, architectural patterns express fundamental structure organization schemas for software systems [Buschmann96]. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them. Architectural patterns help software developers to specify the fundamental structure of an application.

This chapter proposes to use high-level architectural patterns, described in UML, as a basis for transforming a software architecture into a performance model. A subset of frequently used patterns are described in this chapter in the form of UML collaborations. Some of them have already been described and transformed to LQN models using CCB description in the previous chapter.

As already mentioned, UML is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems [Rational]. It becomes more and more popular in industry. That is why we considered using UML to describe the high level software architecture after we did the similar research using an ad-hoc notation named CCB descriptions.

UML is very rich when it comes to format. It has a number of different views and diagrams to fit the different purposes of the designers. Since we need to describe architectural patterns which capture relationships at the subsystem level [Shaw96b] and are a further step up in granularity from the class and object level, we chose to use a UML feature named *collaboration* which is better suited to describe the software architectural patterns for the purpose of the thesis.

A UML collaboration is not a UML collaboration diagram (which is a type of interaction diagrams) [Booch99]. According to the authors of UML, a collaboration is a notation for describing a mechanism or pattern, which represents "a society of classes, interface, and other elements that work together to provide some cooperative behavior that is bigger than the sum of all of its parts." [Booch99] A collaboration has two aspects: structural (usually represented by a class/object diagram) and behavioral (an interaction diagram). Collaborations can be used to hide details that are irrelevant at a certain level of abstraction; these details can be observed by "zooming" into the collaboration. The symbol for collaboration is an ellipse with dashed lines, and may have an "embedded" square showing template classes. (as in Figure 4.1 to 4.3) Another special UML notation employed in this chapter is that of an active class (object) which has its own thread of control, represented by a square with thick lines. An active object may be implemented either as a process (identified by the stereotype <<process>>), or as a thread.

Figure 4.1 through 4.3 illustrates UML collaborations for client-broker-server pattern. They are used as an example to illustrate how UML collaborations describe high level

software architectural patterns. The UML collaboration of other patterns will be illustrated in later sections along with their transformation to LQN models.

Figure 4.1 illustrates the UML collaboration (both the structural and behavioral parts) for the forwarding broker pattern [Adebayo97], where 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 excessive network traffic when the client, broker and server reside on different nodes. An alternative that reduces the network traffic is the half-forwarding broker [Adebayo97] from Figure 4.2, 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 (such as load balancing and centralized recovery from failure). A handle-driven [Adebayo97] broker (as in Figure 4.3) 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 in this case. The small arrows with a small circle in Figure 4.1,4.2 and 4.3 indicate the data flow.

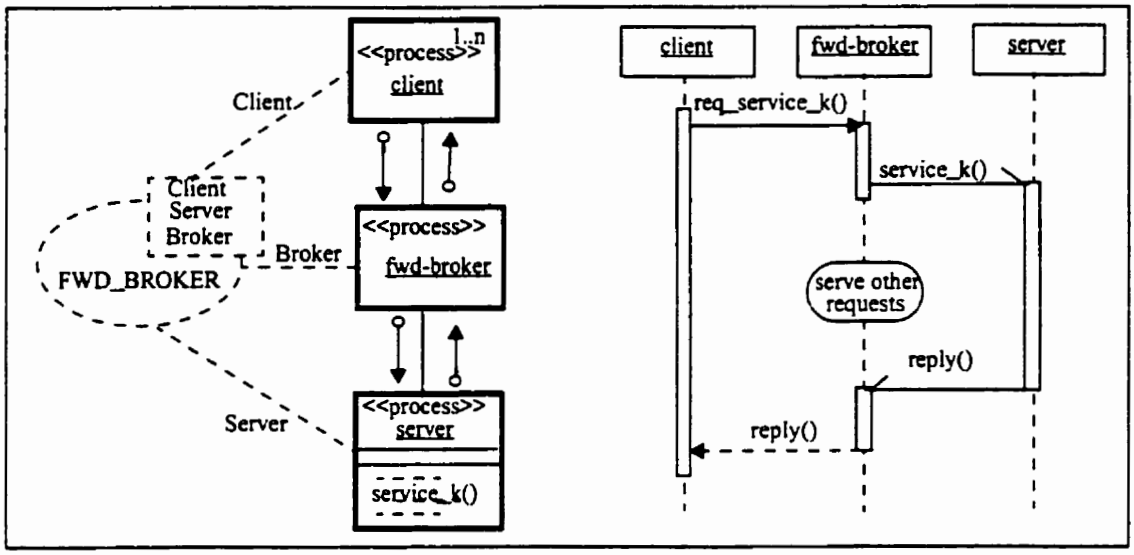


Figure 4.1: UML collaboration for client-server pattern with a forwarding broker

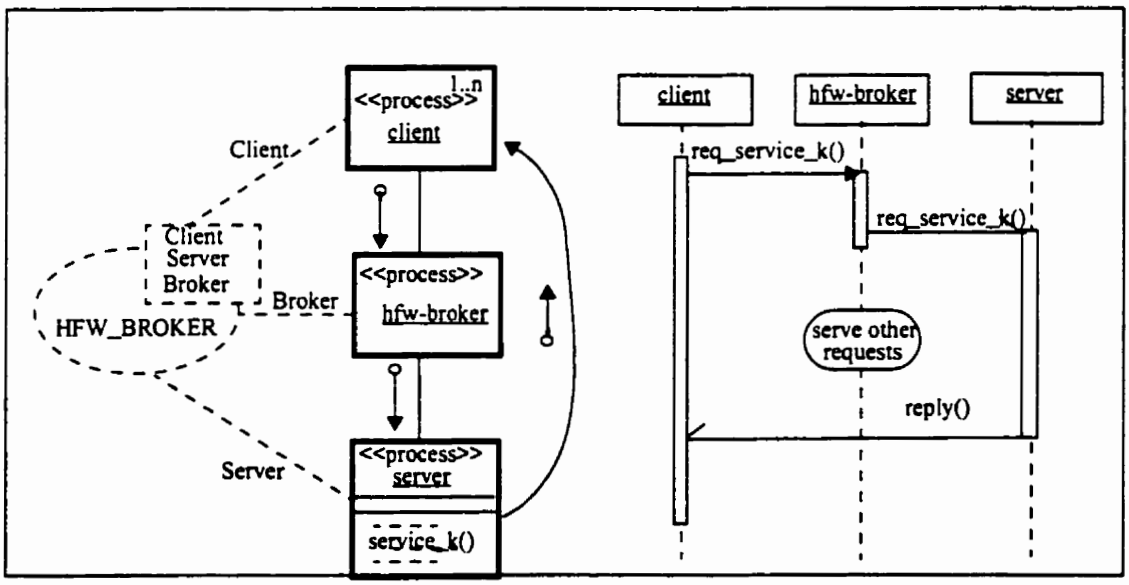


Figure 4.2: UML collaboration for client-server pattern with a half-forwarding broker

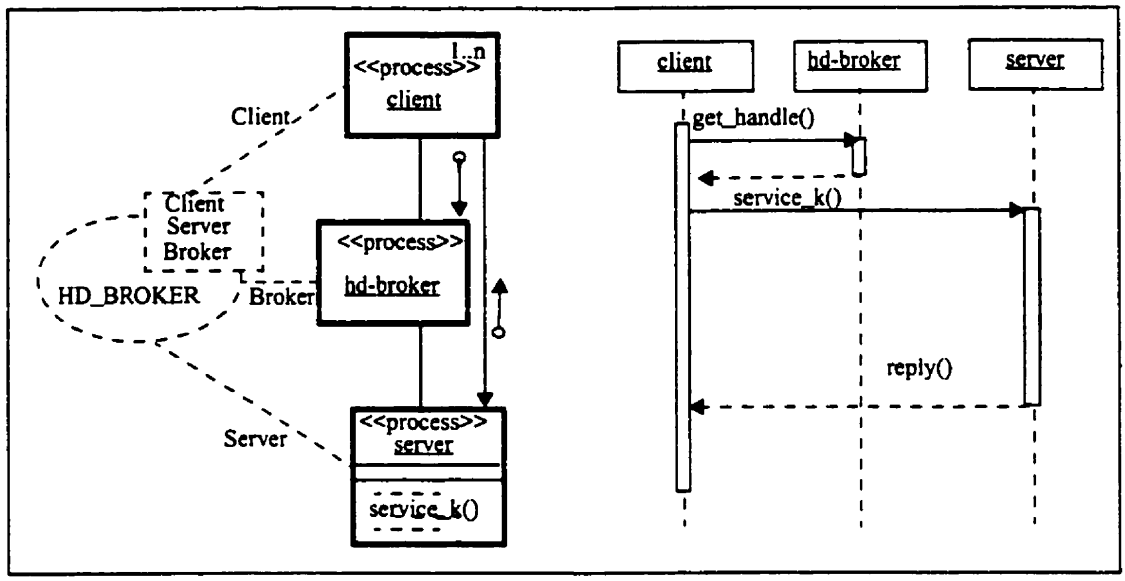


Figure 4.3: UML collaboration for client-server pattern with a handle-driven broker

4.2 Scope of the Thesis Research

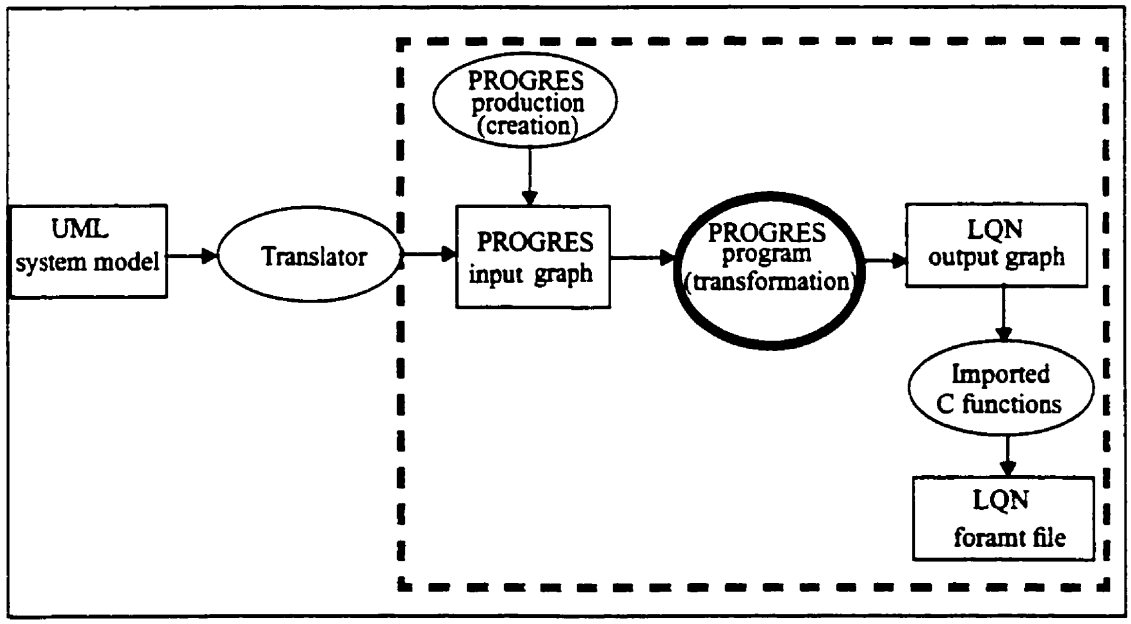


Figure 4.4: Scope of the thesis research

Figure 4.4 illustrates the scope of the proposed approach and the part we implemented using PROGRES, which is included in the grey dotted rectangle. First, the PROGRES graph for a given system according to the schema from Figure 4.5 is directly generated by a PROGRES program, then the input graph is transformed by applying the appropriate rules, into an output graph representing an LQN model. An LQN model file that can be fed directly to an LQN solver is also generated by imported functions written in C to the PROGRES program.

It is necessary to mention that the thesis does not solve the problem of converting UML notation to a PROGRES input graph. (like the part outside the grey dotted rectangle in Figure 4.4) The transformation of UML diagrams to a PROGRES input graph must be done in the future in order to integrate the generation of LQN models into an UML CASE tool.

4.3 PROGRES Graph Schema

The graph schema for the transformation from UML collaboration to LQN defines the types of nodes and edges allowed in an input graph (architectural description), an output graph (LQN model) and an intermediary graph. (a combination of both input and output graphs, see Figure 4.5) The upper part of the figure contains the input schema for architectural descriptions and the lower part contains the output schema for LQN models (light-gray nodes). The input schema does not capture all the richness of UML, but only those elements that are necessary for converting a high-level architecture into an LQN model.

The advantage of basing the transformation on architectural patterns expressed by UML collaborations is that such higher-level of abstraction artifacts greatly simplify the

graph schema and the transformation process. The disadvantage is that these artifacts have to be pre-identified and represented in the schema and in the transformation rules, which limits the extensibility of the transformation process. This disadvantage is somehow mitigated by the fact that the number of high-level architectural patterns identified in literature and used in practice is relatively small.

In order to accommodate graphs in intermediary transformation stages, the two schemas are joined together by three nodes illustrated in dark-gray at the base of the node class hierarchy (*NODE*, *OBJ_TASK*, and *OP_ENTRY*). The collaborations nodes representing architectural patterns make up a big part of the input schema. Inheritance is useful for classifying the different patterns and their variants. "Role" edges, like *client or server*, connect the collaboration nodes to the architectural *component* nodes, which are *active* and *passive* objects, their *operations* and links. The output schema reflects closely the LQN graph notation presented in chapter 2. The node types are *task* and *entry*. The LQN arcs may represent three types of requests (*synchronous*, *asynchronous* and *forwarding*); a parameter indicates the average number of visits associated with each request. Since PROGRES edges cannot have attributes, we represent an LQN arc by three elements: an incoming edge, a node carrying the parameter (*ARC_PARAM*) and an outgoing edge.

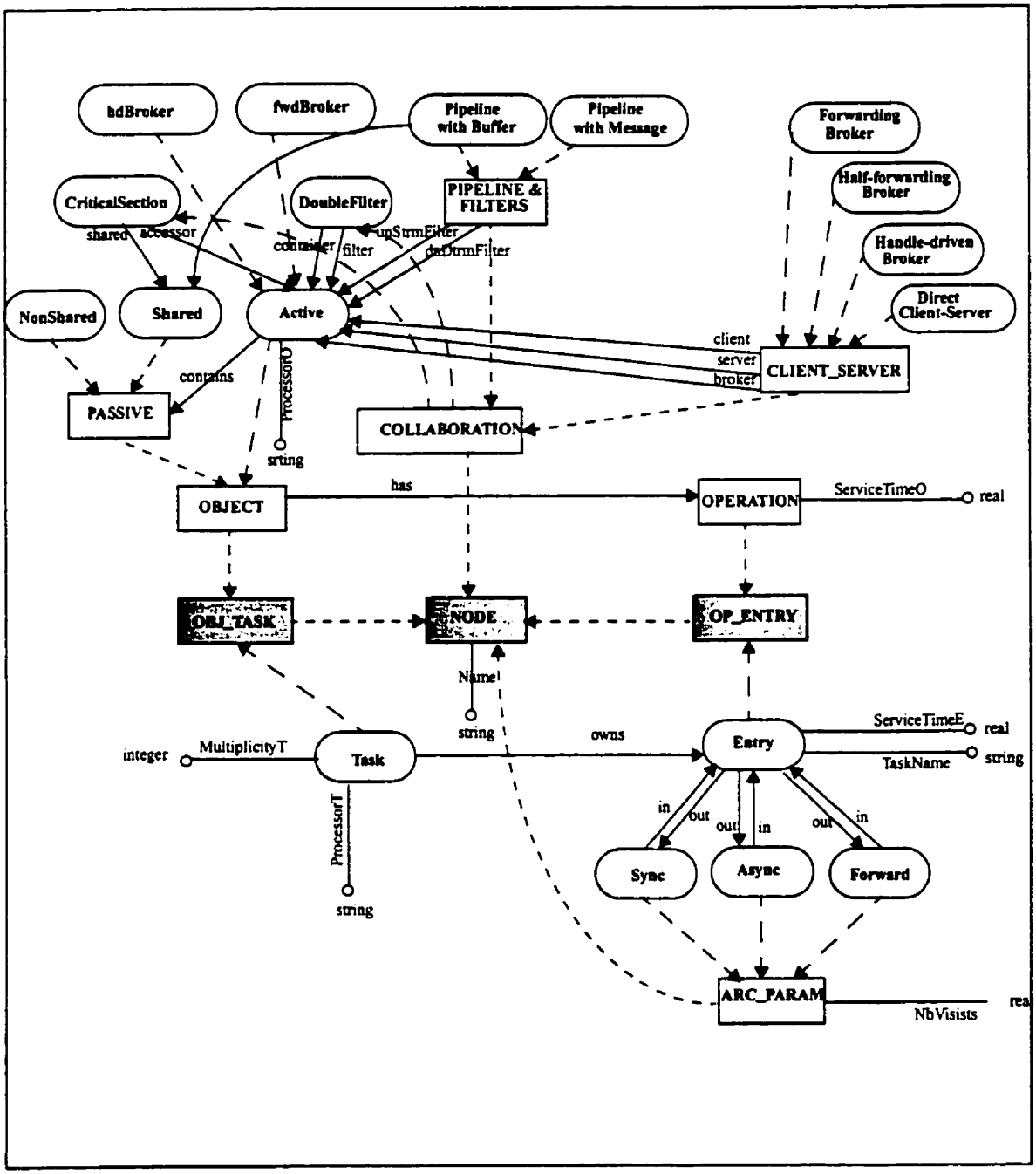


Figure 4.5: Joint graph schema for architectural pattern and LQN models

4.4 Transformations from Architectural Patterns to LQN

Graph transformation rules have been defined for each architectural pattern, following closely the transformations illustrated in section 4.4.2 through 4.4.5. A PROGRES transaction is executed for every architectural pattern found in the input architectural description graph. The transformation process ends when all the patterns have been processed.

As expected, the performance of the system depends on the performance attributes of its components and on their interaction. Performance attributes are not central to the software architecture itself, but must be specified by the user in order to transform the architecture into a performance model. Such attributes describe the demands for hardware resources by the software components: allocation of processes to processors, average execution time for each software component, average demands for other resources such as I/O devices, communication networks, etc. In the early design stages of a system, these value can be either estimated from previous experience with similar systems or can be based on desired "*time budgets*" allocated to various subsystems. In the case of reusable components, the resource demand values can be based on actual measurements.

4.4.1 General Transformation Principles

The general transformation principles from an input to an output graph are as follows:

- Each architectural component (i.e., object) is converted to an LQN task, for which reason a common base class *OBJ_TASK* was defined in the graph schema. However, the correspondence between components and tasks is not bijective, as in some cases a single object may generate more than one task for the following reasons: to charge cor-

rectly the execution times to various processors (Figure 4.7b and 4.21b), or to model processes that are part of the underlying midware (such as brokers in Figure 4.14, 4.15, and 4.18).

- Object operations are usually converted into an LQN entry, with some exceptions as in Figure 4.7b and 4.21b, when an operation is converted into an entry and a task.
- The collaboration nodes from the input graph do not have an LQN equivalent. They may be remained in the intermediary graph, but will be removed in clean-up procedure.

4.4.2 Pipeline and Filter Pattern

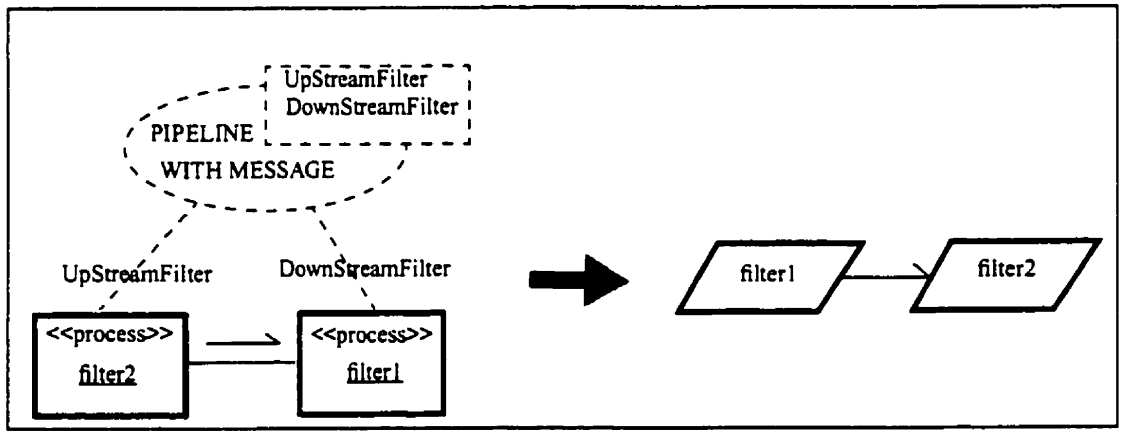


Figure 4.6: Transformation of pipeline and filter pattern with a message

Figures 4.6 and 4.7 illustrate the transformation of two versions of Pipeline and Filters pattern, the first using asynchronous messages for the pipeline, and the other using a shared buffer. A regular arrow with a solid line in the figure represents a synchronous request in LQN model. A half arrow represents an asynchronous request and a dotted line

arrow represents a forwarding request. We use these arrows in the LQN models throughout the thesis. Each active filter becomes an LQN task whose service time includes the processing time of the filter. The pipeline connector is modelled as an asynchronous LQN request in Figure. 4.6. The CPU times for send/receive system calls are added to the service times of the two LQN tasks, respectively. A network delay for the message can be represented in LQN as a delay attached to the arc.

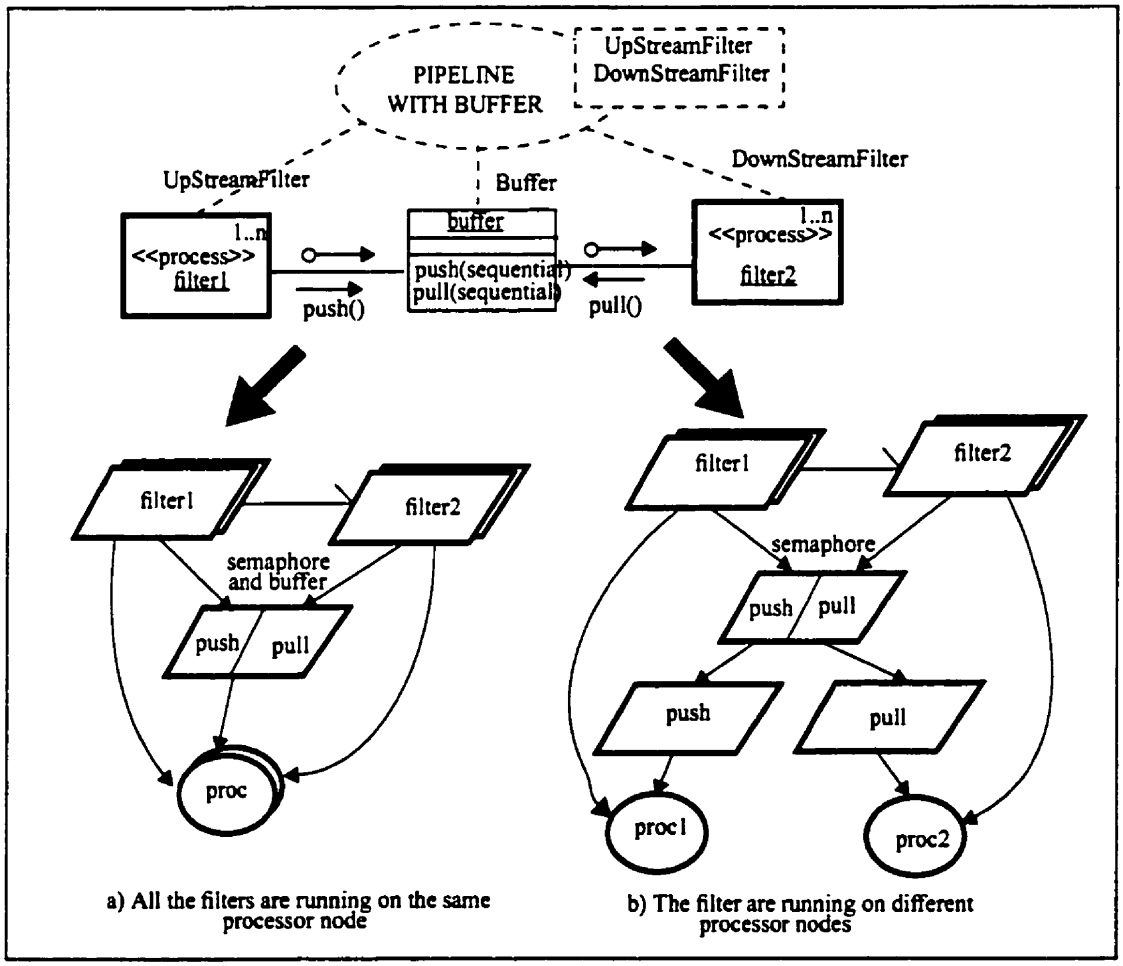


Figure 4.7: Transformation of pipeline and filter pattern with a buffer

In the case of a pipeline with buffer (see Figure 4.7), an asynchronous LQN arc is still

required, but this does not take into account the serialization delay due to the constraint that buffer operations must be mutually exclusive. A third task will enforce this constraint. It has as many entries as the number of operations executed by the tasks accessing the buffer (two in this case, push and pull). In Figure 4.7, exactly the same architectural pattern has two LQN counterparts, due to a difference in processor allocation. The execution of all buffer operations is charged to the same processor node in Figure 4.7a, and to different processor nodes in Figure 4.7b.

The following fragments of PROGRES code (Figure 4.8 and 4.9) illustrate the graph production rules for the two variants of the pattern. The application of a graph production rule in PROGRES performs first the selection of a subgraph that is matching the left-hand side of the rule and then the replacement of this subgraph by the right-hand side of the rule. The rule shows the details of the transformation implemented (i.e. the *nodes and edges* that are kept, added or removed, as well as the changes of the *attributes*. In this program the left-hand side of a rule corresponds to an architectural pattern from the input graph and the right-hand side to the LQN submodel for this pattern.

In the case of pipeline with message, the left-hand side of the graph consists a *PPWithMsg* node, which is derived from *COLLABORATION* class (node 3), two *Active* Object nodes (node 1 and 2) and two edges *upStrmFilter* and *dnStrmFilter* that indicate the roles of the objects.

The transformation is pretty straightforward. In the right-hand side, the *PPWithMsg* node does not exist any more because a collaboration node does not have an LQN equiva-

lent. The two *Active* object remain the same, because they may be connected by other *COLLABORATION* nodes too. They will be transformed to LQN *tasks* at the last stage of transformation after all the collaborations are found and transformed. A new *Async* node (node 4) is added representing the attributes of a asynchronous request from node 1 to node 2 in the LQN model. The *transfer* keyword indicates the modifications on attributes of the right-hand side.

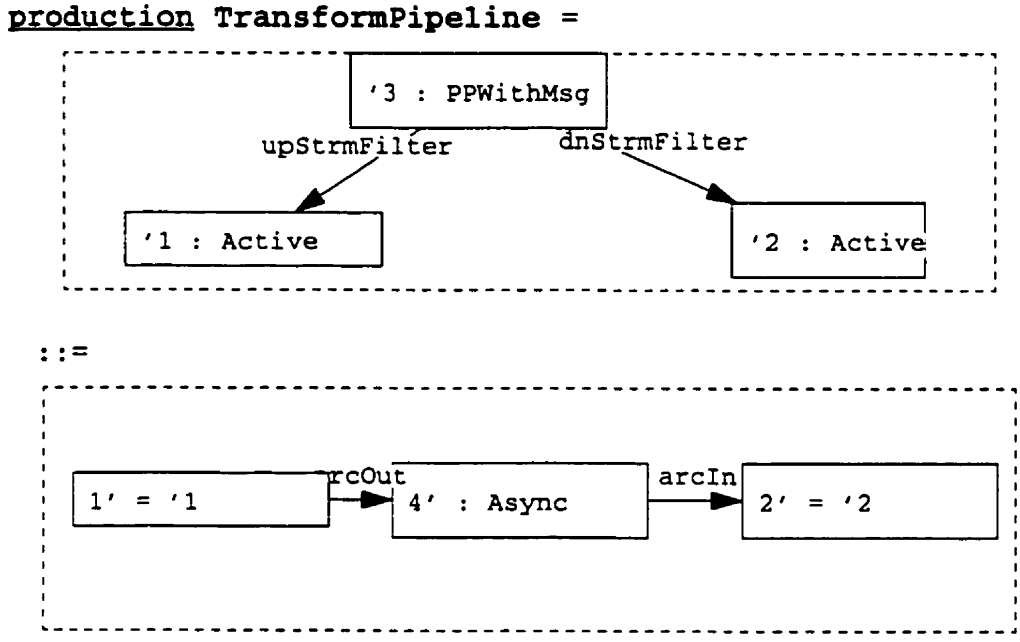


Figure 4.8: Production rule for pipeline and filters pattern with a message

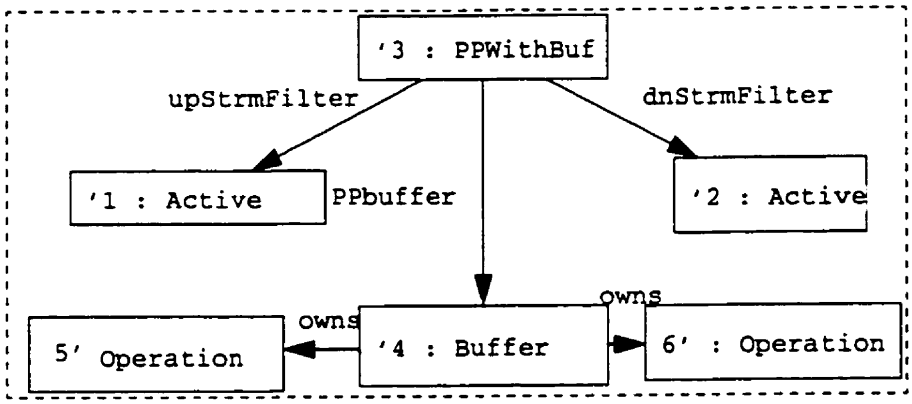
In the case of pipeline with buffer (as in Figure 4.7a), where both filters are located on the same processor, the left-hand side of the graph consists a *PPWithBuf* node (node 3),

two *Active* Object nodes (node 1 and 2), one *Buffer node* (node 4) with two *Operations* (node 5 and 6) and three edges *upStrmFilter, dnStrmFilter* and *buffer* that indicate the roles of the objects.

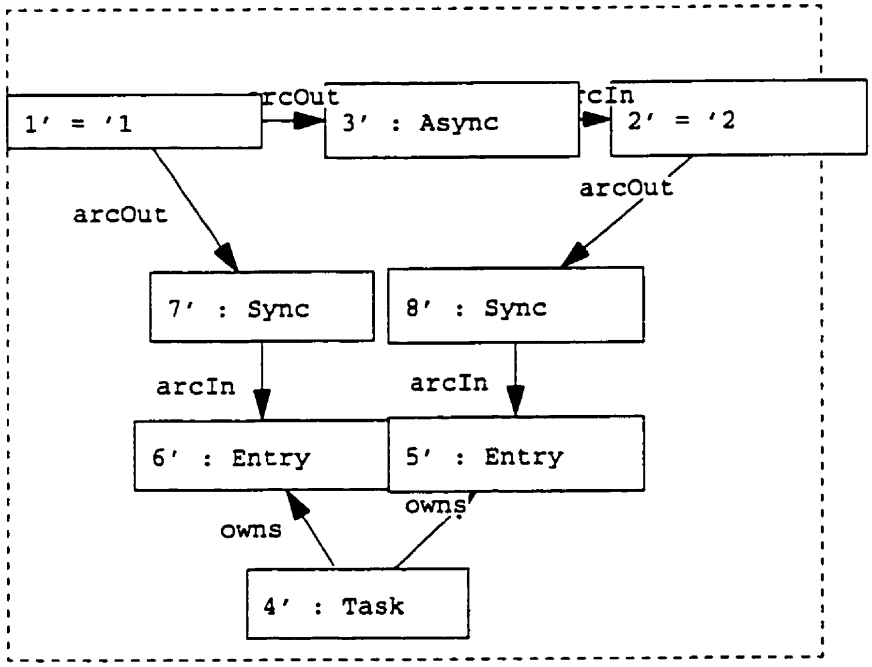
In the right-hand side, apart from the addition of an *Asnyc* node (as in Figure 4.8), a new *task* node (node 4) is added representing the buffer in the left-hand side. The two new *entries* (node 5 and 6)represent the two mutually exclusive operations (node 5 and 6 in LHS). Two new *Sync* node (node 7 and 8) are added representing the attributes of two synchronous requests in LQN model. The *transfer* keyword below the graph indicates the modifications of the attributes of the right-hand side. For example, the *serviceTimeO* of *Operation* node 5 and 6 in the LHS become the *serviceTimeE* of *Entry* node 5 and 6 respectively.

It is worth to mention that the *condition* keyword in this production indicates an additional condition on any subgraph that matches the LHS in order for the production to be executed. In this case the condition is both *Active* object must locate on the same processor. For the other case (as in Figure 4.7b, where the filters are located on different processors), the LHS of the production rule is similar except it has a different *condition* which means that the two *Active* object must locate on different processors. The RHS of the rule is bit different too. The tasks will be located on different processors. Since we do not have a *Processor* node in the graph schema, we use the *Processor* attribute instead. So, the *Processor* attributes of the tasks will be different.

production TransformPipelineWithBuf =



::=




```
condition 1'.processor := '2.processor;
transfer 4'.name := '4.name;
          6'.name := '6.name;
          5'.name := '5.name;
          7'.fromName := '1.name;
          7'.toName := '6.name;
          8'.fromName := '2.name;
          8'.toName := '5.name;
          3'.fromName := '1.name;
          3'.toName := '2.name;
          3'.Type := "a";
```

```
            7'.Type := "y";
            8'.Type := "y";
            5'.TaskName := '4.name;
            6'.TaskName := '4.name;
           5'. ServiceTimeE:= '5.ServiceTimeO;
           6'. ServiceTimeE:= '6.ServiceTimeO;
end;
```

Figure 4.9: Production rule for pipeline and filters pattern with a buffer

4.4.3 Double Filter Collaboration

Figure 4.10 represents the transformation for the so-called "double filter" collaboration, that can be used either in conjunction with a "pipeline with message", or a "pipeline with buffer". This collaboration can be generalized for any type of active objects that are sharing the same execution thread. It describes the case of two filters that are running in the same process. The LQN model captures the contention of the two objects for the same execution thread. Since the LQN version used for this thesis did not accept cyclic graphs for reasons related to deadlock prevention, we represented each passive object filter as a LQN "dummy" task, therefore treating it as an active object. In order to prevent the dummy tasks from executing simultaneously, a third "executive" task serializes the first two. All filter's processing is charged to the "executive" task entries. The dummy tasks don't do any real work, and are waiting instead for the executive task to do the work on their behalf. They are allocated on a dummy processor (not to interfere with the scheduling of the "real" processor node).

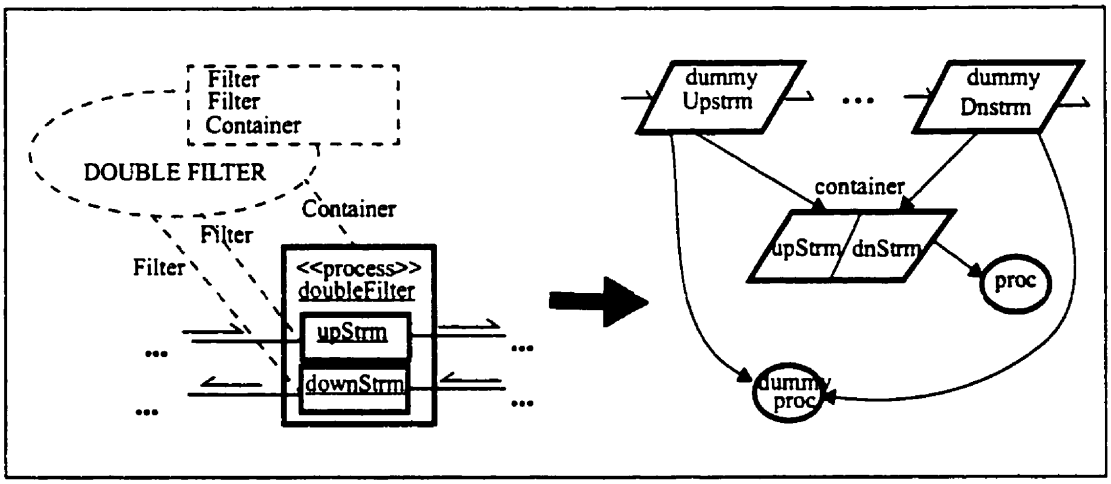


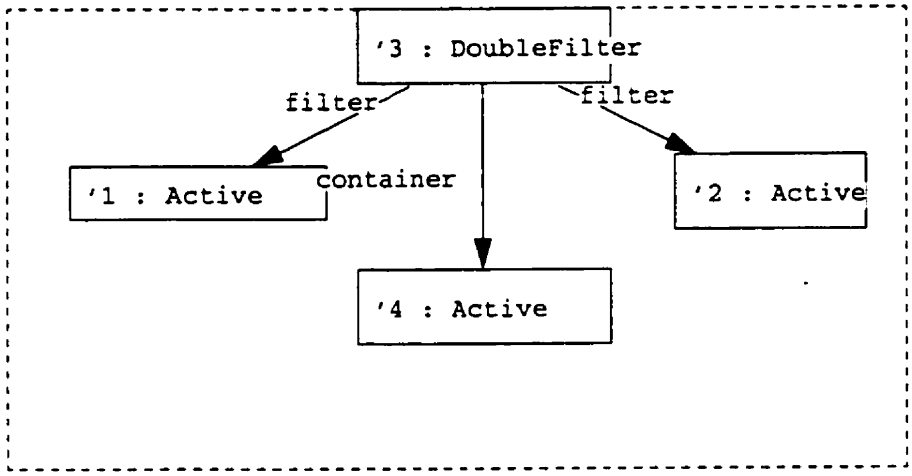
Figure 4.10: Transformation of double filter collaboration

The following fragment of PROGRES code (Figure 4.11) illustrates the graph production rules for this collaboration. The left-hand side of the graph consists of a *DoubleFilter* node (node 3), two *Active* Object nodes (node 1 and 2) that represents the two filters, another *Active* Object node (node3) that represents the container three edges indicate the roles of the objects.

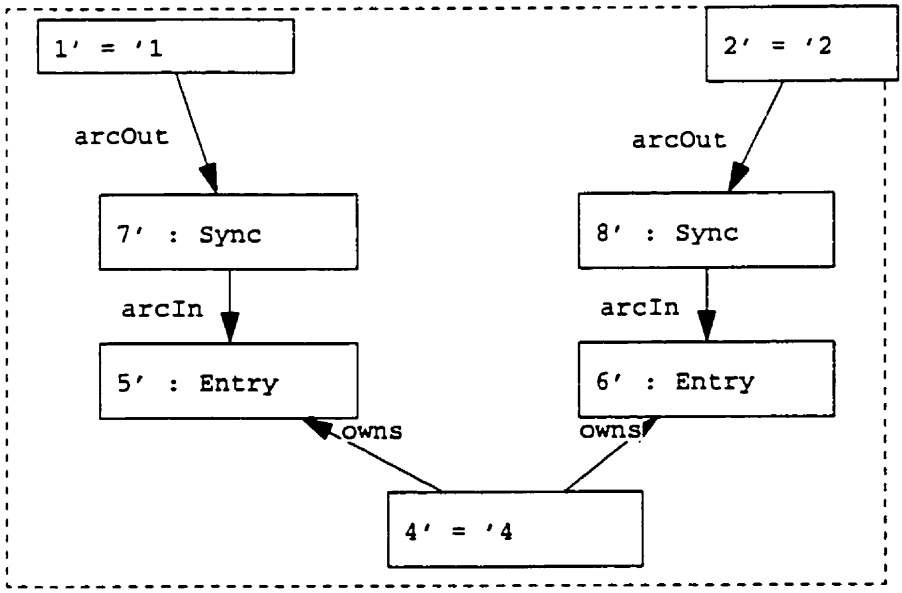
The transformation is pretty straightforward. In the right-hand side, the *DoubleFilter* node does not exist any more because a *collaboration* does not have an LQN counterpart. The two *Active* object (node 1 and 2) become *dummy* tasks as explained before. In order to prevent the dummy tasks from executing simultaneously, a third *executive* task (node 4) serializes the first two. Two new *executive* task entries (node 5 and 6) represent the processing of the two filters. Two new *Sync* node (node 7 and 8) are also added representing the attributes of a synchronous request in the LQN model. The *transfer* part below the

graph indicates the modifications of the attributes of the right-hand side, like update the name of node 1 and 2 with the "dummy" prefix.

production TransformDoubleFilter =



::=




```
transfer 1'.name := "Dummy" & '1.name;
         2'.name := "Dummy" & '2.name;
         5'.name := '1.name;
         6'.name := '2.name;
         7'.fromName := "Dummy" & '1.name;
         7'.toName := '1.name;
```

```
            8'.fromName := "Dummy" & '2.name;
            8'.toName := '2.name;
            7'.Type := "y";
            8'.Type := "y";
            5'.TaskName := '4.name;
            6'.TaskName := '4.name;
            4'.name := '4.name & "exec";

end;
```

Figure 4.11: Production rule for double filter collaboration

4.4.4 Client-Server Pattern

4.4.4.1 Client-Server Pattern with Direct Connection

Figure 4.12 illustrates the transformation to LQN of a direct client-server connection through a synchronous communication (*rendezvous*), where the client sends a request to the server and blocks until the reply from the server comes back. A server may offer a wide range of services (represented here as the server's object methods) each one with its own performance attributes (execution time and number of visits to other servers). A client may invoke more than one of these services at different times.

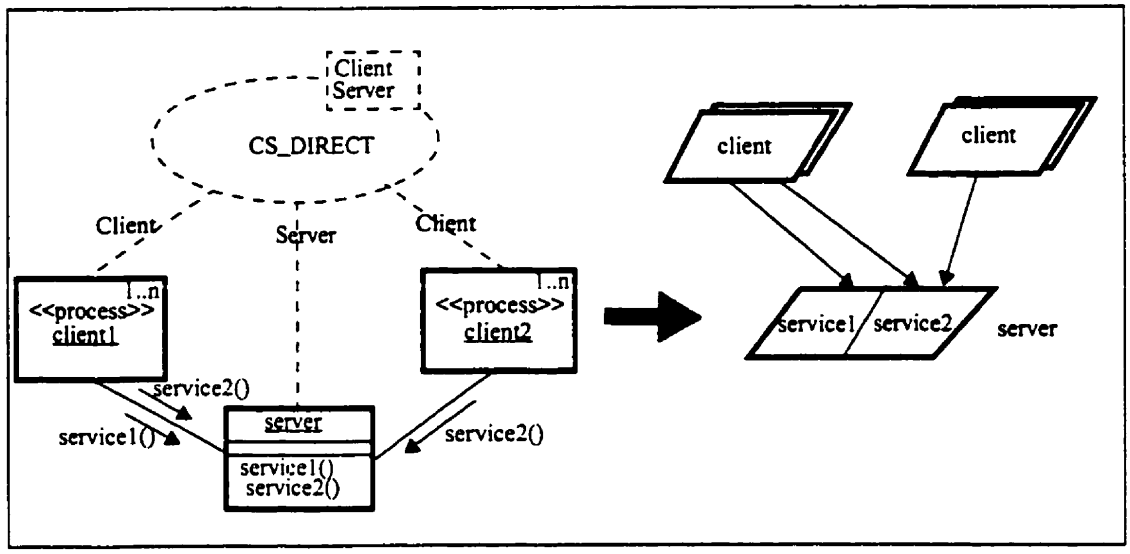
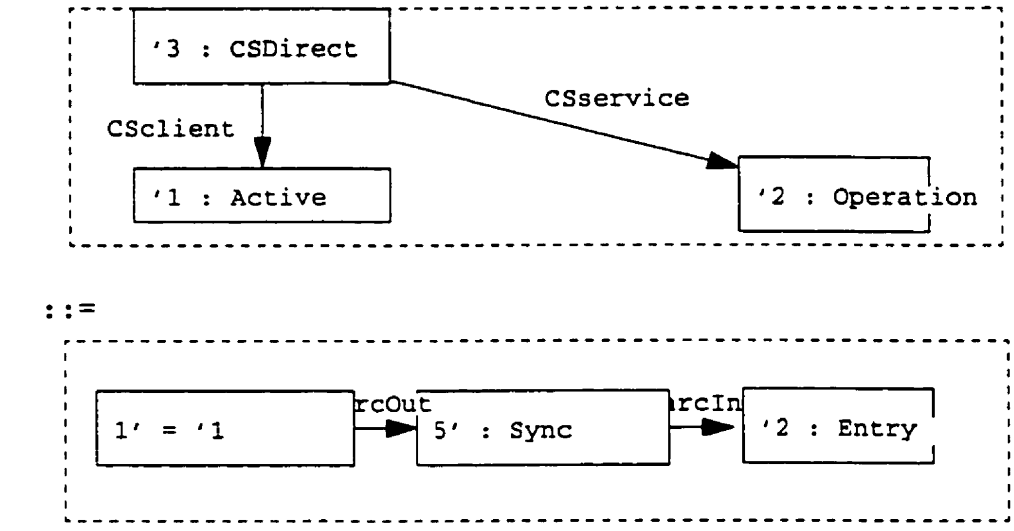


Figure 4.12: Transformation of the client-server pattern with a direct connection

The following fragment of PROGRES code (Figure 4.13) illustrates the graph production rule for this case. The left-hand side of the graph consists of a *CSDirect* node (node 3), an *Active Object* node (node 2) that represents the client, an *Operation* node that represents the service provided by the server and two edges (*client and service)* that indicate the roles of the objects.

In the right-hand side, the *CSDirect* node remains in the graph because it may be connected to other operations of the same server as well. The *Operation* node (node 4) becomes an *Entry.* A new *Sync* node (node 5) is also added representing the attributes of a synchronous request in the LQN model. Since the server may be offering other services (i.e. Operations) the production will be execute in a loop to transform all the operations to entries. The loop will end when no match of the LHS will be found in the input graph. The

transfer keyword below the graph indicates the modifications of the attributes of the right-hand side nodes. For example, the serviceTimeO of *Operation* node 2 in the LHS become the serviceTimeE of *Entry* node 2.




```
transfer 1'.isEntry := 1;
         5'.fromName := '1.name;
         5'.toName := '2.name;
         5'.ServiceTimeE := '2.ServiceTimeO;
         5'.Type := "y";
end;

transaction TransformAllCSDirect=
   loop
         TransformCSDirect
   end
end;
```

Figure 4.13: Production rule for client-Server pattern with a direct connection

4.4.4.2 Client-Server Connection by Forwarding and Half-Forwarding Broker

Software developers of client-server systems are mostly interested in the components that are part of their application, and less in the details of the underlying midware, operating system or networking software. The use of UML collaborations comes in handy, because it allows us to hide unnecessary details. For example, client-server applications using a CORBA interface do not have to show explicitly the "broker" component in their architecture (as it is not part of the software application). Instead, a collaboration can be used to indicate the type of desired client-server connection. However, the performance model will represent explicitly the broker and it's interaction with the client and server counterparts.

Figures 4.14 and 4.15 illustrate the transformation of the client-server connections which use the forwarding and half-forwarding broker. Since the architecture does not show the broker explicitly, the input graphs have similar architectural descriptions, differentiated only by the kind of UML collaboration used. However, their LQN models are quite different, as the connections have very different operating modes and performance characteristics.

In the client-server connection with a forwarding broker case, 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. The forwarding broker (Figure 4.14) is modelled as an LQN multi-server with as many entries as server entries. Each forwarding broker replication models a "virtual" thread that is dedicated to a given client request until the completion of its service. (A virtual thread may be implemented either as a thread or as a process). While some of the broker thread will be blocked, waiting for the server's reply, other such threads will accept new client requests. However, all virtual threads compete for the same processor and execute one at a time.

In the case of half-forwarding broker from Figure 4.15, 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). The half-forwarding broker model (Figure 4.15) uses LQN forwarding arcs (drawn with dotted lines) which have a special semantic. After accepting a request from a client, the acceptor task will do some processing, then will forward the request to another task. The forwarder is free to continue its activity, while the client remains blocked, waiting for the reply. The second task that continues to serve the request may eventually complete it and send the reply directly back to the client, or may decide to forward the request to another task. The LQN semantic implies that a reply will be sent to the client by the last task in the forwarding chain (but the reply is not represented as an arc in the model). In Figure 4.15, the forwarding 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 allocation of tasks to processors is not shown in figures, because the transformation does not depend on it (each LQN task is allocated exactly as its architectural component counterpart).

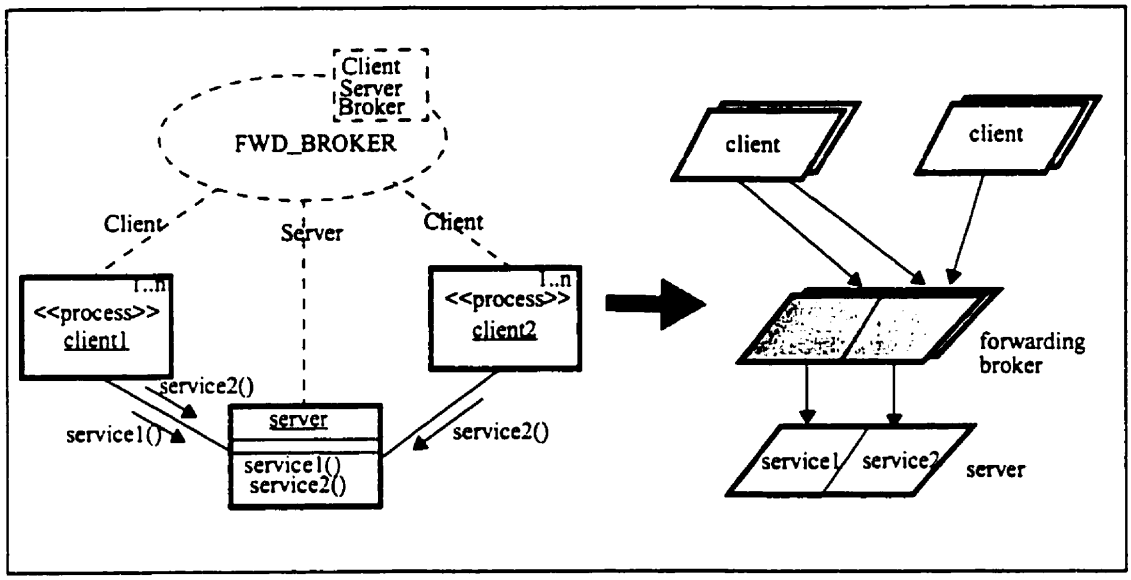


Figure 4.14: Transformation of the client-server pattern with a forwarding broker

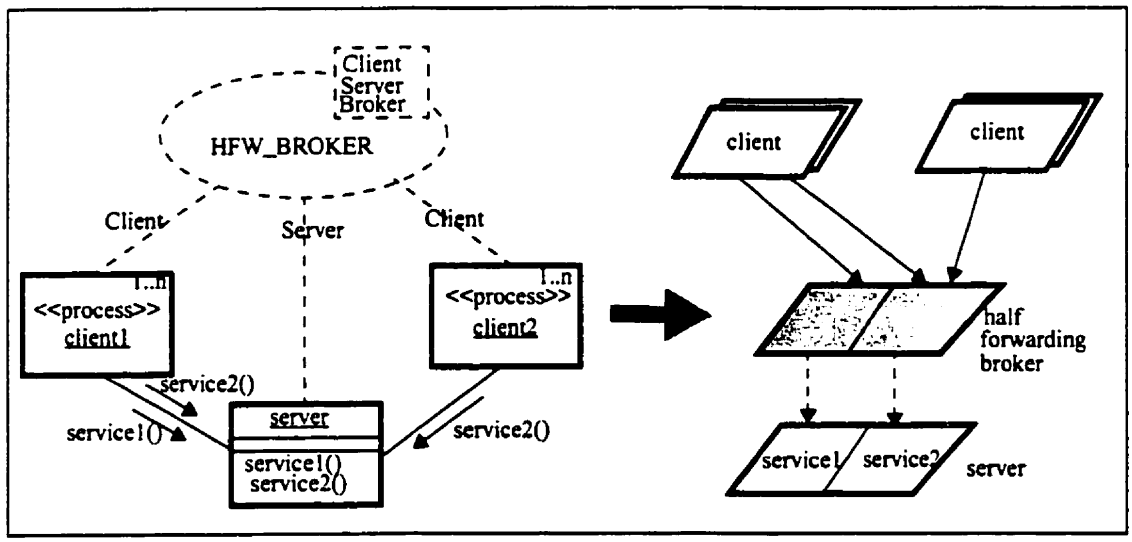


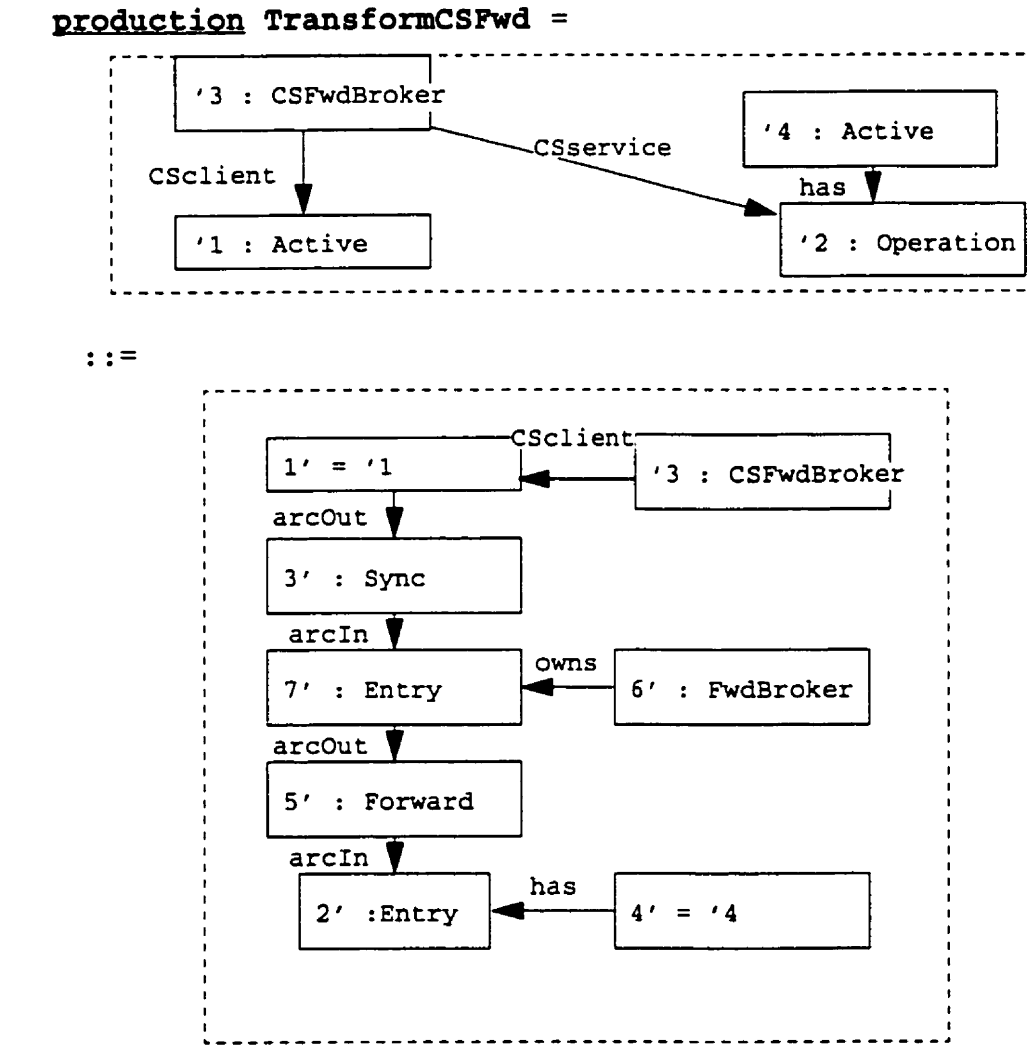
Figure 4.15: Transformation of the client-server pattern with a half-forwarding broker

The following fragment of PROGRES code (Figure 4.16) illustrates the graph production rule for client-server connection with half-forwarding broker. The left-hand side of the graph has a*CSFwdBroker* node (node 3). An *Active* Object nodes (node 2) that represents the client, an *Operation* node that represents the service provided by the server and two edges (*client* and *service) that* indicate the roles of the objects. (The broker does not appear in the left-hand graph, being hidden in the *collaboration*)

In the right-hand side, the *CSFwdBroker* node remains in the graph because it may be connected to other *Operation* nodes of the same server. A new *fwdBroker* node that represents the broker is added (node 6). The *Operation* node (node 4) becomes an *Entry* of the new *fwdBroker* node.A new *Sync* node (node 3) and a new *Forward* node are also added representing the attributes of a synchronous request and a forwarding request, respectively. Since the *CSFwdBroker* node still exists, if there are other *Operation* node of the same server, the production will be execute in a loop to transform all the operations to entries. These entries are owned by the same task to prevent them to execute simultaneously.The *transfer* section below the graph indicates the modifications of the attributes of the right-hand side, like the *ServiceTime* or *name* of the entries.

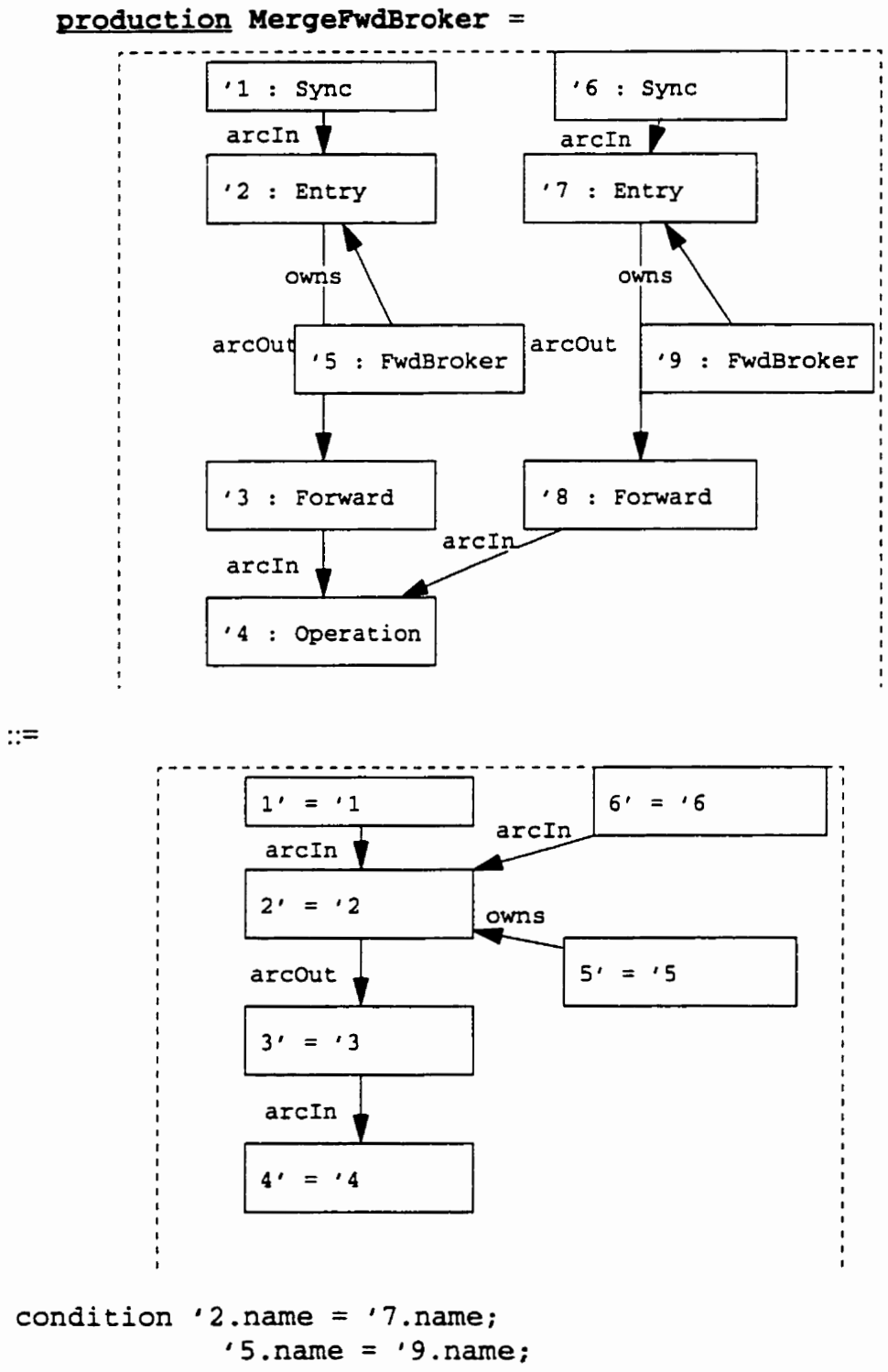
In order to simplify the rule for the transformation of this pattern, a new *fwdBroker* node represents the broker and a entry of the broker task are added each time when a match of the left side is found in the graph. This approach was chosen for simplicity to reduce the number of production rules and the conditions for their application. However this will lead to a duplication of the *fwdBroker* node and its entries. So, when no more matches

from the LHS are found a clean-up will be done by the transaction *MergeAllFwdBroker,* which calls the production *MergeFwdBroker* (see Figure 4.17) in a loop. The basic idea of this production is to merge two *fwdBokers* into one if there is a duplicated fwdbroker is found.




```
    transfer 6'.name := '2.name & "Broker";
             7'.name := '4.name & "BrokerEntry";
             2'.name := '2.name ;
             2'.ServiceTimeE := '2.ServiceTime0;
end;
```

Figure 4.16: Production rule for client-server pattern with a half forwarding broker



```
end;
```

Figure 4.17: Production rule for merging two duplicated forwarding brokers

4.4.4.3 Client-Server Connection by Handle-Driven Broker

In the case of a Client-Server connection with a handle-driven broker, a handle-driven broker 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. The LQN model for the handle-driven broker sends two separate messages for each client request: one to the broker for getting the handle, the other directly to the desired server entry. (as in Figure 4.18)

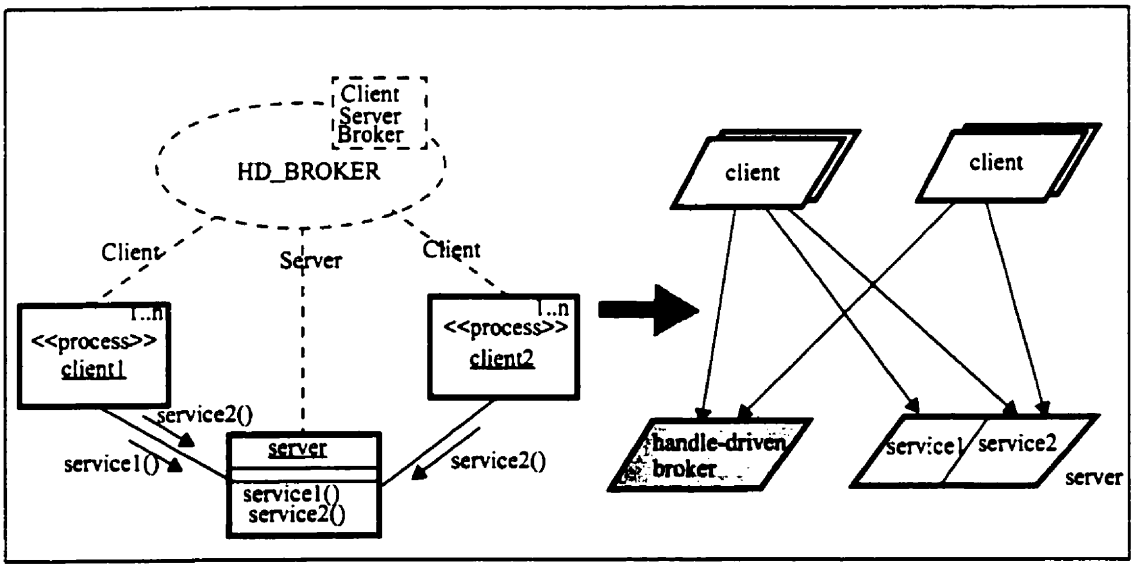


Figure 4.18: Transformation of the client-server pattern with a handle-driven broker

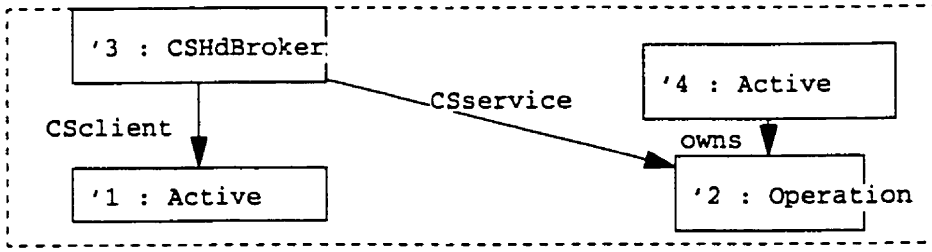
The following fragment of PROGRES code (Figure 4.19) illustrates the graph produc-

tion rule for client-server connection with handle-driven broker case. The left-hand side of the graph has a *CSHdBroker* node (node 3), an *Active* Object nodes (node 2) that represents the client, an *Operation* node represents the service provided by the server and two edges (*client* and *service)* that indicate the roles of the objects. The broker does not appears in the LHS of the graph, as mentioned before.

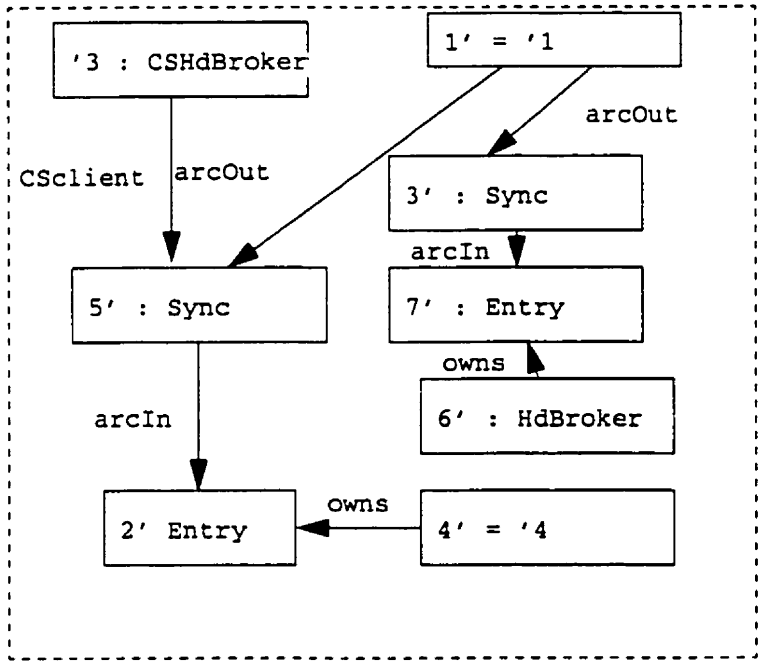
In the right-hand side, the *CSHdBroker* node remains in the graph because it may be connected to other *Operation* nodes of the same server as well. A new *fwdBroker* node is added (node 6). The *Operation* node (node 4) becomes an Entry of the new *fwdBroker* node. Two new *Sync* node (node 3 and 5) representing the attributes of a synchronous requests are also added. Since the *CSHdBroker* node still exists, the production will be executed in a loop to transform all the operations of the same server to entries. The *transfer* keyword below the graph indicates the modifications of attributes of the right-hand side.

In order to simplify the rule for the transformation of this pattern, a new *HdBroker* node that represents the broker and an entry of the broker task are added each time when a match of the left side is found in the graph. This approach will lead to a duplication of *HdBroker* node and its entries. Therefore, when no more matches from the left-side are found, meaning that all operations are transformed into entries, a clean-up will be done by the transaction *MergeAllHdBroker*, which calls the production *MergeHdBroker* (see Figure 4.20) in a loop. The basic idea of this production is to merge two fwdBokers into one if there is a duplicated fwdbroker is found. This work is similarly to the clean-up procedure for the half-forwarding broker explained before.

production TransformCSHd =



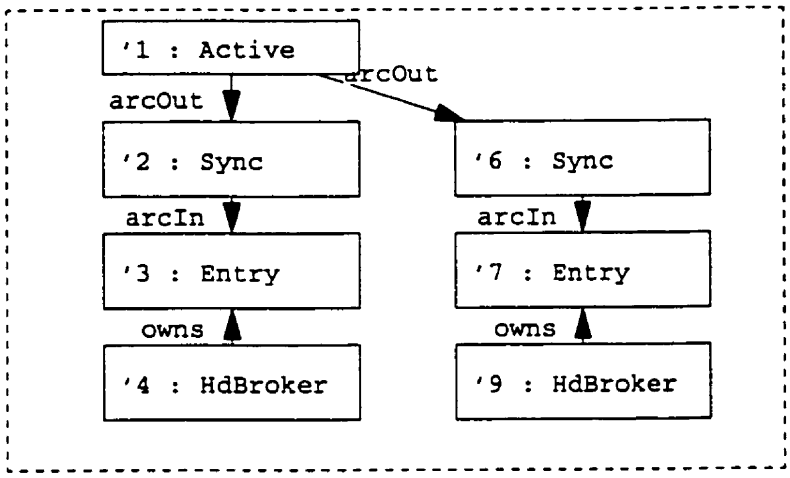
::=



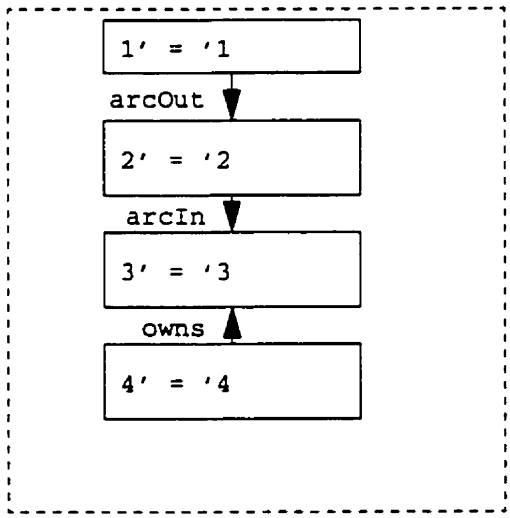

```
transfer 6'.name := '4.name & "Broker";
         7'.name := '2.name & "BrokerEntry";
         3'.fromName := '1.name;
         3'.toName := '2.name & "BrokerEntry";
         5'.fromName := '1.name;
         5'.toName := '2.name;
         3'.Type := "y";
         5'.Type := "y";
        2'.name := '2.name;
        2'.ServiceTimeE := '2.SerciveTimeO;
         2'.TaskName := '4.name;
         7'.TaskName := '4.name & "Broker";
end;
```

Figure 4.19: Production rule for client-server pattern with a handle-driven broker

```
production MergeHdBroker =
```




```
::=
```




```
  condition '4.name = '9.name;
end;
```

Figure 4.20: Production rule for merging two duplicated handle-driven brokers

4.4.5 Critical Section Pattern

The critical section pattern applies to cases where two or more active objects share the same passive object. The constraint {sequential} attached to the methods of the shared object indicates that the callers must coordinate outside the shared object (for example, by the means of a semaphore) to insure correct behaviour. The transformation of the critical section collaboration produces either the model given in Figure 4.21a or 4.21b, depending on the allocation of user processes to processors (similar to the pipeline case). The premise is that an LQN task cannot change its processor allocation. Since the operations on the shared object (i.e., critical sections) may be executed by different threads of controls of different users running on different processors, each operation is modelled as an entry that belongs to a different task f_1 to f_N running on its user's processor. However, these tasks must be prevented from running simultaneously, reason for which the semaphore task was introduced. The performance attributes to be provided for each user must specify critical and non-critical execution times separately.

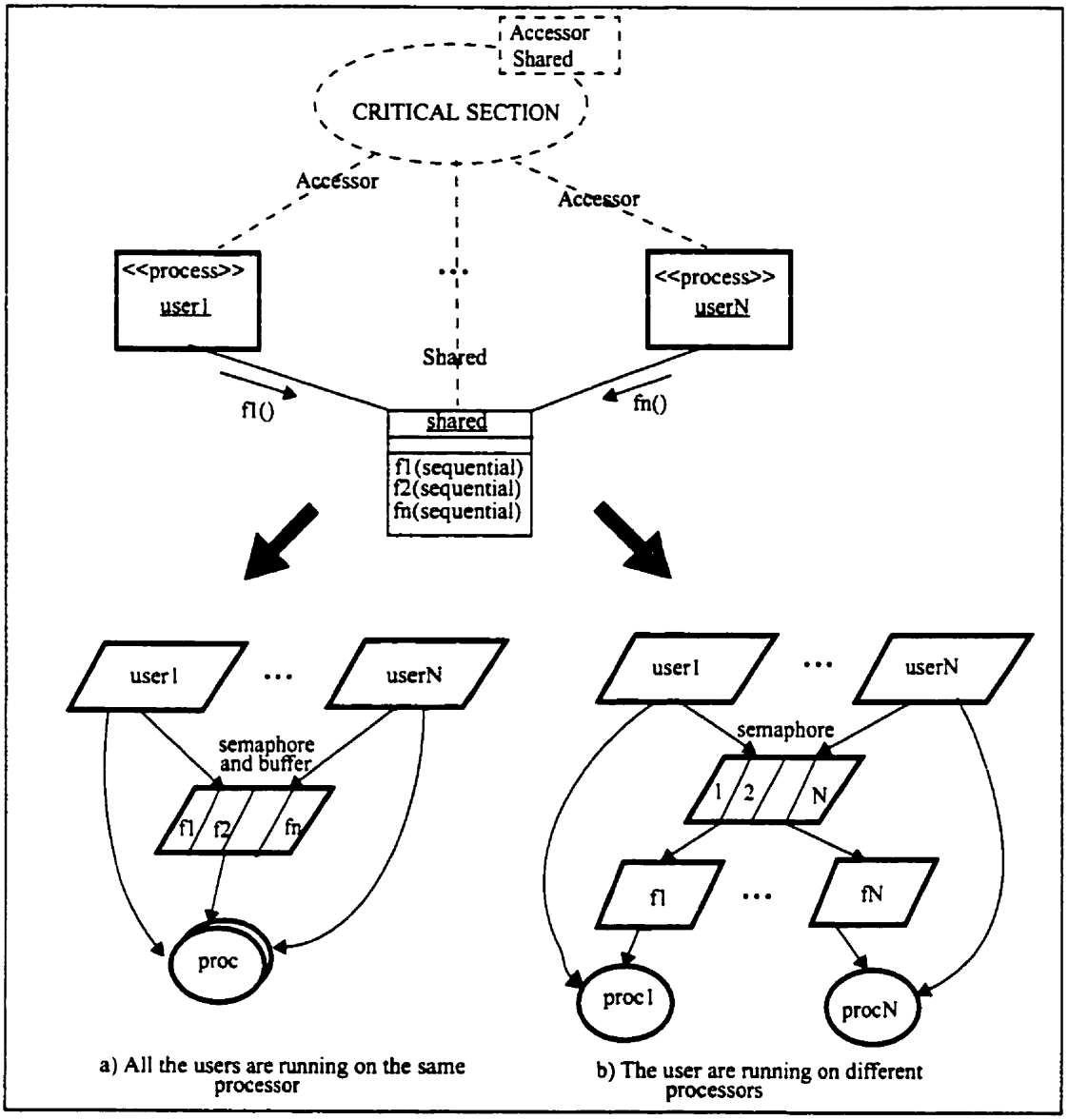


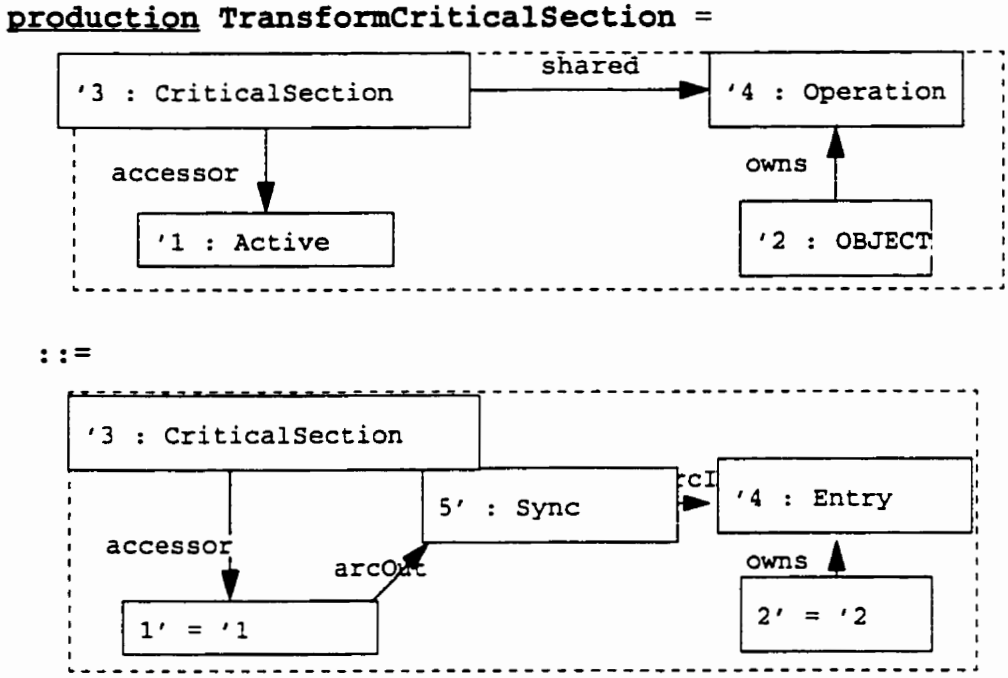
Figure 4.21: Transformation of critical section pattern

The following fragment of PROGRES code (Figure 4.22) illustrates the graph production rule for the case presented in this pattern (as in Figure 4.21a, where all users are located on the same processor). The left-hand side of the graph consists of a *CriticalSection* node (node 3), an *Active* Object nodes (node 1) represents the acessor, an *OBJECT* node

(node2) that owns an *Operation* node and three edges (*accessor* and *shared)* that indicate the roles of the objects.

In the right-hand side, the *CriticalSection* node remains in the graph because it may be connected to other *Operation* nodes as well. The *Operation* node (node 4) becomes an *Entry* owned by node 2. A new *Sync* node (node 5) is also added representing the attributes of a synchronous request in LQN model. If there are other *Operation* nodes for the same OBJECT exist, the production will be execute in a loop to transform all operations to entries. The *transfer* keyword below the graph indicates the modifications of the attributes on the right-hand side.

The *condition* keyword in this production indicates an additional condition on any subgraph that matches the LHS in order for the production to be executed. In this case the condition is both Active object must locate on the same processor. For the other case (as in Figure 4.21b, both objects are located on different processors), the LHS of the production rule is similar except it has a different condition which means that the two Active object must locate on different processors. The RHS of the rule is bit different too. The tasks will be located on different processors. Since we do not have a Processor node in the graph schema, we use the Processor attribute instead. So, the Processor attributes of the tasks will be different.




```
condition 1'.processor := '2.processor;
transfer 5'.fromName := '1.name;
          5'.toName := '4.name;
          5'.Type := "y";
          4'.TaskName := '2.name;
end;
```

Figure 4.22: Production rule for critical section pattern

4.5 Other Production Rules

So far we have shown the production rules for the main patterns. There are still other production rules in the program dealing with the creation of the input graph, setting attributes, clean-up procedure, retrieving attributes from all tasks and entries, writing to file etc. They will be explained in this section.

4.5.1 Create Input Graph and Set Attributes

As mentioned before, in this thesis the input graph is created by using PROGRES productions. An input graph is created component by component and pattern by pattern. When creating an input graph, the performance specific attributes like *ServiceTime*, *VisitRatio* and *Processor* can also be set.

The following example show how to create two *Operations* with 3-phase *ServiceTime* attributes:

```
   CreateOperation ( "service1", "0", "0.2", "0" )
& CreateOperation ( "service2", "0", "0.3", "0" )
```

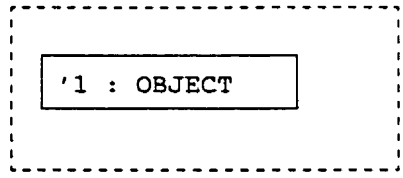
The following codes show how to create two *Active* Objects with *Processorname*, *Processortype* and *multiplicity* attributes:

```
   CreateActive ( "client1", "proc1", "f", "i", 10 )
& CreateActive ( "client2", "proc2", "f","i", 20 )
```

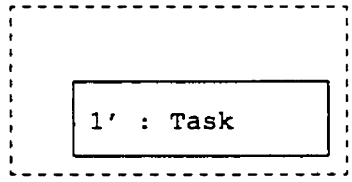
4.5.2 Transform All Objects to Tasks

This production (Figure 4.23) will be executed at the last stage of the transformation, when all the patterns are transformed and there is no need for intermediate nodes any more. This production will be called in a loop to transform all the OBJECT node to Task nodes. The edges associate with OBJECT node still remain associated with the corresponding Task node, due to the *redirect* keyword from the *embedding* section (If the *redirect* part would be missing, all the edges associated with the node in LHS which has no counterpart in RHS would be deleted).

```
production TransformObjectToTask =
```




```
::=
```



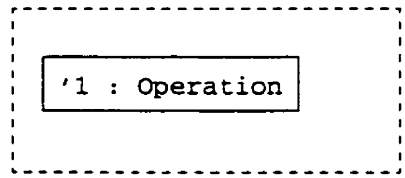

```
  embedding
    redirect -owns->, <-arcIn-, -arcOut-> from '1 to 1';
  transfer 1'.name := '1.name;
           1'.isEntry := '1.isEntry;
           1'.MultiplicityT := '1.MultiplicityO;
           1'.Processor := '1.Processor;
           1'.ProcType := '1.ProcType;
end;
```

Figure 4.23: Production rule for transforming an *OBJECT* to a *Task*

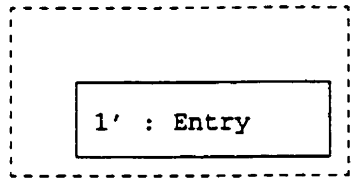
4.5.3 Transform All Operations to Entries

This production (Figure 4.24) will be executed at the last stage of the transformation, after all the patterns were transformed. This production will be called in a loop to transform all the Operation node to Entry node. The edges associate with Operation node still remain in the same association with the corresponding Entry node.

```
production TransformOperationToEntry =
```



```
    ::=
```



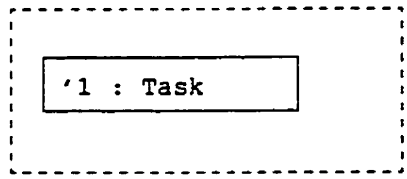

```
    embedding
      redirect <-owns-, <-arcIn-, -arcOut-> from '1 to 1';
    transfer 1'.name := '1.name;
             1'.TaskName := '1.TaskName;
             1'.ServiceTimeE1 := '1.ServiceTime01;
             1'.ServiceTimeE2 := '1.ServiceTime02;
             1'.ServiceTimeE3 := '1.ServiceTime03;
  end;
```

Figure 4.24: Production rule for transforming an *Operation* to an *Entry*

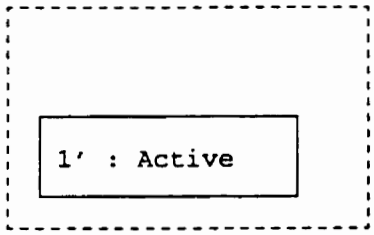
4.5.4 Get Attributes of All Tasks and Entries

These productions (Figure 4.25-4.27) basically retrieve all the attributes of all Task nodes, Entry nodes and ARC_PARAMETER nodes and return them (like the *return* keyword indicates). Later these attributes will be saved in an data structure by calling external C functions. The attributes like *taskName, entryName, ServiceTime, visitRation, multiplicity* and types of *arcs* are used to generate the LQN file.

```
   production GetTaskAtt( out tname, proc, proctype : string
; out m : integer)
  =
```



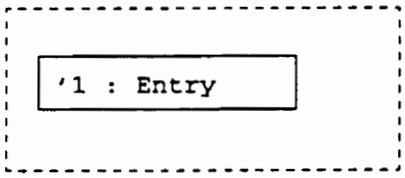
```
  ::=
```



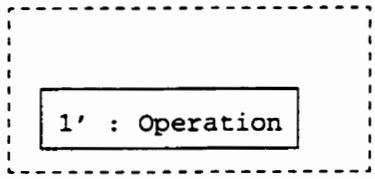

```
  embedding
    redirect -owns->, <-arcIn-, -arcOut-> from '1 to 1';
  return tname := '1.name;
         proc := '1.Processor;
         proctype := '1.ProcType;
         m := '1.MultiplicityT;
end;
```

Figure 4.25: Production rule for retrieving *attributes* of a *Task*

```
production GetEntryAtt( out ename, taskname : string ;
                        out serT1, serT2, serT3 : Real) =
```



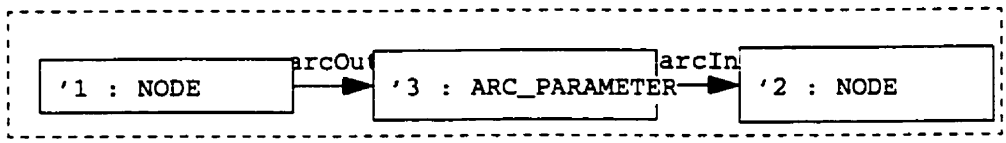

```
  ::=
```




```
  embedding
    redirect <-owns-, <-arcIn-, -arcOut-> from '1 to 1';
  return ename := '1.name;
         taskname := '1.TaskName;
         serT1 := '1.ServiceTimeE1;
         serT2 := '1.ServiceTimeE2;
         serT3 := '1.ServiceTimeE3;
end;
```

Figure 4.26: Production rule for retrieving *attributes* of an *Entry*

```
production GetArcAtt( out arctype, fromname, toname :
string ;
                      out visit1, visit2, visit3 : Real)
  =
```




```
  ::=
```


```
  return arctype := '3.Type;
         fromname := '3.fromName;
         toname := '3.toName;
         visit1 := '3.NbVisit1;
         visit2 := '3.NbVisit2;
         visit3 := '3.NbVisit3;
  end;
```

Figure 4.27: Production rule for retrieving *attributes* of an *ARC_PARAM*

4.6 Generating An LQN Model File

PROGRES Program itself provide limited features to deal with output to files. The solution is to write external functions using C programming language and import them into a PROGRES program. In this case, we want to generate a text file according to a predefined

LQN model format [Franks95], which can be used as an input file for existing LQN solvers which can solve the model and provide performance results. 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.

The following fragment of PROGRES code illustrates how a PROGRES program imports external C functions:

```
   from writeLQN import

   functions
writeLQNFile    : ( string) -> integer,
saveEntryAtt    :
        ( integer, string, string, string, string, string) ->
          integer                                              ,
saveArcAtt      :
         ( integer, string, string, string, string, string,
string)-> integer                                              ,
saveTaskAtt     :
        ( integer, string, string, string, integer) -> inte-
ger;

    end;
```

These C function basically save the attributes of all tasks and entries and write them to a text file according to the predefined LQN input file format. The main transaction in PROGRES will call these function after it does all the graph transformations.

4.7 Control Structure for Graph Transformation

A software system contains many components involved in various architectural connection instances (each described by a pattern/collaboration), and a component may play different roles in different patterns. The transformation of the architecture into a performance model is done in a systematic way, pattern by pattern. PROGRES searches for subgraphs in the underlying input graph that match the left-hand side of the rules for different given pattern. The transformation process ends when all UML collaborations have been processed.The final result is an LQN model that can be written to a file.

The following shows the sequential steps of the whole transformation in pseudo-code:

```
Transaction UMLtoLQN
Create input graph;
begin
While (there is pattern 1 in input graph)
       Transform it;
   While (there is pattern 2 in input graph)
       Transform it;
      ......
While (there is pattern n in input graph)
       Transform it;
    Clean collaborations();
    Merge Tasks;
    Merge Entries;
    Get Attributes;
    Write to LQN file;
end;
```

The graph will be in an intermediate stages if not all interactions are covered by known patterns. In this case the user gets an warning message from the tool. This problem can be solved by defining more patterns.

The control structure for this PROGRES program is similar to the control structure we have discussed in the previous chapter. The whole approach is illustrated with a case-study in section 4.8.

4.8 Case-Study: A Telecommunication System

This section presents the architecture of an existing telecommunication system which is responsible for developing, provisioning and maintaining various intelligent network services, as well as for accepting and processing real-time requests for these services (see Figure 4.28). The system was modelled in LQN, and its performance analysed in [Shousha98a]. Here we consider only the transformation from the system's UML architecture to its LQN model. The real time scenario modelled in [Shousha98a] starts from the moment a request arrives to the system and ends after the service was completely processed and a reply was sent back. As illustrated in Figure 4.27, a request is passed through several filters of a pipeline: from *Stack* process to *IO* process to *RequestHandler* and all the way back. The main processing is done by the *RequestHandler*, which accesses a real-time database to fetch an execution "script" for the desired service, then executes the steps of the script accordingly. The script may vary in size and types of operations involved, and hence the workload varies largely from one type of service to another (by one or two orders of magnitude). Due to experience and intuition, the designers decided from the beginning to allow for multiple replications of the RequestHandler process in order to speed up the system. Two shared objects, *ShMem1* and *ShMem2*, are used by the multiple *RequestHandler* replications.

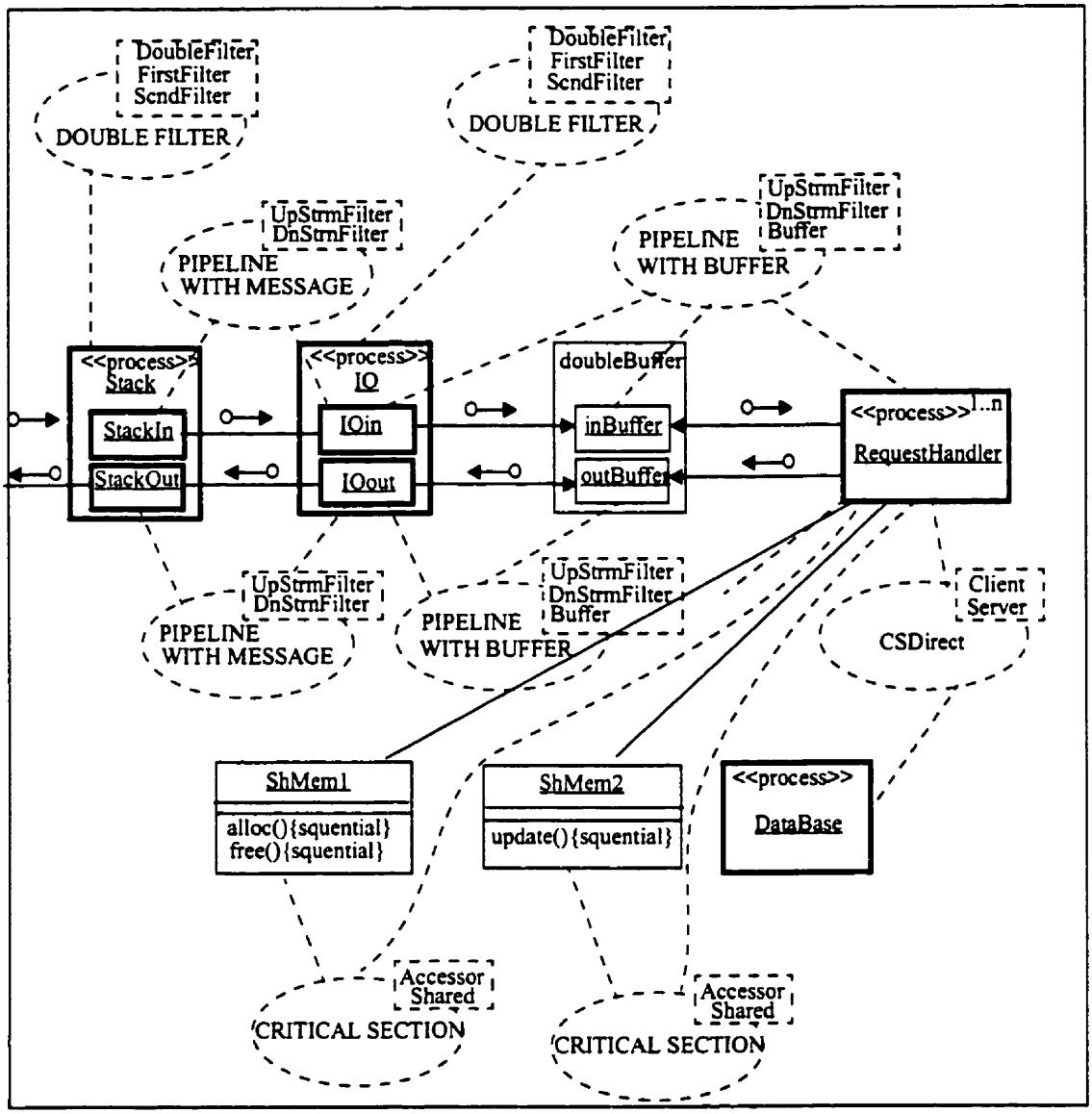


Figure 4.28: UML descriptions of high-level architecture of a telecommunication system

The following is a part of the PROGRES program illustrates the main control structure of this case.

```
   transaction MAIN =
CreateInputGraph
& TransformPatterns
```

```
& writeToFile
    end;

  transaction CreateInputGraph =
CreateDoubleFilter
( "Stack", "StackIn", "StackOut", "proc1", "f" )
& CreateDoubleFilter ( "IO", "IOin", "IOout", "proc1", "f" )
& CreateDoubleBuffer ( "Buffer", "proc1", "f" )
& CreateActive ( "RequestHandler", "proc1", "f", 10 )
& CreateOperation ( "service1", "0", "0.8", "0" )
& CreateServer1 ( "DataBase", "service1", "proc1", "f" )
& AddPipeline ( "StackIn", "IOin" )
& AddPipeline ( "IOout", "StackOut" )
& AddPipelineWithDoubleBuf
        ( "IOin", "Buffer", "RequestHandler" )
& AddPipelineWithDoubleBuf
        ( "RequestHandler", "Buffer", "IOout" )
& AddCSDirect ( "RequestHandler", "service1" )
& CreateSharedMem2 ( "ShMem1", "alloc", "free", "proc1", "f"
)
& CreateSharedMem1 ( "ShMem2", "update", "proc1", "f" )
& AddCriticalSection ( "RequestHandler", "ShMem1", "alloc" )
& AddCriticalSection ( "RequestHandler", "ShMem1", "free" )
& AddCriticalSection ( "RequestHandler", "ShMem2", "update"
)

  transaction TransformPatterns =
TransformAllDoubleFilter
& TransformAllPipeline
& TransformAllPipelineWithBuf
& MergeAllDoubleBuffer
& TransformAllCSDirect
& TransformAllCriticalSection
    end;

  transaction TransformAllPipeline =
      loop
        TransformPipeline
      end
    end;

  ......
```

```
transaction writeToFile =
    use int : integer
    do
        DeleteCollaborationNodes
      & TransformAllObjectToTask
      & TransformAllOperationToEntry
      & GetAllTaskAtt
      & GetAllEntryAtt
      & GetAllArcAtt
      & int := writeLQNFile ( "testcase.lqn" )
    end
  end;
```

We create an input graph first, and transform it pattern by pattern until no untransformed node is found in the whole graph. Finally the program saves all the relevant attributes and writes to a file according to the LQN input file format.

The case-study system was built to run either on a single-processor or on a multi-processor with shared memory. Processor scheduling is such that any process can run on any free processor (i.e., the processors were not dedicated to specific tasks). Figure 4.29 illustrates the LQN model of the system obtained by applying the graph transformations described in this chapter.

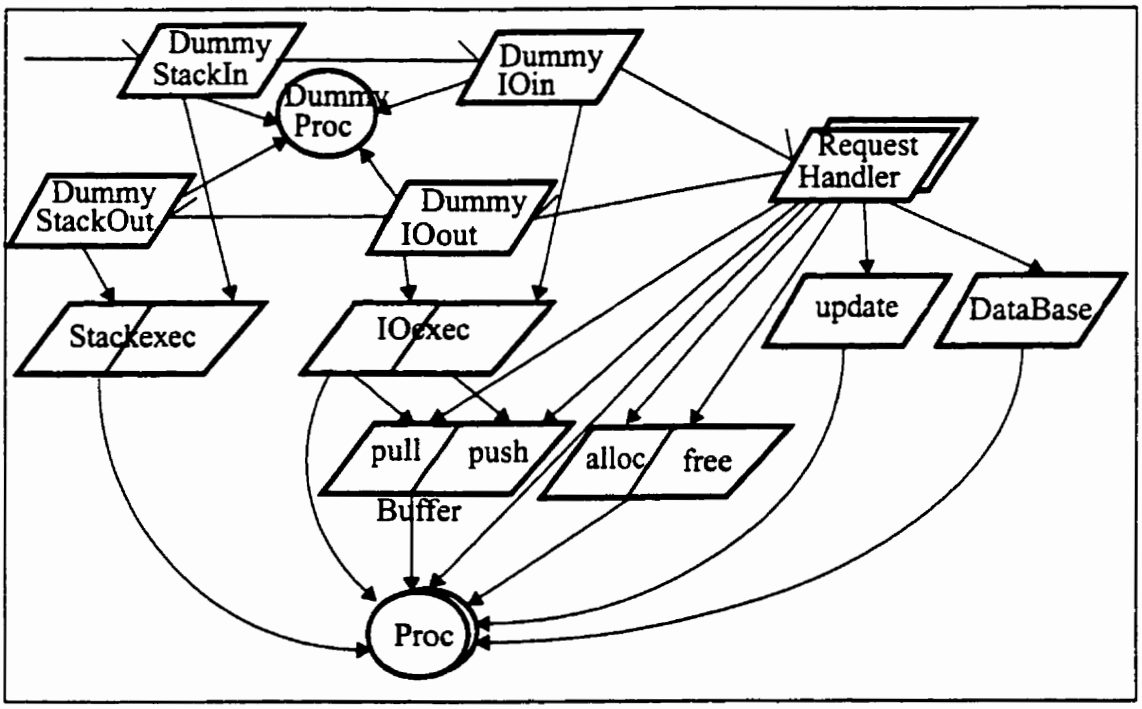


Figure 4.29: LQN model of the telecommunication system

The LQN model file for this case study can be generated by the PROGRES program using external functions written in C programming language. This file can be directly used as an input file of an LQN solver.The LQN model file for this case study will be shown in Appendix 3.

The LQN model can be solved with existing LQN solvers [Franks95] either analytically or with simulation. The performance analysis of the model is outside the scope of the thesis.

CHAPTER 5

Conclusions

5.1 Conclusions

The purpose of this thesis is to contribute to bridging the gap between software architecture and performance analysis. By automating the construction of the performance models from software architectures, the time and effort required for SPE will be considerably reduced, and the consistency between the model and the system under development more easily maintained.

The thesis proposes an approach to automate the derivation of the structure of an LQN performance model from the patterns used in the high-level architecture of the system. The proposed approach is based on the graph grammar formalism, and it was implemented by using PROGRES, a known visual language and environment for programming with graph rewriting systems[Schuerr90]. Two versions of the implementation are presented. In

the first one we used an ad-hoc language to describe the high-level architectural patterns of the system, whereas in the second we used the new UML standard. The first approach has the advantage that the notation is "*tailor-made*" for our needs, and thus contains exactly the kind of information requests for building a performance model. The disadvantage is obvious due to the fact that notation is ad-hoc, not standard. We tried to overcome this by using UML in our second version. However, this choice being a new challenge. UML is a very rich notation that contains many kinds of diagrams and much more information than we need to express high-level architectures.The approach taken in the thesis is to use only a subset of UML features for the input graph to the PROGRES transformation program. This does not solve the problem of accepting UML models as input to our transformation program, which opens directions for future work.

5.2 Future Work

The matter of this thesis can be further researched in the following direction:

- Extend the research with more software architectural patterns of systems with both descriptions we have used in this thesis.
- Extend the tool with a interface with UML graphical tool to generate the PROGRES input graph by graphical or textual definition.
- Extend the process of building LQN model by taking into account more detailed UML design documents. Whereas the structure of the LQN model is desired from the high-level architecture of the system, as shown in the thesis, the model parameter could be

obtained from lower-level behavioural views that show what happens inside the higher level architectural components. By aggregating the resource demands at lower levels of abstraction, one should be able to obtain the average resource demands of the high level component (which are the parameters of the performance model, such as execution times and visit ratios).

References

[Adebayo97] O. Adebayo, J. Neilson, D. Petriu, *"A Performance Study of Client-Broker-Server Systems"*, in Proceedings of CASCON'97, pp 116-130,

[Adler95] R. Adler, *"Distributed Coordination Models for Client/Server Computing"*, IEEE Computer, pp 14-22, April 1995.Toronto, Canada, November 1997.

[Allen97] 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.

[Booch99] G.Booch, J.Rumbaugh, I.Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[Buschmann96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley Computer Publishing, 1996.

[Franks95] 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.

[Franks96] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, C.M. Woodside. *"Performance Analysis of Distributed Server Systems"*. In Proceedings of the 6th International Conference on Software Quality, pp. 15-26, Ottawa, Canada, October 1996.

[Franks98] G. Franks, C.M.Woodside, *"Performance of Multi-Level Client-Server Systems with Parallel Service Operations"*, Proceedings of the First International Workshop on Software and Performance, Santa Fe, USA, pp.120-130, Oct. 1998.

[Dilley97] J.Dilley, R.Friedich, T.Jin, J.Rolia, *"Measurement Tool and Modelling Techniques for Evaluating Web Server Performance"* in Lecture Notes in Computer Science, vol.1245, Springer, pp.155-168, R.Marie, B.Plateau, M.Calzarosa, G.Rubino (eds), Proc. of 9-th Int. Conference on Modelling Techniques and Tools for Performance Evaluation, June 1997.

[Hesselgrage98] Mary Hesselgrage, *"Avoiding the Software Performance Crisis"*, Proc. of the First International Workshop on Software and Performance, Santa Fe, USA, pp.78-79, Oct.1998.

[Fowler98] Martin Fowler, *UML Distilled,* Addison-Wesley, 1998.

[Hrischuk96] Curtis. E. Hrischuk *"Implementing Angio Trace Analysis using the Graph Rewriting Tool PROGRES"*

[Hrischuk99] Curtis. E. Hrischuk, C.Murray Woodside, Jerome A Rolia, and Rod Iversen, *"Trace-based load characterization for generating performance software models"* IEEE Transactions on Software Engineering, Vol.25, Nb.1,pp 122-135, January 1999.

[Douglass98] Bruce Powel Douglass, *Real-time UML*, Addison-Wesley, 1998.

[Neilson95] J.E.Neilson, C.M.Woodside, D. Petriu, and S. Majumdar, *"Software bottlenecking in client-server systems and rendezvous networks"*, IEEE

Transactions on Software Engineering, vol. 21(19) pp.776-782, September 1995.

[OMG92] Object Management Group, *The Common Object Request Broker: Architecture and Specification,* Object Management Group and X/ Open, Framingham, MA and Reading Berkshire UK, 1992.

[Petriu98] D. Petriu, X.Wang, *"Deriving Software Performance Models from Architectural Patterns by Graph Transformations"*, Proc. of the Sixth International Workshop on Theory and Applications of Graph Transformations TAGT'98, Paderborn, Germany, Nov. 1998.

[Petriu99] D.C. Petriu, X. Wang *"From UML Descriptions of High-level Software Architecture to LQN Performance Models"*, Proc. of Applications of Graph Transformation with Industrial Relevance, Monastery Rolduc, Kerkrade, The Netherlands, September 1999.

[Rolia87] J.A. Rolia. *Performance Estimates for Multi-tasking Software Systems.* Master's Thesis. University of Toronto, Canada, January 1987.

[Rolia82] J.A. Rolia. *Software Performance Modelling.* Ph.D. Dissertation, CSRI Technical Report#260, University of Toronto, Canada, January 1992.

[Rolia95] J.A. Rolia, K.C. Sevcik, "The Method of Layers", IEEE Trans. On Software Engineering, Vol. 21, Nb. 8, pp 689-700, August 1995.

[Schuerr94] A. Schuerr, *"PROGRES: A Visual Language and Environment for PROgramming with Graph Rewrite Systems"*, Technical Report AIB 94-11, RWTH Aachen, Germany, 1994.

[Schuerr90] 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.

[Schuerr97] A. Schuerr, *"Programmed Graph Replacement Systems"*, in Handbook of Graph Grammars and Computing by Graph Transformation, G. Rozenberg (ed), pp 479-546, 1997.

[Schuerr97] A. Schuerr, *"PROGRES for Beginners"* RWTH Aachen, D-52056 Aachen, Germany

[Schuerr97] A. Schuerr, *"A Guideed Tour through the PROGRES Environment"* RWTH Aachen, D-52056 Aachen, Germany

[Shaw96a] M. Shaw, D. Garlan, *Software Architectures: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[Shaw96b] 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.

[Smith90] C.U. Smith, *Performance Engineering of Software Systems,* Addison Wesley, 1990.

[Shousha98a] C.Shousha, D.C. Petriu, A. Jalnapurkar, K.Ngo, *"Applying Performance Modelling to a Telecommunication System"*, Proceedings of the First International Workshop on Software and Performance, Santa Fe, USA, pp.1-6, Oct.1998.

[Shousha98b] C.Shousha. "*Applying Performance Modelling to a Telecommunication*

System". M.Eng thesis. Carleton University, Ottawa, Sep,1998

[Spitznagel98] B.Spitznagel, D.Garlan, *"Architecture-Based Performance Analysis"*, Proc. of the Int. Conference on Software Eng. and Knowledge Eng. SEKE'98, pp. 146-151, 1998.

[UML] *UML Summary version 1.1* September 1997. *"http://rational.com/UML"*.

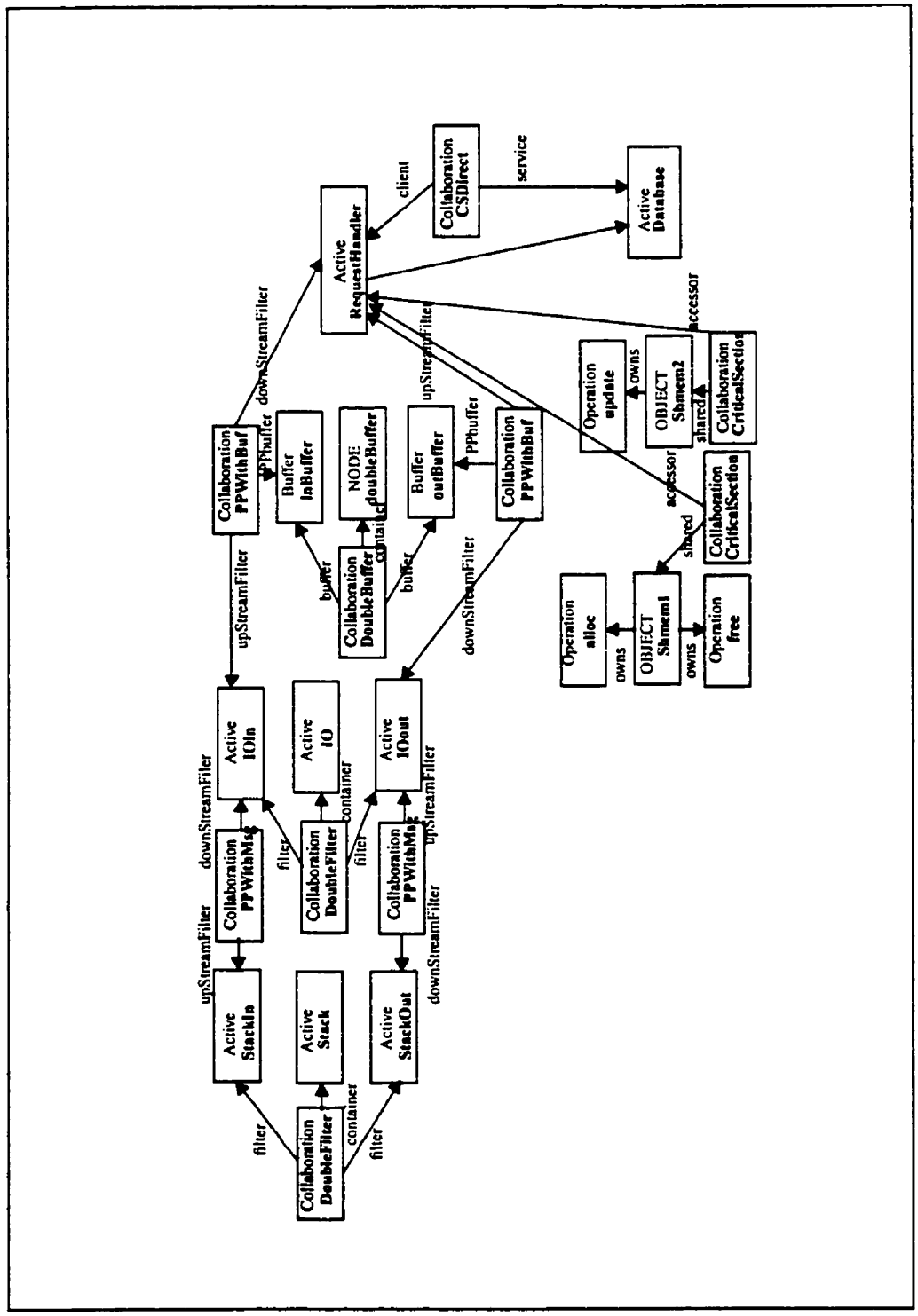
[Williams98] L.G Williams, C.U.Smith, *"Performance Evaluation of Software Architectures"*, Proceedings of the First International Workshop on Software and Performance, Santa Fe, USA, pp.164-177, Oct. 1998.

[Woodside88] C.M. Woodside. *"Throughput Calculation for Basic Stochastic Rendezvous Networks"*. Performance Evaluation, vol.9(2), pp. 143-160, April 1988.

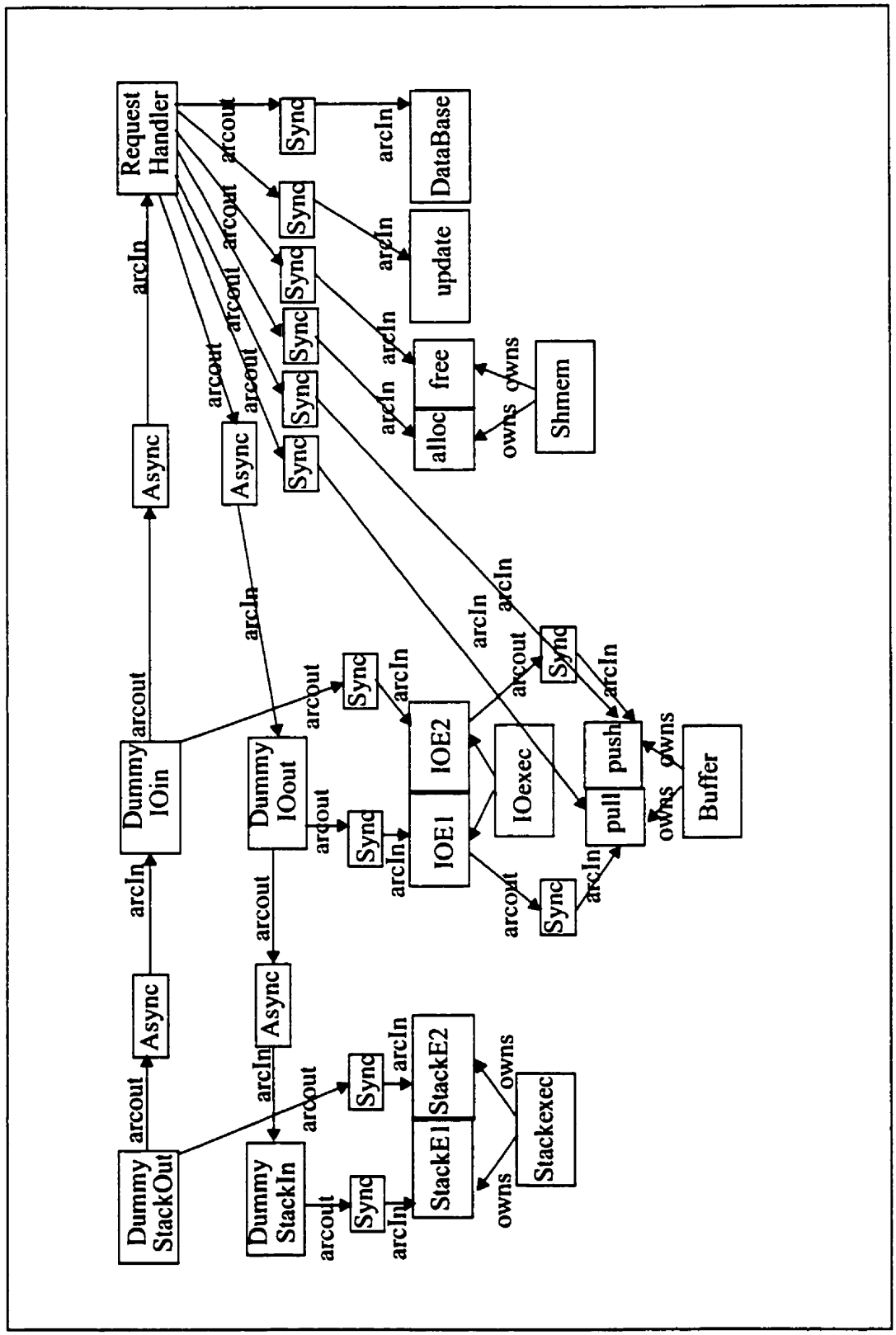
[Woodside95a] C.M. Woodside, J.E. Neilson, D.C. Petriu, S. Majumdar, *"The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software"*, IEEE Transactions on Computers, Vol.44, Nb.1, pp 20-34, January 1995.

[Woodside95b] C.M. Woodside, S. Maj*umdar, J.E. Neilson, D.C. Petriu, J.A. Rolia, A.Hubbard, R.b. Franks "A Guide to Performance Modelling of Distributed Client-Server Software Systems with Layered Queueing Networks"*.

Appendix 1: PROGRES Input Graph for Case Study



Appendix 2: PROGRES Output Graph for Case Study



Appendix 3: LQN Input File for Case Study

The LQN model file for the case study is illustrated below as well. It is generated by the PROGRES program using external functions written in C programming language. This file can be directly used as an input file of an LQN solver.

```
G
0.00001
100
1
0.9
-1
# End of General Information

# Processor Information: No of processors

P 1
p proc1 f
# End of Processor Information

-1

# Task Information: No of Tasks

T  11
t DummyStackOut n   DummyStackOutEntry -1 proc1  m  1
t ShMem2 n   update -1 proc1 m  1
t DataBase n   service1 -1 proc1 m  1
t Stack n   StackOut StackIn -1 proc1 m  1
t DummyStackIn n   DummyStackInEntry -1 proc1  m  1
t ShMem1 n   alloc free -1 proc1 m  1
t Buffer n   BufferRead BufferWrite -1 proc1  m  1
t IO n   IOin IOout -1 proc1 m  1
t DummyIOin n   DummyIOinEntry -1 proc1 m  1
t RequestHandler n   RequestHandlerEntry -1 proc1  m  10
t DummyIOout n   DummyIOoutEntry -1 proc1 m  1
# End of Task Information

-1

#Entry Information: No. of Entries
```

```
E  15
s StackOut 0 0.1 0 -1
s StackIn 0 0.1 0 -1
s BufferRead 0 0.1 0 -1
s RequestHandlerEntry 0 0.1 0 -1
s DummyIOoutEntry 0 0.1 0 -1
s DummyStackOutEntry 0 0.1 0 -1
s DummyStackInEntry 0 0.1 0 -1
s update 0 0.1 0 -1
s DummyIOinEntry 0 0.1 0 -1
s service1 0.8 0 0 -1
s alloc 0 0.1 0 -1
s free 0 0.1 0 -1
s BufferWrite 0 0.1 0 -1
s IOin 0 0.1 0 -1
s IOout 0 0.1 0 -1
a DummyIOout DummyStackOut 0 1 0 -1
a RequestHandler RequestHandler 0 1 0 -1
y DummyStackOut StackOut 0 1 0 -1
a DummyIOin RequestHandler 0 1 0 -1
y DummyStackIn StackIn 0 1 0 -1
a DummyStackIn DummyIOin 0 1 0 -1
y IOout BufferRead 0 1 0 -1
y RequestHandler BufferRead 0 1 0 -1
y RequestHandler service1 0 1 0 -1
y RequestHandler alloc 0 1 0 -1
y RequestHandler free 0 1 0 -1
y RequestHandler update 0 1 0 -1
y DummyIOout IOout 0 1 0 -1
y RequestHandler BufferWrite 0 1 0 -1
y IOin BufferWrite 0 1 0 -1
y DummyIOin IOin 0 1 0 -1
#End of Entry Information

-1
```