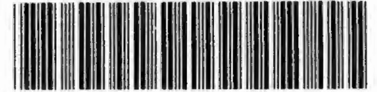
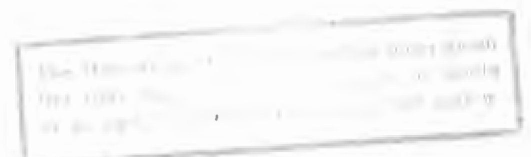


XSNAP : A QUEUEING NETWORK ANALYSIS PACKAGE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Hylton Donnelly
September 1992

Supervised by
Prof P.S. Kritzinger



The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

Abstract

This dissertation describes the design and implementation of a sophisticated X-Windows based modelling package called XSnap, which can be used to solve product-form mixed multi-class queueing networks. A Graphical User Interface allows interactive network specification, whilst the modeller can also define complex network experiments and request customised output through the use of a language called SnapL.

The solution modules used by XSnap are grouped together to form the Calculation Modules ToolBox (CMTB), which can be easily integrated into any modelling package which provides an appropriate user interface. Solution statistics are found using Reiser's Mean Value Analysis (MVA) algorithm, which has been extended to allow for the approximate solution of networks with PRIORITY servers or non-integral closed chain populations. A routing validation algorithm is used to validate the routing information for the network to be solved, and equations defining the relative throughput (or visit ratio) of each class at each centre in the network, are solved using a version of LU-Decomposition called *Crout's method with partial pivoting.*

The dissertation also includes a study of a number of other available modelling packages. The choice of features included in the XSnap GUI has been largely influenced by this study. A number of different algorithms for solving product-form queueing networks are also discussed, and relevant points from this discussion are presented as part of the motivation for using the MVA algorithm for finding solution statistics.

Acknowledgements

For their involvement and association with this study I would like to express gratitude to:

- My fellow graduate students at UCT, who have often helped me in one way or another. Special thanks to Guido Zsilavecz (who taught me to RTFM), Quinton Hoole, Stephen Donaldson, Julian Hansen and Grant Wyatt. Your help was much appreciated.
- My girlfriend, Tan Lowe, who so sweetly offered to wade through the early drafts of this dissertation, correcting grammar and spelling.
- Andrew Hutchison, who has given me valuable feedback regarding MicroSnap.
- Prof. Kritzinger, who has managed to help and guide me in preparing this dissertation, despite being so many miles away.

Contents

Abstract ii

Acknowledgements iii

1 Introduction 1

1.1 What is XSnap? 1

1.2 XSnap and Performance Modelling 1

1.3 Dissertation Overview 2

1.4 The Development Environment 3

1.5 Main Results 3

2 Performance Modelling 4

2.1 The need for Performance Modelling 4

2.2 Simulation and Analytic Techniques 4

2.3 Solution Methods 5

2.3.1 Product-form Queueing Networks 5

2.3.2 Direct Markovian Analysis 6

2.3.3 Convolution algorithm 7

2.3.4 Mean Value Analysis 7

2.3.5 LBANC 7

2.3.6 RECAL 8

2.3.7 Algorithm Extensions 8

2.4 Existing Modelling Packages 8

2.4.1 RESQ 9

2.4.2 HIT 9

2.4.3 MACOM 10

2.4.4 MicroSnap 11

2.4.5 Other packages and references 12

3 The XSnap Solution Algorithms 13

3.1 The MVA Calculation Modules Toolbox (CMTB) 13

3.2 Applications using the CMTB 13

3.3 Models that can be solved by the CMTB 14

3.4 What's in the CMTB? 15

3.5 Model Definition and Manipulation Routines 17

3.5.1 Interfacing with the CMTB 17

3.5.2 Memory Allocation 18

3.5.3 User-defined variables 18

3.5.4 Model data structures 18

3.6 Model Validation and Solving Relative Throughputs 20

3.6.1 Model Validation 20

3.6.2 Relative Throughputs 20

3.7 The MVA algorithm 21

4 Route Validation and Relative Throughputs 23

4.1 Preliminaries 23

4.2 Route Validation 25

4.2.1 Restating Rule 1 26

4.2.2 Restating Rule 2 27

4.2.3 Flooding 27

4.2.4 The Route Validation Algorithm 28

4.2.5 Implementation of the algorithm 29

4.3 Solving relative throughputs 30

4.3.1 Introducing *nodes* 30

4.3.2 One workload at a time 31

4.3.3 Open chains 31

4.3.4 Closed chains 32

4.4 Checking for infinite queues 33

4.5 Solving linear equations 33

4.5.1 Using two solution algorithms 33

4.5.2 The Conjugate Gradient Method 34

4.5.3 LU-Decomposition 35

4.6 Solving sets of linear equations in the MVA algorithm 36

5 MVA and Statistical Measures 37

5.1 Introduction to the Mean Value Analysis algorithm 37

5.2 The Algorithm 37

5.2.1 Outline of the algorithm 38
5.2.2 Storing intermediate results at each population vector $\vec{i}$ 38
5.2.3 Step 1 — Initialisation 40
5.2.4 Step 2 — Looping over all closed population vectors 41
5.2.5 Step 3 — Looping over all closed chains 41
5.2.6 Step 4 — Computing average waiting times $W^*_{ik}(\vec{i})$ 41
5.2.7 Step 5 — Computing the relative chain throughputs $T^*_k(\vec{i})$ 43
5.2.8 Step 6a — Computing $Q_i(\vec{i})$ 44
5.2.9 Step 6b — Estimating W_{ir} for priority service centres i and customers of class r from open chains 44
5.3 Storing the values W_{ir} for PRIORITY centres i and classes r belonging to open chains 45
5.4 Interpolation of the MVA results 46
5.4.1 The interpolation algorithm 46
5.4.2 Collecting the MVA results for interpolation 47
5.4.3 Interpolating the values W_{ir} for PRIORITY centres i and classes r belonging to open chains 48
5.5 Dynamic memory allocation in the MVA algorithm 48
5.5.1 Deleting old solution results 48
5.5.2 Storage for interpolation 49
5.5.3 The function `startup mva()` 49
5.5.4 The function `shutdown mva()` 50
5.6 The function `checkmodelsoln()` in the module "stat41.c" 50
5.6.1 The function `check priority centres()` 50
5.7 Statistical measures 51
5.7.1 Notes on the statistical measures formulae 51
5.7.2 Average queue length 52
5.7.3 Average throughput rate 52
5.7.4 Average waiting time 53
5.7.5 Average queueing time 53
5.7.6 Average cycle time (Closed Chains) 53
5.7.7 Average turnaround or residence time (Open chains) 53
5.7.8 Average utilisation 54

6 Designing the XSnap User Interface 55

6.1 Requirements of the Interface 55
6.2 Constructing Models 56

6.2.1 The Canvas . . . 57
6.2.2 Default Settings . . . 59
6.2.3 Variables and Expressions . . . 59
6.2.4 Workloads . . . 59
6.2.5 Centres . . . 61
6.2.6 Paths . . . 63
6.3 Saving and Retrieving Models . . . 66
6.4 Textual Representation of the Model . . . 66
6.5 Solution Statistics . . . 67
6.5.1 Generating Solutions . . . 67
6.5.2 Viewing Solution Statistics . . . 67
6.5.3 Saving Solution Statistics . . . 68
6.6 Repetitive Model Evaluation . . . 68
6.6.1 Background to the SnapL language . . . 69
6.6.2 The definition and evaluation sections . . . 69
6.6.3 Editing Evaluation Code . . . 70
6.6.4 Parsing Evaluation Code . . . 70

7 Implementing XSnap 71
7.1 The Development Environment . . . 71
7.1.1 Choosing **C** as a programming langauge . . . 71
7.1.2 X-Windows . . . 72
7.2 Components of the XSnap package . . . 73
7.3 The XSnap GUI . . . 74
7.3.1 The XSnap GUI data structures . . . 75
7.3.2 Managing the Canvas . . . 75
7.3.3 Solving the Model . . . 76
7.3.4 Evalution-section code . . . 77
7.4 The SnapL parser . . . 77
7.4.1 Backus Naur Form of the SnapL language . . . 77
7.4.2 Lexical Analyzer . . . 81
7.4.3 The Parser . . . 81

8 Performance and Testing 84
8.1 Performance Analysis . . . 84
8.1.1 Performance criteria . . . 84
8.1.2 MVA Memory Requirements . . . 85
8.1.3 MVA Time Requirements . . . 88

8.1.4 Some sample models 90
8.1.5 Adding Open Chains to the model 90
8.1.6 Relationship between memory and time requirements of the MVA algorithm 91
8.1.7 Solving Relative Throughputs 92
8.2 Testing XSnap 93
8.2.1 Testing the CMTB solution results 93
8.2.2 Testing the XSnap GUI 98
8.2.3 Testing the other modules 98
8.2.4 Results of the testing procedures 99

9 Conclusion **100**
9.1 Successes 100
9.2 Shortcomings 101
9.3 Further Research 101

A Sample Evaluation-Code Output **103**
A.1 Evaluation Code 103
A.2 The output produced by XSnap 104

B Sample output generated using "CMTB Text" **110**

List of Tables

1 MVA memory requirements 86

2 Time and space requirements for solving a number of simple models 90

3 Results for the M/M/1 Queue 95

4 CPU Statistics in the Central Server model 96

5 Statistics for Disk A (or B) in the Central Server model 96

6 Throughput and Solution Times for the Central Server model 96

List of Figures

1 The Calculation Modules Toolbox 16
2 The CMTB model data structure 19
3 The MVA algorithm for solving multiclass open and closed queueing networks. 39
4 The *XSnap* screen with sample model 57
5 A 'zoomed-in' model 58
6 The Workload Manager 60
7 The Centre Manager 62
8 The Path Manager 65
9 Sample workload solution statistics 67
10 Sample centre solution statistics 68
11 The X-Windows Architecture 72
12 Components of the XSnap package 74
13 M/M/1 Queue. 94
14 The Central Server Model. 95
15 Sample Model 'ex.xsn' 98

Chapter 1

Introduction

Welcome to XSnap. This dissertation has been submitted along with the source code for XSnap V1.0 in fulfillment of the requirements for the degree of Master of Science at the University of Cape Town, South Africa.

1.1 What is XSnap?

XSnap is a Queueing Network Analysis package which can be used to analyse product-form multi-class queueing networks. The letters "Snap" in the title stand for Stochastic Network Analysis Package, and the prefix "X" has been added in reference to the fact that the networks are defined and manipulated using a sophisticated Graphical User Interface (GUI) running under the UNIX based X-Windows environment.

XSnap uses the Mean Value Analysis (MVA) algorithm (i.e. an analytical approach) for finding solution statistics to queueing networks.

A number of service disciplines, including priority servers, are supported by the solution modules. Furthermore, both open and closed chains are allowed and these may be combined to form mixed queueing networks.

1.2 XSnap and Performance Modelling

A large number of analysis packages already exist for use in the field of Performance Modelling. These packages are typically used to analyze and to help predict the performance of real systems. In order to carry out such analyses, the real system is normally modelled, and this model is then either simulated or solved using analytical techniques. Modellers typically use modelling packages to carry out this Performance Modelling since the calculations and theory needed to conduct simulations and to find analytical solutions are quite rigorous.

XSnap represents a new tool that can be applied in this field of Performance Modelling. The development of the package has been largely an engineering exercise, with the primary goal being to develop a user-friendly and powerful package based on current knowledge of queueing network product-form solvers. Furthermore, the package has been designed to represent and improvement over similar existing packages.

1.3 Dissertation Overview

Any description of a modelling package would clearly be lacking unless it was complemented by an overview of Performance Modelling in general. The choice of solution algorithms used to solve the underlying queueing network will also be motivated, as will the nature of the GUI that has been developed.

To this end the dissertation has been divided into three parts:

- **Chapter 2 : Performance Modelling and Solution Methods**
  An introduction to Performance Modelling and an overview of different analytical solution algorithms used for solving product-form queueing networks. This part provides interesting background to Performance Modelling in general and includes a description of a number of other modelling packages currently available.

- **Chapters 3, 4 and 5 : The XSnap Solution Modules**
  A detailed description of the solution algorithms used in XSnap including some notes on the implementation of these modules. The first chapter in this section describes how the solution modules were first implemented by the author in another package called MicroSnap. These modules have been grouped together into a toolbox called the MVA Calculation Modules Toolbox (CMTB). Chapter 3 also gives an overview of the CMTB modules and shows how they may be integrated into any modelling package. Of particular interest in this part are the algorithms used for Route Validation, the calculation of Relative Throughputs and Mean Value Analysis.

- **Chapters 6, 7 and 8 : The XSnap graphical user interface**
  The design and implementation of the XSnap GUI. This interface is used to define and manipulate the queueing networks that are solved by the XSnap solution modules. Of particular interest in this section is the design of the Graphical User Interface, and the choice of features that have been implemented to make the package easy to use, practical and attractive. Chapter 8 gives an analysis of the performance of XSnap, and also describes the testing procedures used to test XSnap.

1.4 The Development Environment

The solution algorithms implemented in XSnap were originally written in TURBO_C as part of a package called MicroSnap. This package ran under DOS and is described in more detail in Chapter 3.

When moving to the UNIX environment, these modules were ammended slightly. The X-windows GUI has also been written in C.

The development environment and, more specifically, the choice thereof, will be discussed in more detail later in the dissertation. Suffice it to say, at this point, that the system was implemented using the C programming language under UNIX and X-Windows.

1.5 Main Results

The main result of the study is obviously the finished XSnap package.

Other significant results include the new Routing Validation Algorithm, developed by this author, which can be implemented as part of any similar package.

The implementation of the XSnap package also lends itself well to further extensions, and the solution modules used in XSnap can be easily integrated into other packages (as, indeed, they have already formed part of MicroSnap and XWan, both of which will be described later in this dissertation).

The dissertation also forms a useful reference, describing many factors relevant to the implementation of an analytically based queueing network solver, as well as the design and implementation of a Graphical User Interface.

Chapter 2

Performance Modelling

This chapter will discuss the different approaches that can be taken in Performance Modelling, and will then go on to describe a number of different solution algorithms that can be used to solve product-form queueing networks.

A study of a number of existing modelling packages that can be used to solve product-form queueing networks will also be presented.

2.1 The need for Performance Modelling

Computers and communications are expensive. It is therefore obviously important to be able to quantify the performance of such systems, both those already existing as well as those still being designed.

Analysis of an existing system can be used to predict the efficiency of the system under different workload conditions, and also to model the effect of changes made to the system in an effort to optimise its performance.

When designing new systems, a cost/benefit type analysis can be used to choose between different design alternatives. Typical questions asked while designing a system may include: What is the likely throughput rate, and what can we charge per transaction? Will the use of two disk drives be more effective than the use of a single, more expensive model? What response time can be expected for simple queries in the system? How large must our buffer be for packets in a packet-switched network?

2.2 Simulation and Analytic Techniques

There are three main approaches to Performance Modelling. The first is to build the new or modified machine, the performance of which can then be tested using a standard set of tasks. This is called *benchmarking*. Clearly, this option is very expensive since it may

not always be possible to borrow the new machinery. Also there is a significant overhead in installing and configuring the machine. Strictly speaking, this method does not really fall under the heading Performance Modelling, but rather under the heading Performance Analysis, since no *model* of the system is defined.

The second approach is to write a computer program to *simulate* the new system. This approach has the advantage *and disadvantage* that the system can be written with an arbitrary amount of detail. Unfortunately, such simulations typically include too much low level detail making them slow to run. These simulations also often need to be left running for a good number of hours to allow the system to settle down. Once in a "steady state", the necessary statistics can be captured, and paramaters defining the system can be changed and the simulation allowed to run again so that an impact analysis can be performed.

The third approach to Performance Modelling is to build a *mathematical model* of the system. This model can then be validated and evaluated – usually in a fraction of the time needed to run a simulation. Model parameters can be changed easily making impact analysis investigations easier to perform. This approach does, however, require that the modeller is skilled enough to define a model which can be solved and which captures the significant behaviour of the system.

Although the first of the above three methods would be impractical in most circumstances, is it difficult to motivate convincingly which of the other two methods is the better. Each has its own advantages and disadvantages, the study of which is beyond the scope of this dissertation.

For the remainder of this chapter we will focus on the last of the three approaches (i.e. an *analytical* approach) to modelling computer systems.

2.3 Solution Methods

This section will describe a number of different solution algorithms that are available for solving product-form queueing networks.

The purpose of this section will be to allow the reader to compare the capabilities, advantages and disadvantages of each of the solution algorithms.

2.3.1 Product-form Queueing Networks

For many years now, the predominant technique used for analytical modelling has been the use of queueing theory.

The most common queueing networks are called BCMP-networks, named after Baskett, Chandy, Muntz and Palacios-Gomez who are the four authors of the seminal work on the

topic (see [Baskett 75]). These networks have also come to be known as *product-form* networks since the solution of the queueing network is given by

$$\Pr(S = (k_1, k_2, k_3, \ldots, k_M)) = \frac{1}{G} d(S) f_1(k_1) f_2(k_2) f_3(k_3) \ldots f_M(k_M) \qquad (1)$$

where $\Pr(S = (k_1, k_2, k_3, \ldots, k_M))$ is the probability that the network is in a steady state $(k_1, k_2, k_3, \ldots, k_M)$, $d(S)$ is a function of the number of customers in the system, and $f_i(k_i)$ is a function which depends on the customers present at node i and the service discipline of the node. G is called the normalisation constant and has an arbitrary value ensuring that the probabilities $\Pr(S = (k_1, k_2, k_3, \ldots, k_M))$ sum to 1.

Some aspects of real systems cannot be modelled accurately using *exact* product-form queueing networks. Examples of these are:

- batch arrivals of customers
- forking or joining of customers
- congestion leading to blocking and customer loss
- state dependent routing
- simultaneous resource possession

None the less, product-form queueing networks can be used to represent most features of real systems and are therefore still a popular vehicle for Performance Modelling.

2.3.2 Direct Markovian Analysis

The theory of mixed multiclass product-form queueing networks is well established, and the form of the probability function for a particular state of the network (as shown above) is well known. However, *solving* the probability density function for a particular network is a different problem altogether, and is a purely computational one.

Since product-form queueing networks are made up of Markov Chains they can, admittedly, be solved using direct Markovian Analysis (MACOM[1] is one example of a tool which uses this method). However, one could quite arguably call this the naive approach since in the larger models the state-space can easily become too large to use this method. Even if the state probabilities *could* be calculated for all states in a large model, the accumulated round-off error in the normalistaion constant G would make the results meaningless.

A significant redeeming feature of the method is that one is able to encompass a much wider modelling domain which includes some of those features listed in the previous section, such as forking or joining of customers; these features cannot be represented in BCMP networks. The method cannot, therefore, be altogether disregarded.

[1] MACOM is decribed in more detail in section 2.4.3.

2.3.3 Convolution algorithm

The Convolution Algorithm was developed by Buzen [Buzen 73] and is concerned primarily with the calculation of the normalisation constant G shown in equation 1. Fortunately, all performance measures that are of interest can be calculated as a side effect of calculating G.

Although this was one of the first really fast algorithms, the algorithm often suffered from numerical overflow when implemented. Futhermore, most of the calculation effort goes into the valuation of normalisation constants which have no simple interpretation in terms of the system being modelled.

2.3.4 Mean Value Analysis

In 1978 Martin Reiser [Reiser 81] proposed a solution algorithm which is now known as Mean Value Analysis, or MVA for short. This algorithm proved *at worst* to be as computationally inefficient as the Convolution Algorithm, and moreover did not suffer from any under or overflow problems.

MVA depends on Little's theorem and on the so-called 'Arrival Theorem' which states that a job in a closed queueing network, when it enters a queue, observes the mean state of the queue with itself removed. One of the other main differences between Convolution and MVA is that while the main iteration in the Convolution algorithm is over *nodes*, the main iteration in MVA is over different *network populations*.

For networks consisting of only load-independent servers or server-per-job nodes, the MVA algorithm is easier to implement than Convolution. Unfortunately, MVA is unsuitable for networks with load-dependent servers, due to errors introduced through the subtraction of nearly equal numbers. Reiser has addressed this problem by producing another algorithm called Augmented MVA. This algorithm requires that MVA is performed 2^m times when solving the model, where m is the number of queue-dependent servers in the model.

A number of other extensions have been made to MVA over the years, and these include tree-MVA and Mean Value Analysis by Chain (MVAC). Readers are referred to the excellent book by Peter King [King 90] for more information regarding MVA.

2.3.5 LBANC

Yet another algorithm used for solving product-form queueing networks is LBANC, which stands for "Local Balance algorithm for Normalising Constants" (see [Chandy 80]).

This algorithm is similar to Convolution in that it is used to calculate the normalisation constant G. However, the main iteration in the algorithm is over the different

customer populations (as for MVA) rather than over the different nodes in the network (as for Convolution), and LBANC therefore has a mixture of the properties of MVA and Convolution.

The algorithm is perculiar in that it works with *unnormalised* probabilities, which are the probabilities given by equation 1 when G equals 1. Unfortunately, this algorithm suffers from the same numerical problems as MVA when dealing with load-dependent servers.

Lam [Lam 83] has shown that each of the Convolution algorithm, MVA and LBANC is derivable from any one of the others. This confirms that they are all equally powerful.

2.3.6 RECAL

RECAL (Recursion by Chain method) is one of the more recent algorithms derived for solving product-form networks. This algorithm was discovered by Conway and Georganas (see [Conway 86]) in 1986, and relates the normalisation constant G of a network with r classes, to the normalisation constant with one class removed.

When compared to RECAL, the MVA algorithm has still proved to be equally as efficient. Readers are referred to the study done by Renate Schmidt [Schmidt 89] on the comparison between RECAL and MVA when implemented under MicroSnap.

2.3.7 Algorithm Extensions

Each of the above algorithms have been extended in numerous ways over the years. Of particular relevance are the extensions to allow for the solution of Mixed Networks, which are networks consisting of both open and closed chains. A closed chain is a chain with a finite customer population, while an open chain accepts customers from and releases customers to the external environment.

2.4 Existing Modelling Packages

There is currently quite an array of different modelling packages available which have been designed to solve models depicting computer performance. Clearly, it would be impractical (and futile) to try to list them all here. Rather it has been decided to describe only four of these packages, enough to give one a flavour of the different types of package available. Only packages which can be used to find analytical solutions to models have been chosen.

2.4.1 RESQ

An interesting alternative to finding a *numerical solution* to a mathematical model of a queueing network, is to *simulate* the queueing network. It should be noted that this is not the same as a simulation of the actual *system* itself.

The Research Queueing Package (RESQ) [Sauer 81] can be used to simulate *extended* product-form queueing networks. These extended networks allow simultaneous resource possession, parallelism and synchronisation and are therefore, strictly speaking, not product-form networks at all.

One can use extended queueing networks to distinguish between *passive* and *active* queues and to allow for the forking and joining of customers.

When using RESQ, one can specify the models to be solved by using either an interactive dialogue facility, or by constructing a description of the model using a model specification language. Unfortunately, the user interface is sadly unsophisticated and it is often difficult to follow the logic of model specifications. One is thus very likely to make mistakes, especially when defining large models.

2.4.2 HIT

HIT is a comprehensive modelling tool designed to allow the model-based evaluation of the performance of computer systems.

HIT has been under development at the Universität Dortmund since 1983, and is continually being updated [Beilner 90]. An especially interesting feature of the package is that it allows the modeller to solve a model using any one (or even a combination) of different evaluation techniques. The model specification is independent of the technique to be used and the modeller can therfore apply any of the different solution techniques to the same single model specification.

The different evaluation techniques currently offered include:

- exact product-form solutions for seperable queueing networks.
- numerical evaluation for Markov Chain representations of general models.
- approximation techniques for large seperable networks.
- approximations for certain non-seperable networks including multiclass FCFS and priority stations.
- stochastic discrete-event simulation with appropriate statistical result estimation.

Sub-model analysis is also possible, and different evalution techniques can be used for each of the different sub-models.

HIT models are defined using a specification language called HI-SLANG. A seperate utility called HITGRAPHIC can be used to automatically generate HI-SLANG code, thereby enabling interactive model construction. However, HITGRAPHIC can only be used to specify the higher level components of the model, and the more detailed, specialised characteristics of the model need to be included using HI-SLANG. This is actually quite an attractive approach since the modeller is able to benefit from the high-level view offered by HITGRAPHIC, without sacrificing any of the power or flexibility offered by the HI-SLANG specification language.

2.4.3 MACOM

MACOM is a software tool being developed at the Universität Dortmund by Sczittnick and Muller-Clostermann, and the Deutsche Bundespost [Sczittnick 90]. This package is used to solve performance models of complex computer systems.

MACOM runs under SunView, and allows the modeller to build up a graphical representation of the model to be solved. This package uses direct Markovian Analysis to solve models and thus a large set of simultaneous linear equations needs to be solved in order to find solution statistics.

By using direct Markovian Anaylsis, MACOM has been able to model systems which include features which cannot be modelled using exact product-form queueing networks. Such features include forking or joining of customers, and congestion leading to customer loss. Unfortunately, a major weakness of the approach is the fact that models with too many states can easily become intractable.

Small systems of equations are solved using an algorithm developed by Grassmann, Kumar and Billinton [Grassmann 87]. This algorithm avoids the subtraction of near equal numbers and is therefore more stable than the conventional Gaussian elimination method. For larger matrices, a Gauss-Seidal technique, or successive over-relaxation is used. While the results generated by MACOM are theoretically exact, in practice the potential for rounding errors is significant.

Since MACOM uses Direct Markovian Analysis it is able to accomodate models which include state-dependant routing, and centres in the model can have any one of a wide array of service disciplines. Blocking can also be modelled since the modeller is able to specify the capacity (i.e. the number of customers that can be held simultaneously) of each server. Arrivals to a network can have exponential or phase-type Coxian interarrival time distributions.

One of MACOM's predominant features is its attractive graphical user interface. This interface is used to create, edit and solve models. A discussion of the strengths and weaknesses of this interface is given in chapter 6 along with the motivation for the features

that have been implemented in the XSnap graphical user interface.

MACOM does not support closed chains.

2.4.4 MicroSnap

MicroSnap has been developed at the University of Cape Town as a tool for modelling computer systems, and can used to solve models consisting of multi-class mixed product-form queueing networks. Although MicroSnap was originally written as a PC-based system, it has also been ported to UNIX.

MicroSnap is in fact an *interpreter* of a high level programming language called SnapL (Stochastic Network Analysis Programming Language) [2]. The MicroSnap interpreter is batch-oriented, and takes as input a program written in SnapL, interprets it and produces output resulting from that program. If the program is syntactically and semantically valid, and furthermore specifies a valid queueing network, then the results produced by the program may be in the form of tables, reports etc. as specified using high-level language constructs in the program.

If the program does not meet any one of these conditions then error information detailing the error(s) detected will be produced. The user is offered various options in terms of the statistics produced and whether the results should be either printed or plotted. MicroSnap is described more fully in the MicroSnap User Manual [MSnap 90].

Each SnapL program used in MicroSnap is divided into a *definition-section* and an *evaluation-section*. The former contains a specification of the model, whilst the latter can contain loops so that the model is repeatedly modified and re-evaluated. The evaluation section can also be use to produce customized output. A SnapL program can be edited using any text editor.

The first version of MicroSnap was developed in 1987, but was unfortunately not completely robust and would fail when used with some models. This version was also relatively slow and very space innefficient.

The most recent version of MicroSnap was written by this author entirely from scratch, using new data structures, lexical analyzer and parser. The solution modules developed for the package are also more comprehensive than those offered by earlier versions. In particular, the latest version includes an original routing validation algorithm developed by this author. Furthermore, the implementation of the MVA algorithm in the latest version allows for the solution of open chains and the approximate solution of networks having PRIORITY servers; these features of the MVA algorithm do not represent original work, but are rather an engineering success in that they are especially tricky to implement

[2]SnapL is introduced more formally in Chapter 6. A subset of the language is used in XSnap to allow the user to modify and solve models in batch mode.

efficiently.

Although MicroSnap is a very fast and accurate tool, it has often been critisized for its primitive user interface. Models can only be solved using batch-runs, and no facility exists to allow *interactive* model development. The MicroSnap parser is also relatively in-advanced and will simply abort on finding even the first error (after the error has been reported, of course). The modeller would then need to correct the SnapL program, and then re-run MicroSnap.

2.4.5 Other packages and references

As mentioned before, there are many other modelling packages that do exist; some of which, such as QNAP, are widely used.

Literature describing these packages is somewhat scattered over various journals and conferences, but the proceedings of the "Tools and Techniques" conferences would provide readers with a useful starting point; see [Pooley 92], [Balbo 92] and [Tuig 89].

Chapter 3

The XSnap Solution Algorithms

The preceding chapter has described a number of different methods that can be used to solve queueing networks. XSnap uses an analytical approach which has been achieved using the MVA Calculation Modules Toolbox (CMTB).

3.1 The MVA Calculation Modules Toolbox (CMTB)

The solution modules used by XSnap were first implemented by the author as part of MicroSnap.

Of importance is that the solution modules were written so that they could be used by any appropriate interface. One may like to view this set of modules as a toolbox that can be integrated quickly and easily into any application to be used to find an analytical solution to queueing networks.

The CMTB includes procedures that allow the calling interface to define, query and modify the model and, of course, request solution statistics. The toolbox also ensures that the defined model is valid and will return relevant error messages to the calling procedures when appropriate.

3.2 Applications using the CMTB

As evidence of the portability of the CMTB, the solution modules have already been successfully used by three applications – MicroSnap, XWan and XSnap.

XWan[1] is an X-Windows based application used to model the performance of Wide Area Networks. However, the package is different to XSnap in that only a subset of the

[1] XWan was developed by Andrew Hutchison of the University of Cape Town and formed part of his dissertation for the degree of Master of Science.

features offered by the toolbox have been used and the XWan application interface restricts the types of networks that can be defined.

3.3 Models that can be solved by the CMTB

The CMTB is able to solve multi-class queueing networks consisting of open and/or closed chains in the same network.

Resources in the real system are represented as *centres* in the model, and the load by *customers* within the network. Customers are grouped into *chains* which may be open or closed. Open chains accept customers from (and release customers to) the external environment, while closed chains each have a finite customer population. Customers within each chain can be of different classes, and can change class whilst routing between the different centres in the model.

Centres can be of one of the following service disciplines:

- **FCFS** (First-Come-First-Served)
  Customers receive service in the same order that they enter the queue.

- **LCFSPR** (Last-Come-First-Served-Preemptive-Resume)
  Customers arriving at the centre enter service immediately. The customer receives service until completion of its request or until another customer arrives at the centre. If another customer arrives, the customer which was in service is taken out of service and put at the front of the queue of customers waiting to be served. Once the interrupting customer has completed service, the customer at the front of the queue then resumes service.

- **PS** (Processor-Sharing)
  Customers arriving at a centre receive service immediately. However, if n customers are all being served, then each customer receives service at $1/n$ times the rate at which one customer would receive service.

- **DELAY** (Infinite Server)
  This service discipline assumes that there is always a server available to serve each customer at the centre. The customer will only be delayed for a time equal to the service it demands at the centre, and no longer.

- **MICRO**
  This is not a service centre in the true sense since customers do not queue or receive service at the centre. Rather, the centre is used merely as a reference point in the network, usually to measure the throughput on a particular route.

- **PRIORITY**
  Two levels of priority are supported (i.e. customers are either priority customers or they are not). Customers are served on a FCFS basis taking into account their priority status. Since BCMP networks do not support priority servers, the results are approximate.

Neither state-dependant routing nor stats-dependant service requirements are supported by the CMTB.

3.4 What's in the CMTB?

The remainder of this chapter describes briefly each of the algorithms and modules that make up the CMTB, while the remaining chapters in this part [2] of the dissertation describe in more detail the mathematical bases underlying each of the algorithms and give detailed notes on the implementation of these in the CMTB.

The functions provided by the CMTB can be divided into 4 main groups:

- functions to allow the definition and manipulation of models.
- functions to validate the routing information, and to find the relative throughput of each class at each centre in the model.
- functions to return solution statistics for the defined model.
- functions to erase an old model definition so that a new model can then be defined.

Although each of these components of the CMTB will be described in the following sections of this chapter, chapters 4 and 5 will be used to describe more fully the algorithms for route validation, solving relative throughputs and Mean Value Analysis. The remaining sections of the this chapter, therefore, are meant to provide only a top-level view of the CMTB.

A diagrammatic view of the CMTB is given in figure 1. From this diagram it can be seen that there are 7 modules in the CMTB, four of which provide interfacing functions to the rest of the modelling package.

Model definition and manipulation is performed by "book41.c" which is responsible for setting up (or modifying) the CMTB representation of the model using the CMTB data structures.

The route validation algorithm is also contained in "book41.c", and the relative throughputs are solved by "through.c". The module "ludcmp.c" is used to perform LU-Decomposition on the set of simultaneous linear equations defining relative throughputs.

[2] Chapters 3 through to 5. See Chapter 1, the Introduction.

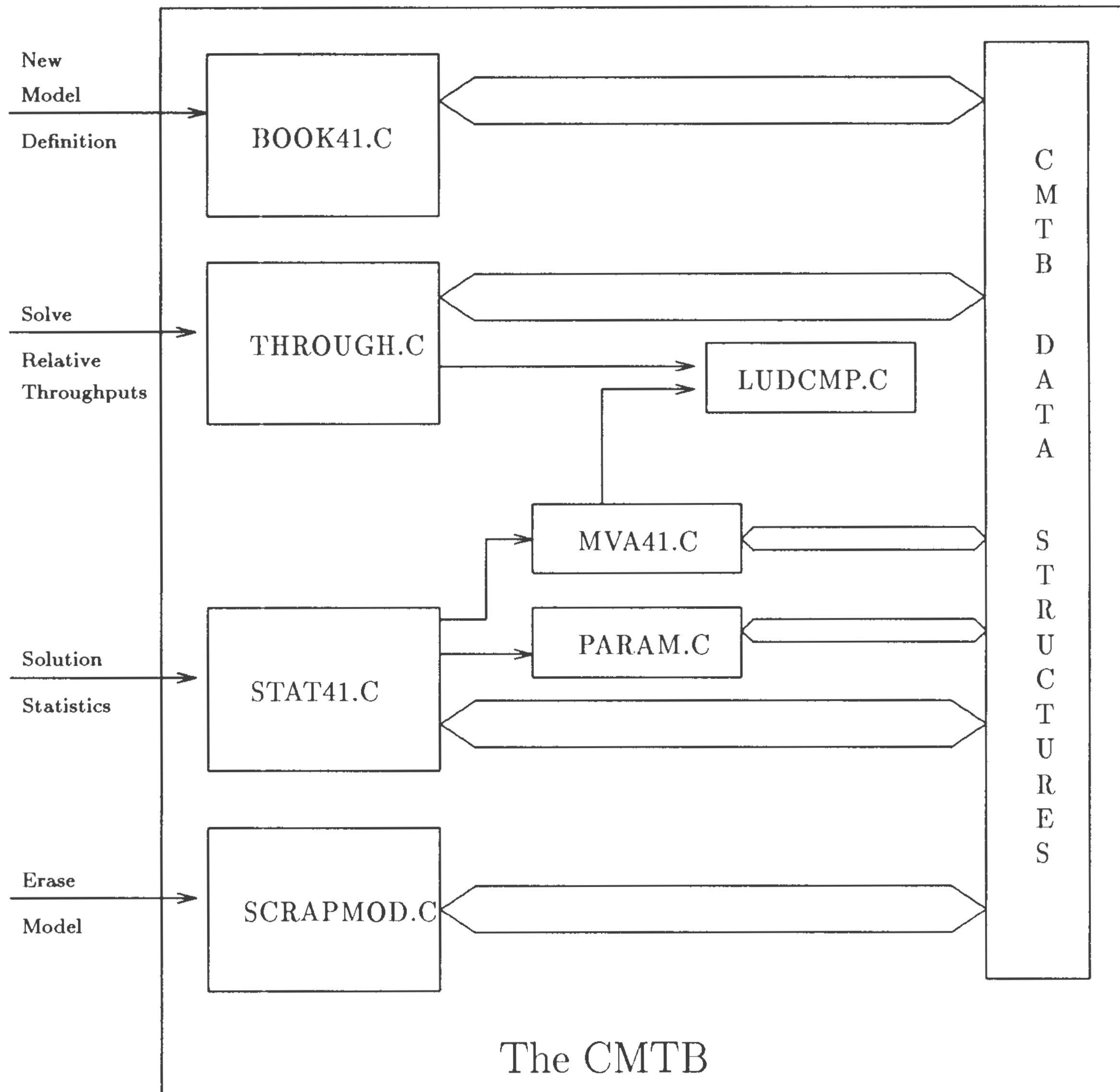


Figure 1: The Calculation Modules Toolbox

Once a model has been defined, solution statistics can be requested using functions provided by the module "stat41.c". This module will invoke the MVA algorithm contained in "mva41.c" (unless the model has already been solved) and return the statistics calculated. "param.c" is used to perform some preliminary calculations before the MVA algorithm is initiated.

When solving models which include both PRIORITY servers and open chains, the MVA algorithm needs to solve numerous sets of simultaneous linear equations. These are solved using the LU-Decomposition algorithm provided by the module "ludcmp.c".

A final module called "scrapmod.c" has been included to allow the application to discard a previously defined model. This module did not form part of the original CMTB implemented under MicroSnap, since that package would only be used to solve a single model each time the program was run.

3.5 Model Definition and Manipulation Routines

Before a model can be solved, it needs to be represented in the CMTB data structures. A special 'bookkeeping' module called 'book41.c' included in the CMTB is responsible for populating these data structures.

The following paragraphs describe briefly the interaction between the user interface and the CMTB 'bookkeeper', and also give an overview of the data structures used by the CMTB to represent a model.

3.5.1 Interfacing with the CMTB

Models are built up piece by piece by making appropriate calls to functions in "book41.c".

Functions are provided for adding workloads (chains), centres and routing information to the model. Seperate functions are used to *change* the actual model parameters of each of these defined workloads, centres or paths, so that one can also modify an existing model using these routines. Each function used to define or modify a model includes appropriate checks to ensure that the model is not invalid. For example, one cannot define service requirements for a class of customer that has not be included in any workload definition in the model.

Information is often passed to the CMTB using linked lists. These lists are easy to contruct using special routines provided by the CMTB, and are especially convenient for user interfaces which rely on a textual model specification which needs to be parsed. A number of different types of lists are provided to carry different kinds of information (eg. strings or values). These lists are replaced by dynamically allocated arrays in the CMTB data structures which are more space efficient.

In the CMTB, all centres and classes are allocated centre and class numbers. Functions are provided for converting these numbers into centre names or workload:class names, and vice versa. These numbers (rather than names) are used when requesting solution statistics.

3.5.2 Memory Allocation

"book41.c" provides a special function to be used for dynamic memory allocation which keeps an account of memory usage by the system. In this way the CMTB is able to provide a summary of the amount of memory used by the calculation routines and by the application as a whole.

3.5.3 User-defined variables

The CMTB also includes a number of routines for manipulating user-defined variables, and will keep a symbol table of all variables and allocate space for storing the values of these variables. These functions can be used by any interface when parsing expressions containing user-defined variables. Such expressions would commonly be used to define service rates, routing probabilities or the like.

3.5.4 Model data structures

Figure 2 gives a diagrammatic representation of the CMTB data structures used to store the model. Centres and workloads in the model are stored in alphabetically sorted binary trees, which can be searched quickly using a binary-search algorithm. Once a model has been defined, an additional mapping is created to each of the centres and workloads using arrays of pointers indexed by centre and workload numbers. This allows the solution algorithms to run more quickly.

Routing information is stored using a sparse matrix representation to take advantage of the fact that nodes in a network are very rarely fully connected. This results in a significant space saving in the CMTB.

Service rates for the different classes at each centre in the model are included with the information defining each centre. This information is also copied to a single array holding *all* the service rates, primarily to speed up the solution algorithms.

Solution statistics calculated using the MVA algorithm are tagged onto the model data structure so that they may be used by the functions in "stat41.c".

It should be noted that any user interface (such as the GUI in XSnap) that requires a more sophisticated representation of the model, has to use its *own* data structures when

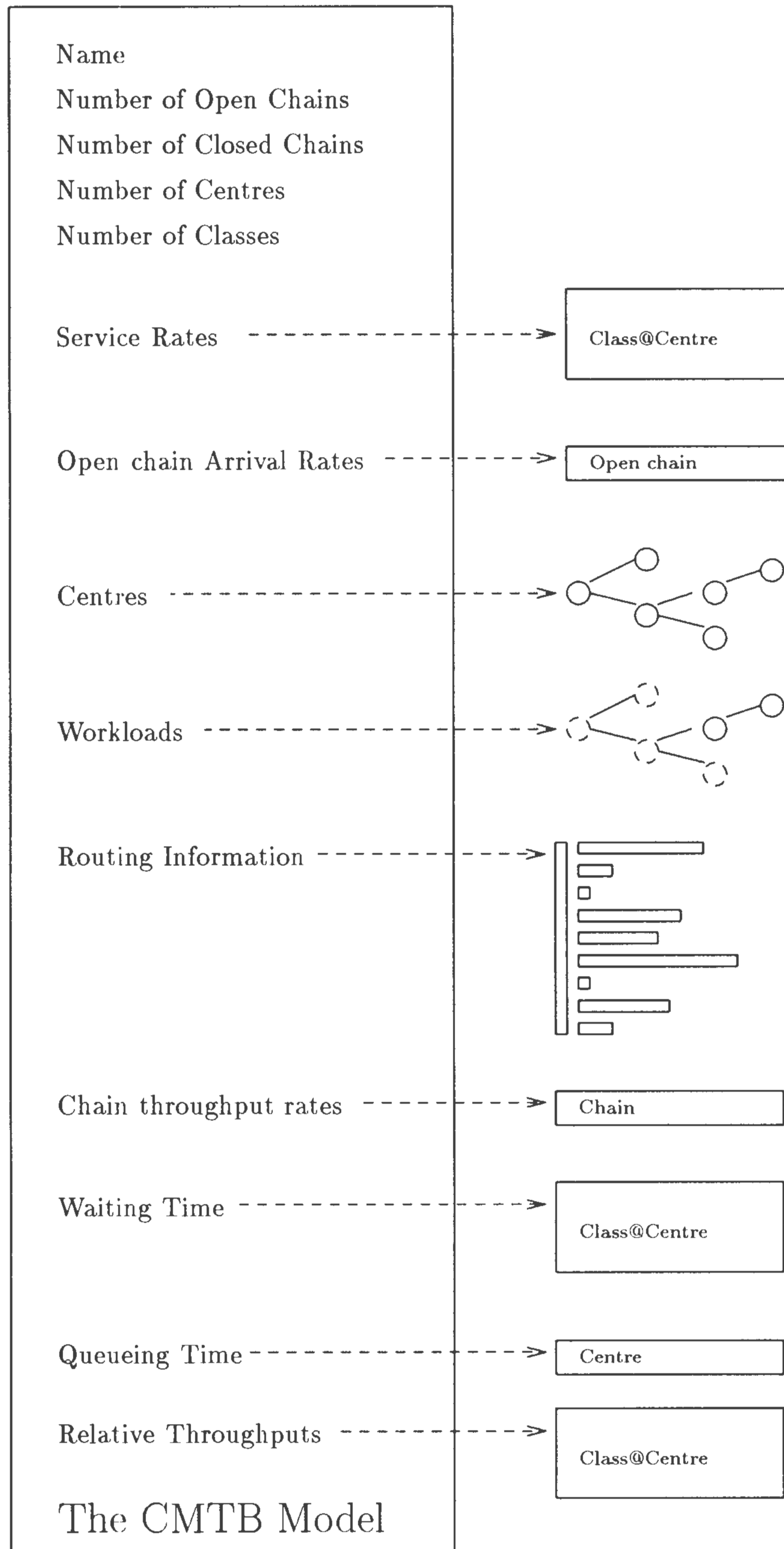


Figure 2: The CMTB model data structure

building and manipulating models. The model is then simply 'copied' to the CMTB before it is solved, by using the appropriate functions provided by "book41.c".

3.6 Model Validation and Solving Relative Throughputs

Once a model has been set up, we need to firstly validate the routing information, and then solve the relative throughputs. This must be done before the MVA algorithm can be used to find any solution statistics.

3.6.1 Model Validation

A route validation algorithm is needed within the CMTB to ensure that the defined routing information is valid. Examples of invalid routing descriptions include:

- Nodes in a closed chain which can be reached from the external environment.
- Routes from a node in a closed chain to the external environment.
- Nodes in an open chain which cannot be reached from the external environment.
- Nodes in an open chain which have no route to an exit point.
- Nodes which are visited in a network, but which are never left.
- Nodes which are left, but never visited.

If a model has invalid routing information, then the relative throughputs of each class at each centre in the model has no unique solution.

In section 4.2, an algorithm is presented which can be used to check the validity of a network's routing information. A significant feature of the algorithm is that it is $O(n)$ where n is the number of nodes in the model. The implementation of this algorithm in the CMTB is also described.

3.6.2 Relative Throughputs

Central to the solution of product-form queueing networks is the solution of the relative throughputs (or the visit ratio) of each class at each centre in the network. This information is needed by the MVA algorithm before any solution statistics can be found.

The solution of these relative throughputs involves solving a set of simultaneous, sparse linear equations. Two methods for solving such a set of simultaneous equations have been implemented in the CMTB. They are:

- **The Conjugate Gradient Method**
  This method takes advantage of the sparse-nature of the set of linear equations to be solved and requires *no extra storage*. Unfortunately, although very quick, this method generates an unacceptably large sum of square errors when used on large matrices (say, 100 by 100 elements). The algorithm has been retained, none the less, since it is apparently possible to increase the accuracy by manipulating the original matrix. This would require further research beyond the scope of this dissertation. In any event, the method is available as an option, and can be used by setting the appropriate boolean variable in the CMTB.

- **LU-Decomposition**
  Due to the 'failure' of the Conjugate-Gradient method, another algorithm has been implemented in the CMTB. This algorithm is called LU-Decomposition, and the version of the algorithm used in the CMTB is called *Crout's method with Partial Pivoting*. Although this algorithm does require extra storage, and does not take advantage of the sparse-nature of the set of simultaneous linear equations to be solved, it is nevertheless very accurate. The algorithm is therefore the default for solving relative throughputs in the CMTB.

Both the Conjugate-Gradient Method and LU-Decomposition are discussed in [Recipes 87], and their implementation in the CMTB is described more fully in section 4.3.

3.7 The MVA algorithm

The CMTB uses the Mean Value Analysis algorithm for finding solution statistics for queueing networks. The discussion given in the previous chapter has shown that MVA still represents one of the better algorithms for solving product-form queueing networks, especially those with no state-dependent routing. MVA overcomes the numerical under and overflow problems often experienced with Convolution and is no less efficient than any of the other algorithms presented.

The MVA algorithm implemented in the CMTB can be used to solve multi-class mixed queueing networks. The algorithm has also been extended to allow for non-integral population levels in closed chains, and to calculate approximate solutions when the model includes centres with PRIORITY service disciplines [3].

The implementation of the MVA algorithm is described in detail in Chapter 5 of the dissertation. Significant features of this particular implementation of MVA are:

[3]The reader will remember that PRIORITY servers cannot be represented exactly using product-form queueing networks.

- the optimal use of dynamically allocated storage for recording intermediate solution results.
- the integration into the algorithm of approximation methods for PRIORITY servers.
- the interpolation algorithm used to find solutions for networks involving chains with non-integral MPL levels.

Chapter 5 also sets out the mathematical formulae used for finding solution statistics based on the MVA solution results. Such statistics include average utilisations, queue lengths, throughput rates, waiting times, queueing times, cycle times and residence times. The functions calculating these statistics are included in "stat41.c".

Chapter 4

Route Validation and Relative Throughputs

This is the first of two chapters which will describe the algorithms implemented in the CMTB in more detail.

This chapter describes the algorithms for

- route validation, and
- computing the relative throughput solution.

Solution of linear equations in general is also discussed in this chapter.

4.1 Preliminaries

This section will introduce some of the notation used throughout the following sections. The notation adopted in this dissertation is very similar to that in [Schmidt 89] and [Krit 82], and the equations presented in this section are also taken from these works referenced.

There are N service centres and J closed chains. There are K chains and R classes in total.

A centre can have any one of the following disciplines:

- FCFS (first come first served)
- LCFSPR (last come first served, preemptive resume)
- PS (processor sharing)
- D (delay)

- R (priority server – there are only 2 priority levels)
- M (micro server)

Let $\vec{L} = (L_1, \ldots, L_J)$ denote the closed chain population vector where L_k is the integral number of customers in chain k.

The routing of customers is defined by a transition matrix given by the probabilities $p_{ir;js}$, where $p_{ir;js}$ denotes the probability of a customer of class r at centre i, moving instantaneously to centre j as a member of class s, after being served.

The visit ratio ξ_{ir} (also called the relative throughput) of class r at centre i is given by

$$\xi_{ir} = p_{out;ir} + \sum_{j=1}^{N} \sum_{s=1}^{R} p_{js;ir} \xi_{js} \tag{2}$$

where $p_{out;ir}$ is the probability that a customer of class r will route to centre i on its arrival into the network. For classes r belonging to closed chains the value $p_{out;ir}$ is zero for all (i, r).

Note that for closed chains all the values $p_{out;ir}$ are zero and that the values ξ_{ir} could therefore all be multiplied by a scaling factor without affecting the validity of the above balance equations. In order to obtain a solution we need to introduce an extra equation. I have added the constraint $\sum_{i=1}^{N} \sum_{r \in E_k} \xi_{ir} = 1$, where E_k is a closed chain. Another way to obtain a solution would have been to set one of the values, ξ_{11} say, to some arbitrary constant and then to derive each of the other ξ_{ir} values using the balance equations. Since we are interested only in the *relative* values its makes little difference which method is used.

The following intermediate results must also be defined:

$$\rho_{ik} = \sum_{r:(i,r) \in E_k} \gamma_{ir} \tag{3}$$

$$\gamma_{ir} = \begin{cases} \frac{\xi_{ir}}{\mu_{ir}} & \text{for } E_k \text{ a closed chain} \\ \lambda_k \frac{\xi_{ir}}{\mu_{ir}} & \text{for } E_k \text{ an open chain} \end{cases} \tag{4}$$

$$\Xi_{ik} = \sum_{r:(i,r) \in E_k} \xi_{ir} \tag{5}$$

Finally we need the equation

$$\rho_i = \sum_{k=J+1}^{K} \rho_{ik} \tag{6}$$

Although many of the above results are stored by the CMTB for use in further calculations, the values γ_{ir} are not stored. These can be easily calculated as needed, and

would also take up much more space than any of the other values if they were stored. The other values ρ_{ik} , ρ_i and Ξ_{ik} are calculated and stored by the module "param.c".

`rho_chain[i][k]` ("param.c")	stores the values ρ_{ik}
`rho_centre[i]` ("param.c")	stores the values ρ_i
`xi_chain[i][k]` ("param.c")	stores the values Ξ_{ik}

4.2 Route Validation

It was decided that a "Route Validation Algorithm" should form an integral part of the CMTB to ensure that the model definition (and solution results) are valid in all instances. Since no existing algorithms for validating queueing networks could be found by this author, the algorithm presented in this section has been developed from scratch, and therefore represents original work.

For convenience we will consider class@centre combinations as nodes. Routing information is represented as directed paths between nodes. In the following discussions we refer only to those class@centre combinations for which routing information exists. If a class does not visit or leave a centre then that class@centre combination (or node) does not exist in the chain.

Before a model can be solved, the CMTB must ensure that the routing information is valid. The rules defining the validity of a network are:

- Rule 1 (closed chains)

  In a *closed* chain we require that each node must be connected by some route to *all other nodes* in the chain, otherwise there can be no steady state solution for the network. To show this, we consider a chain with some node a that cannot be reached by any path from some other node b in the chain. Under this scenario, any customer visiting node b will be forever trapped in that part of the network which excludes node a. Over time, as more and more customers visit node b, they too will become trapped in that part of the network which excludes node a, until eventually no customers are able to visit node a ever again. Every time another customer visits node b for the first time, thus becoming "trapped", the nature and relative loads in the different parts of the network will change. The state of the network is thus always changing until no more customers are able to route to node a. Clearly, it could take an almost infinite amount of time for this state to be reached, and there is therefore no useful "steady-state" solution for the network.

- Rule 2 (open chains)

  Each node in an *open* chain must be connected by some route to at least one *exit*

point, and must be reachable from an *entry point*. If this were not true then either customers entering the network would not be able to reach the given node, or those that did visit the node would never be able to leave the network.

- Rule 3 (closed chains)
  By definition, *closed* chains cannot accept customers from the external environment, and cannot release customers to the external environment.

Should the model route descriptions prove invalid the model cannot be solved. A valid routing definition also ensures that the relative throughputs matrix (described in the next section) has a unique solution.

The most obvious approach to testing Rule 1 (or Rule 2 in the case of open chains), would be to write a function that could find all the nodes (or exit points in the case of open chains) that could be reached from a given start node. One could then use this function repeatedly with each different node in the chain as the starting node. At each call, the function would need to visit *all* the other nodes in the network, and this approach is therefore $O(n^2)$.

While this may be the easiest and obvious, it is definitely not the best approach. The CMTB actually uses an $O(n)$ algorithm to show that a network satisfies all of Rules 1 to 3. This algorithm is derived below.

4.2.1 Restating Rule 1

To understand the algorithm it is necessary to restate Rule 1 as two seperate conditions:

- Condition 1
  For any given arbitrarily chosen starting node, there must exist a path to all other nodes in the chain. This follows directly from the fact that each node must be reachable from every other node in the chain.

- Condition 2
  For each node in the network a route must exist leading back to the original start node above. Once again, since each node must be reachable from every other node in the chain, the starting node must be reachable from all the other nodes.

Under condition 2 each node that can be reached from the starting node has a path leading back to the starting node, which in turn, under condition 1, has a path leading to every other node in the chain. Thus, each node in the network has a path leading to every other node, and the network therefore observes Rule 1.

4.2.2 Restating Rule 2

As with Rule 1, Rule 2 can also be divided into 2 parts:

- Condition 1
  Every node in the open chain must be reachable from the entry point.

- Condition 2
  For each node that can be reached from the entry point a route must exist to an exit point.

These conditions taken together are clearly equivalent to Rule 2.

4.2.3 Flooding

Before we can describe the routing validation algorithm used to test the above conditions, it is necessary to introduce the concept of *flooding*, which forms an integral part of the algorithm.

4.2.3.1 Forward-flooding

Forward-flooding involves visiting each and every node reachable *from* a given node according to the routing information provided for the particular chain. Each node visited can be tagged so that we know that the node can be reached from our original "start" node.

Forward-flooding can be implemented using recursion.

- If the current node is already tagged then do nothing. Otherwise,

  1. tag the current node, and
  2. for each other node that can be reached *from* the current node repeat the above steps by recursively calling the forward-flooding procedure passing the next node in the path as the new "current" node.

- If there are no paths leaving the node then we detect an error.

- If a path leads from this node to the external environment and the chain is a closed chain then detect an error (by Rule 3).

After forward-flooding the network, one can simply check the tags to see which nodes, if any, cannot be reached from the original start node (or entry point in the case of open chains). If all the nodes are tagged then we have satisfied the *first condition* needed to show that our network satisfies Rule 1 (or Rule 2 in the case of open chains).

Since each node is visited only once, forward-flooding is $O(n)$ where n is the number of nodes in the chain.

4.2.3.2 Reverse-flooding

Another version of the flooding algorithm described above is one which floods the network finding all nodes that can route *to* a given node. I will call this type of flooding 'reverse-flooding'.

- If the current node is already tagged then do nothing. Otherwise,
  1. tag the current node, and
  2. for each other node that has a path leading *to* the current node repeat the above steps by recursively calling the reverse-flooding procedure passing the other node as the new "current" node.
- If there are no paths leading to the node then we detect an error.
- If a path leads to this node from the external environment and the chain is a closed chain then detect an error (by Rule 3).

Reverse-flooding is $O(n)$ where n is the number of nodes in the chain, provided that routing information is stored in a form that allows us to know immediately which nodes route *to* the current node. Otherwise, at each node we would need to consider *all* other nodes to see whether they have a path leading to our current node, thereby making the algorithm $O(n^2)$!

4.2.4 The Route Validation Algorithm

The route validation algorithm involves one forward-flood and one reverse-flood to show that a given chain satisfies Rules 1 to 3. Since both forward-flooding and reverse-flooding (given the appropriate populated data structures) are $O(n)$ the algorithm is then itself also $O(n)$, where n is the number of nodes in the chain.

During forward-flooding temporary data structures are built up to record for each node, a list all other nodes which have routing information leading *to* that specific node. This data is needed to allow the reverse-flooding to be performed in $O(n)$ rather than $O(n^2)$.

The complete validation algorithm can be described as follows:

4.2.4.1 Open Chains

For each node accepting customers routing *from* the external environment

- Test Condition 1
  Forward-flood the network from each entry point and then check that each node in the network has been tagged. This means that all nodes in the network are reachable from the entry node.

- Test Condition 2
  Reverse-flood the network from each exit point and then check that each node in the network has been tagged. This means that each node in the network has a path leading to an exit point.

If no entry points or exit points exist we detect an error.

4.2.4.2 Closed Chains

Using any node in the chain as the start node,

- Test Condition 1
  Forward-flood the network from the chosen start node. Check that each node in the network has been tagged to ensure that all nodes can be reached from the chosen start node.

- Test Condition 2
  Reverse-flood the network from the chosen start node. Check that all nodes have been tagged to ensure that the start node can be reached from all other nodes in the network.

If no nodes have routing information we detect an error.

4.2.5 Implementation of the algorithm

The route validation algorithm was developed in conjunction with the routing data structures used in the CMTB. As can be seen from the above description of the algorithm, tagging requires one extra storage space for each node in the network.

Furthermore, to allow reverse-flooding we need to allocate extra temporary storage for each node in the network, listing all the nodes which have a path leading to the current node. This information is recorded in linked lists, and there is one linked list for each node in the network.

The algorithm has been divided into three main functions: `validateworkld()` which will validate a specific workload, `forwardflood()` to perform the forward-flood and `reverseflood()` to preform the reverse-flood. All three functions can be found in the module "book41.c". One other function, `validateroutes()`, can be used to check *all* workloads at once. This function merely calls `validateworkld()` for each workload in the model.

4.2.5.1 The function `validateworkld()`

Calls are made to the function `forwardflood()` and `reverseflood()` to flood the network from a specified start node (or entry point in the case of an open chain).

A call to the function `resettags()` resets the tags to FALSE for a specific workload, while a call to `checktags()` will return the node number of any node which has not been tagged, or -1 if all nodes are tagged.

4.2.5.2 The functions `fowardflood()` and `reverseflood()`

These function use recursion to flood the network starting at a specific node, and operate exactly as described in sections 4.2.3.1 and 4.2.3.2.

4.3 Solving relative throughputs

In Section 4.1 the concept of *relative throughputs* was introduced. The values ξ_{ir} must be solved before the model solution statistics can be found by the MVA algorithm.

Relative workload throughputs are solved in the module "through.c" by a call to the function `solve_thru_workload()`. Alternatively all workloads can be solved by a call to `solve_thru_each_workld()`. The throughputs are solved using either the Conjugate Gradient Method, or LU-Decomposition, both of which are described in section 4.5.

This section will describe how the linear equations are set up for solving these relative workload throughputs.

4.3.1 Introducing *nodes*

The reader is reminded of the equation defining the values ξ_{ir} – the relative workload throughputs of class r customers at the centre i.

This equation can also be represented as

$$\xi_u = p_{out;u} + \sum_{v=1}^{max} \xi_v p_{v;u} \tag{7}$$

where $\sum_{u \in E_k} \xi_u = 1$ for each E_k which is a closed chain. In the above equation we have replaced the subscript ir by a single variable u which represents a single class@centre combination. This is done so that the equation is more easily understood, and since this representation matches that used in the CMTB code more closely. Using this notation $p_{u;v}$ represents the probability that a customer at node u will route immediately to node v once being served. The value max is the total number of nodes in the network.

4.3.2 One workload at a time

While equation 7 can be used to solve the relative throughputs for all the chains in the model simultaneously, it is more economical if we restrict our attention to a single workload at a time. This is especially advantageous when routing information for only one workload has been changed, since it prevents us from re-solving the relative throughputs for all the chains in the model.

Even when solutions have to be found for *all* workloads, solving the equations one workload at a time can be quite significantly faster. Consider a network with J closed chains, and where each workload j has k_j nodes (class@centre combinations). If we solve the set of simultaneous equations for all workloads combined, the solution time is $O((k_1 + k_2 + k_3 + \ldots + k_J)^3)$. Solving the equations one workload at a time, on the other hand, is $O(k_1^3 + k_2^3 + k_3^3 + \ldots + k_J^3)$. This difference in the solution times is especially marked when J and each of the k_j are large.

4.3.2.1 Notation

When we restrict our attention to single workloads the notation for classes, centres and nodes becomes much more complicated, since the nodes in a workload (ie. all class @ centre combinations, where the class is a member of the workload in question) do not come from one continuous subset of the complete set of nodes in the model.

For convenience, the formulae and matrix representations used in the following sections will use node numbers running from 1 through to n. These numbers do not represent the *actual* node number in the model, but rather an enumeration over the subset of nodes that have routing information and which belong to the particular workload. We exclude all nodes in the workload for which no routing information exits, since these nodes will all have a relative throughput of zero. In this interpretation n represents the number of nodes in a single workload for which routing information exits, and not the total number of nodes in the model.

4.3.3 Open chains

In the case of open chains we note that there is no added constraint for equation 7. Instead the equation has a unique solution.

The equation can be translated into

$$p_{out;u} = \xi_u - \sum_{v=1}^{n} \xi_v p_{v;u}$$

which can be represented quite simply as the following matrix equation

$$\begin{pmatrix} (1-P_{1;1}) & -P_{2;1} & -P_{3;1} & \cdots & -P_{n;1} \\ -P_{1;2} & (1-P_{2;2}) & -P_{3;2} & \cdots & -P_{n;2} \\ -P_{1;3} & -P_{2;3} & (1-P_{3;3}) & \cdots & -P_{n;3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -P_{1;n} & -P_{2;n} & -P_{3;n} & \cdots & (1-P_{n;n}) \end{pmatrix} \begin{pmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \\ \vdots \\ \xi_n \end{pmatrix} = \begin{pmatrix} P_{out;1} \\ P_{out;2} \\ P_{out;3} \\ \vdots \\ P_{out;n} \end{pmatrix}$$

which we recognise as having the form $A \cdot x = b$. In this form this set of equations is easily solved using either of the algorithms described in Section 4.5.

4.3.4 Closed chains

In the case of closed chains the equation governing relative throughputs is subject to the constraint that all throughputs (for that workload) sum to one.

The equation and the constraint together translate into

$$p_{n;u} = \xi_u + \sum_{v=1}^{n-1} \xi_v (p_{n;u} - p_{v;u}) \quad \text{when } u < n$$

and

$$\xi_n = 1 - \sum_{u=1}^{n-1} \xi_u$$

which has a matrix representation of the form

$$\mathbf{A} \cdot \begin{pmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_{n-1} \\ \xi_n \end{pmatrix} = \begin{pmatrix} P_{n;1} \\ P_{n;2} \\ \vdots \\ P_{n;n-1} \\ 1 \end{pmatrix}$$

$$\text{where} \quad \mathbf{A} = \begin{pmatrix} (1+P_{n;1}-P_{1;1}) & (P_{n;1}-P_{2;1}) & \cdots & (P_{n;1}-P_{n-1;1}) & 0 \\ (P_{n;2}-P_{1;2}) & (1+P_{n;2}-P_{2;2}) & \cdots & (P_{n;2}-P_{n-1;2}) & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ (P_{n;n-1}-P_{1;n-1}) & (P_{n;n-1}-P_{2;n-1}) & \cdots & (1+P_{n;n-1}-P_{n-1;n-1}) & 0 \\ 1 & 1 & \cdots & 1 & 1 \end{pmatrix}$$

This set of equations can be solved easily using either of the algorithms described in Section 4.5.

4.4 Checking for infinite queues

When a model includes customers belonging to open chains, it becomes possible for a centre to have customers arriving faster than they can be served. In this case we would generate an infinite queue at the centre.

After solving the relative throughputs for all chains in the model, we check that no centre has an infinite queue. This is done by the function `checkmodelsoln()` (in "stat41.c") by a call to the function `verify_open_thruputs()` (in "book41.c").

This check can be accomplished by checking that ρ_i is less than one for each centre i that is not a DELAY centre.

We do, however, have a problem when the model includes PRIORITY servers. In this case the above test does *not* work. Rather, we must check that all the values W_{ir} solved in Step 6b of the MVA algorithm (fig 3) are non-negative. This check must be performed each time equation 17 is solved (see section 5.2.9).

4.5 Solving linear equations

The CMTB must be able to solve simultaneous linear equations for two reasons:

1. To solve the relative throughputs of customers at centres.

2. To allow the MVA algorithm to solve the model when PRIORITY servers have been introduced.

This section will present two algorithms that have been implemented in the CMTB for solving sets of simultaneous linear equations.

4.5.1 Using two solution algorithms

Solving the relative customer throughputs is typically a very time-consuming exercise — complicated models can often have workloads with routing matrices of the order of 100 by 100 elements. One might expect that we could use the same function to solve any set of linear equations. However, the CMTB should ideally take advantage of the fact that these large matrices are typically very sparse and therefore implement an algorithm that will solve such sets of sparse linear equations more efficiently.

In the CMTB one such 'sparse-matrix' algorithm has been included, which is based on the Conjugate Gradient method for solving non-linear equations. This method was originally implemented in the CMTB since it boasts a significant time advantage over other typical algorithms when applied to sparse matrices (see [Recipes 87])..

As stated in chapter 2, it was found that the method was only robust for smaller matrices, and therefore the implementation of the algorithm has not really offered much advantage. None the less, as mentioned earlier, the algorithm has been retained as an option as it is apparently possible to increase the accuracy of the solution obtained by manipulating the original matrix. This would require further research and development beyond the scope of this dissertation.

Obviously a sparse matrix algorithm cannot be used efficiently to solve linear equations for which the representative matrix is *not* sparse. We have already noted that the MVA algorithm must be able to solve such sets of linear equations. Consequently one other algorithm for solving sets of linear equations — The LU-Decomposition Algorithm — has been implemented in the CMTB. Although this algorithm cannot take advantage of any sparse matrix representations of the set of linear equations, it is very accurate and is therefore also the default for solving the relative throughputs.

4.5.2 The Conjugate Gradient Method

This section describes the Conjugate Gradient method for solving a set of linear equations. It also describes how this algorithm has been implemented in the CMTB to solve the relative throughputs for each workload.

A further discussion of the Conjugate Gradient Method can be found in [Recipes 87].

4.5.2.1 Basis of the algorithm

Let us assume that we have a matrix representation of a set of linear equations $A \cdot x = b$. Also, let us consider the function

$$f(x) = \frac{1}{2} |A \cdot x - b|^2 \qquad (8)$$

Clearly this function $f(x)$ has a single minimum, at a value x that satisfies the set of linear equations $A \cdot x = b$. The conjugate-gradient method can be used to minimise such a function, and thus solve the set of linear equations.

The Conjugate Gradient method is an iterative solution algorithm. We start with an initial estimate for our solution vector, and keep adjusting this estimate until we converge on the actual solution.

For the Conjugate Gradient method to work, we only need to be able to calculate $A \cdot y$ and $A^T \cdot y$ where A^T is the transpose of A and y is some arbitrary vector. If the matrix A is sparse then these calculations do not take the usual N^2 operations, but rather a smaller number of operations determined by the number of non-zero elements in A. If we can represent this matrix using a suitable sparse-matrix representation, then these calculations can be done relatively quickly.

4.5.2.2 Implementation of the Conjugate Gradient Method

The code for the Conjugate Gradient method is contained in the module "through.c".

This module has two special functions `asub()` and `atsub()` that will calculate the value of $A \cdot x$ and $A^T \cdot x$ respectively. A is the array with elements as defined in Section 4.3.2 above, and x is an arbitrary vector passed in the function parameter list.

We note that our `prob_list_root` data structure already represents an efficient sparse matrix representation system for our routing information. We can therefore use these routing information data structures directly in the functions `asub()` and `atsub()`.

The body of the Conjugate Gradient method is contained in the function `sparse()` which is responsible for controlling the iteration until the algorithm finally converges on the solution vector. This function will return the solution vector as well as the sum-of-square residuals of the estimated solution.

The code for the function `sparse()` was reproduced from [Recipes 87].

4.5.3 LU–Decomposition

This algorithm is a general purpose algorithm for solving sets of linear equations, and is quoted as the preferred method by many authors for solving such equations. The version of the algorithm that has been implemented in the CMTB is *Crout's Method with Partial Pivoting*, and was reproduced from [Recipes 87].

The code for the LU–Decomposition algorithm is contained in the module "ludcmp.c".

4.5.3.1 Basis of the LU-Decomposition algorithm

Suppose we are able to write the matrix A as a product of two matrices,

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A}$$

where L is *lower triangular* and U is *upper triangular*. Then we can solve the linear set of equations

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{x} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) = \mathbf{b}$$

by first solving for the vector y such that

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b} \tag{9}$$

and then solving

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y} \tag{10}$$

The advantage of breaking the linear set up into two successive ones is that the solution of a triangular set of equations is quite trivial, with equation 9 being solved by *forward substitution* and equation 10 being solved by *backward substitution*.

4.5.3.2 Limitations of the LU-Decomposition Algorithm

Unfortunately, the LU-Decomposition algorithm does not take advantage of the fact that the set of linear equations may be sparse. Instead, the equation matrix must be represented in its entirety. Furthermore, there is little chance of being able to modify the algorithm to take advantage of the sparseness of the matrix A since the algorithm modifies the matrix A, and it is unlikely that the new matrix would remain sparse.

4.5.3.3 Implementation of the algorithm

The functions `ludcmp()` and `lubksb()` have been reproduced from [Recipes 87]. To find the solution to a set of linear equations $A \cdot x = b$ we make the following calls

```
ludcmp(a, n, indx, d)
lubksb(a, n, indx, b)
```

The answer x will be returned in `b[]`. The original matrix `a[][]` will have been destroyed. The array `indx[]` is of dimension `n` (where `a[][]` is `n` $\times$ `n`) and holds the row permutation of `a[][]` after the LU-Decomposition. The value `d` is output as $+-1$ depending on whether the number of row interchanges was even or odd, respectively.

4.6 Solving sets of linear equations in the MVA algorithm

Simultaneous linear equations that need to be solved by the MVA algorithm are solved using LU-Decomposition which is described in Section 4.5. The MVA algorithm is responsible for setting up the matrices which represent the linear equations and passing these to the LU-Decomposition module.

Chapter 5

MVA and Statistical Measures

This is the last of three chapters in the dissertation concerning the CMTB. This chapter describes how the MVA solution algorithm has been implemented in the CMTB, and also gives a detailed account of the mechanics of the algorithm. Formulae defining the statistical measures available to the modeller, after the model has been solved, are given in the final section.

5.1 Introduction to the Mean Value Analysis algorithm

The MVA algorithm is the most complicated solution function in the CMTB.

Although the solution of the linear equations defining relative throughputs can sometimes take longer than the MVA solution itself, the problem of solving a set of linear equations is an old one, and one for which many well-known algorithms have been developed.

The MVA algorithm on the other hand is complicated. In fact, when priority centres are introduced the MVA algorithm must itself be able to repeatedly solve sets of linear equations. The MVA algorithm also requires a large amount of temporary storage for intermediate solution results, and an efficient means to allow mapping to and from this storage.

This chapter will describe the MVA algorithm in detail, and then go on to explain how the algorithm has been implemented in the CMTB.

5.2 The Algorithm

In its most basic form the MVA algorithm is used to solve:

- relative chain throughput rates $T_k^*(\vec{L})$,

- average waiting times $W_{ik}^{*}(\vec{L})$, and
- average total closed population queue lengths $Q_i(\vec{L})$

for all closed chains, and state independent FCFS, LCFSPR, PS and DELAY centres.

In the CMTB this algorithm has been extended to allow centres with a PRIORITY service discipline and PRIORITY classes. In this form the MVA algorithm will also solve the average waiting time W_{ir} for all classes r at priority centres i. Only two levels of priority are supported in the CMTB although this could easily be extended. The reader should note that only an approximate solution can be found where chains include PRIORITY service centres. However, by limiting the number of priority levels to only two, this solution is sufficiently accurate to be used confidently.

Throughout this section the theory regarding the algorithm and notes on the actual implementation of the algorithm will be mixed freely. This approach will hopefully allow readers to grasp more quickly how the MVA algorithm has been implemented in the CMTB. Readers who are interested in more technical texts may wish to study [Krit 82].

The following paragraphs will give an outline of the algorithm, and also describe in detail how each of the steps of the algorithm have been implemented in the CMTB.

5.2.1 Outline of the algorithm

The MVA algorithm is laid out in Figure 3. This algorithm is taken from Kritzinger [Krit 82] with amendments to Step 4 and 6 to allow the solution of models having PRIORITY service centres. This figure represents an outline of the algorithm as it has been implemented in the CMTB. The notation used is the same as that introduced in Section 4.1, except for the value W_{ir} referred to in steps 4 and 6 which is defined in Section 5.7.4 below.

The function `mva()` controls all of the steps 1 through 6, with different functions being called to carry out the calculations specific to each of the steps. The reader should note that the function `mva_solution()` is the 'top-level' function that should be called to initiate the solution algorithm. This function ensures that the correct storage is allocated for the MVA solution functions, and also checks to see whether the MVA results need to be interpolated for non-integral MPL levels (See Section 5.4).

5.2.2 Storing intermediate results at each population vector $\vec{i}$

The MVA algorithm requires certain statistics to be stored during each iteration of the outer loop (step 2). These stored values are needed in each of J later iterations that have the same population vector $\vec{i}$ but with one more customer in some chain k, $1 \leq k \leq J$, (ie. with population vector $(\vec{i} + \vec{e}_k)$).

Figure 3: The MVA algorithm for solving multiclass open and closed queueing networks.

STEP 1 : (Initialisation)
For each state independent FCFS, PS or LCFSPR centre i,
we set $Q_i(\vec{0}) = 0$.

STEP 2 : Main loop over all *closed* population vectors. Let $\vec{i} = (i_1, \ldots, i_J)$
FOR $i_1 = 0$ to $L_1; \ldots ; i_J = 0$ to L_J do; (Note i_J changes most rapidly)

STEP 3 : (Loop over all closed chains)
FOR $k = 1$ to J do;

STEP 4 : (Loop over all centres)
FOR $i = 1$ to N do;

$$W_{ik}^*(\vec{i}) = \begin{cases} \frac{\rho_{ik}}{\Xi_{ik}} \frac{1}{(1-\rho_i)} \left(Q_i(\vec{i} - \vec{e}_k) + 1\right) & \text{for state independent FCFS, PS or LCFSPR centre } i \\ \frac{\rho_{ik}}{\Xi_{ik}} & \text{for a DELAY centre } i \\ \sum_{r:(i,r)\in E_k} \left\{ \frac{\xi_{ir}}{\Xi_{ik}} W_{ir}(\vec{i}) \right\} & \text{for a PRIORITY centre } i \end{cases}$$

end of loop i

STEP 5 : (Compute the relative throughput of the closed chain k)

$$T_k^*(\vec{i}) = \frac{i_k}{\sum_{i=1}^{N} \left\{ \Xi_{ik} W_{ik}^*(\vec{i}) \right\}}$$

applying Little's Law to chain k

end of loop k (Step 3)

STEP 6 : (Loop over all centres)
FOR $i = 1$ to N do;
(Compute $Q_i(\vec{i})$ quantity required during next iteration of step 4.)

$$Q_i(\vec{i}) = \sum_{j=1}^{J} \Xi_{ij} T_j^*(\vec{i}) W_{ij}^*(\vec{i}) \quad \text{for each state independent FCFS, PS or LCFSPR centre } i$$

(not computed for DELAY service centres)

If i is a PRIORITY centre, we calculate the value W_{ir} for all classes r belonging to an open chain. See Step 6b – Section 5.2.9

end of loop i

end of loop $i_1, \ldots, i_J$

5.2.2.1 Calculating the *width* of the temporary stores

The number of temporary stores required for each of these statistics depends on the maximum *life* or *width* of such a store. We can determine this *width* by calculating the number of iterations over the loop $i_1, \ldots, i_J$ that must be made before the stored statistic (calculated in the present iteration $\vec{i}$) is no longer needed. Clearly, this will occur after we reach the population vector $(i_1 + 1), i_2, i_3, \ldots, i_J$ (assuming i_J changes most rapidly).

The width ψ of these temporary stores is therefore given by the equation

$$\psi = \prod_{k=2}^{J} (L_k + 1) \tag{11}$$

In the CMTB this value ψ is calculated by the function `Vectors_mem_req()`, and stored in the global variable `m_cust_vectors` in the module "mva41.c".

5.2.2.2 Minimising the *width*

The reader may note that when arranging the closed chains in the array `mod->workldarray[]` the function `put_wrklds_in_array()` will ensure that the chain with the highest MPL level will be assigned the lowest chain number. This will minimise the value ψ and can represent quite a significant space saving in the MVA algorithm. The maximum saving possible will be in the order of the ratio of the highest to the lowest closed chain MPL levels.

5.2.2.3 Mapping population vectors onto an array

We now need a function that will map the vector $\vec{i}$ onto an array of dimension `m_cust_vectors`. This mapping is provided by the function $\Psi(\vec{i})$ where

$$\begin{aligned}\Psi(\vec{i}) &= i_J + i_{J-1}(L_J + 1) + i_{J-2}(L_J + 1)(L_{J-1} + 1) \\ &\quad + \ldots + i_2(L_J + 1)(L_{J-1} + 1) \cdots (L_3 + 1)\end{aligned}$$

or

$$\Psi(\vec{i}) = \sum_{k=2}^{J} \left\{ i_k \prod_{j=k+1}^{J} (L_j + 1) \right\}$$

Using this notation $\prod_{J+1}^{J}(L_j + 1)$ is defined to be equal to 1. See Step 4 below for a discussion on how the function `index_map()` can be used to calculate $\Psi(\vec{i})$.

5.2.3 Step 1 — Initialisation

The values $Q_i(\vec{i})$ are stored by the array `m_Q_cust[][]`. The first index holds the centre i and the second $\Psi(\vec{i})$.

For DELAY centres we note that the value $Q_i(\vec{i})$ does not need to be calculated, and so the pointer `m_Q_cust[`i`]` is set to `NULL` where i is a delay centre. The reader should note that in **C** a two-dimensional array is represented as an array of pointers to one-dimensional arrays. Similarly, a three-dimensional array is merely an array of pointers to two-dimensional arrays. The technique of setting a pointer to `NULL` and *not* allocating storage space for a second or even third dimension, has been used repeatedly in the CMTB. This not only saves space but also allows some functions to see which values are zero without actually having to calculate them. This is done by checking for NULL pointers. The reader is referred to Section 5.5 for further notes on dynamic memory allocation.

The function `init_Q()` is used to initialise all the elements of `m_Q_cust[][]` to zero.

5.2.4 Step 2 — Looping over all closed population vectors

We cannot explicitly express the loop over all customer population vectors $(i_1, \ldots, i_J)$ using normal `for( ; ; )` statements, without restricting J to be some constant value. Clearly, this would be unacceptable.

The function `next_pop()` is used to simulate J nested `for` statements. When first called, this function will initialise the population vector to zero. On subsequent calls, the function will change the population vector and thereby mimic the action of several nested `for` statements. This function (taken from Nijenhuis [Niji 78]) was originally implemented by Renate Schmidt, and has been only slightly modified in the present implementation.

5.2.5 Step 3 — Looping over all closed chains

This step of the algorithm can be implemented quite trivially using a `for( ; ; )` statement, and is mentioned here only for completeness.

5.2.6 Step 4 — Computing average waiting times $W^*_{ik}(\vec{i})$

The real work of the MVA algorithm is done in steps 4 through 6. The Average Waiting time for customers in chain k at centre i, is calculated according to the formula set out in Step 4 of figure 3. The function `calc_W_chain()` implements this formula.

The values ρ_{ik}, Ξ_{ik} and ρ_i were introduced in Section 4.1, and are stored in the arrays `rho_chain[][]`, `xi_chain[][]` and `rho_centre[]` respectively. Each of these arrays is declared type `extern` in the module "mva41.c", and are originally defined in the module "param.c".

5.2.6.1 Introducing the function index_map()

We note that in the case of state independent FCFS, PS or LCFSPR centres the value $Q_i(\vec{i}-\vec{e}_k)$ is needed to solve $W^*_{ik}(\vec{i})$. This value has been stored in the array m_Q_cust[][] during a previous iteration of Step 6. However, we need a function that will map the population vector $(\vec{i}-\vec{e}_k)$ onto the appropriate array index. The function index_map() performs such a mapping, and allows the calling function to specify which chain k is to have one customer less than that of the present population vector $\vec{i}$. Thus to obtain the array index for population vector $(\vec{i}-\vec{e}_k)$ we pass index_map() the present population vector $\vec{i}$ and the chain number k. If we set the chain number k equal to NODEC (defined as -1) then the function index_map() will return the array index value for the population vector $\vec{i}$.

The reader should note that the function index_map() is not only needed in step 4 of the MVA algorithm, but also in steps 5 and 6b.

5.2.6.2 Calculating W_{ir} for PRIORITY centres i

For PRIORITY centres i we need the values W_{ir} for each class r in the chain k which visits that centre i. The values W_{ir} can be estimated according to the following equation

$$W_{ir}(\vec{i}) = \frac{\dfrac{1}{\mu_{ir}} + \sum\limits_{s:P(s)\leq P(r)} \left\{T_{is}(\vec{i}-\vec{e}_k)W_{is}(\vec{i}-\vec{e}_k)/\mu_{is}\right\}}{1 - \sum\limits_{s:P(s)<P(r)} T_{is}(\vec{i}-\vec{e}_k)/\mu_{is}} \tag{12}$$

where P(s) represents the priority level of class s customers, and P(s) < P(r) implies that class s customers have a *higher* priority than customers of class r.

As can be seen from the formula above the values W_{is} and T_{is} must be available from previous iterations of the loop $i_1, \ldots, i_J$. The values W_{is} can be calculated using equation 12 and stored using the same type of array structure used to store the values Q_i. The values T_{is} can be obtained by equations 21 and 20 for open and closed chains respectively. We note that using equation 20 would involve the value $T^*_k(\vec{i}-\vec{e}_k)$ and so the value T^*_k calculated in each iteration of Step 6 would also need to be stored.

The arrays m_W_cust[][][] and m_T_cust[][] have been used to store the values W_{is} and T_k respectively in each iteration of Step 2. The values W_{is} must be calculated and stored for *all* classes, from both open *and* closed chains, so that they may be used in equation 12 in following iterations of the loop $i_1, \ldots, i_J$. However, equation 12 can only be used to solve $W_{ir}(\vec{i})$ for classes r from *closed* chains. Therefore the values $W_{ir}(\vec{i})$ for classes r from *open* chains must be calculated by some other formula. Such a formula is given in Section 5.2.9 below.

5.2.6.3 Optimising the solution for PRIORITY centres i

Equation 12 must be evaluated many times in the sum represented in Step 4 of figure 3. Therefore the calculation has been optimised as effectively as possible through the use of the *static* variables `ddivide` and `top` in the function `calc_W_cust()` (The reader will remember that in C *static* variables do not lose their values between function calls). To see how this optimisation has been realised we write equation 12 as

$$W_{ir}(\vec{i}) = \frac{\frac{1}{\mu_{ir}} + \texttt{top}}{\texttt{ddivide}} \tag{13}$$

where

$$\texttt{top} = \sum_{s:P(s)\leq P(r)} \left\{ W_{is}(\vec{i} - \vec{e}_k) \times \texttt{d_t_is} \right\} \tag{14}$$

and

$$\texttt{ddivide} = 1 - \sum_{s:P(s)<P(r)} (\texttt{d_t_is}) \tag{15}$$

and the variable

$$\texttt{d_t_is} = T_{is}(\vec{i} - \vec{e}_k)/\mu_{is}$$

`ddivide` has an initial value of one, and `top` is initialised to zero. We can calculate `top` and `ddivide` simultaneously, noting that the value `d_t_is` (calculated for each class s in the sum used to calculate `top`) may also need to be subtracted from `ddivide` if P(s)<P(r), as shown in equation 15.

When we calculate the value $W^*_{ik}(\vec{i})$ using the sum presented in Step 4 of figure 3, equation 13 must be solved repeatedly for each class r belonging to chain k. However, given a priority centre i, then for each class r in a single chain k the values for both `top` and `ddivide` do not change, since all classes belonging to the same chain *must* have the same priority. Thus, by using the *static* variables `top` and `ddivide` we need calculate equations 14 and 15 only once in the sum represented in Step 4 of fig 3. The *static* variables `lastchain` and `lastcentre` are used to check whether `top` and `ddivide` must be recalculated when the function `calc_W_cust()` is called.

5.2.7 Step 5 — Computing the relative chain throughputs $T^*_k(\vec{i})$

The values $T^*_k(\vec{i})$ can be calculated quite trivially using the formula presented in Step 5 of Figure 3. This is done by the function `calc_T_chain()`.

We must remember that if there are PRIORITY centres in the model then the values $T^*_k(\vec{i})$ must be stored so that they may be used in Equation 12 in later iterations of the loop $i_1, \ldots, i_J$. These values are stored in the array `m_T_cust[][]`.

5.2.8 Step 6a — Computing $Q_i(\vec{i})$

The values $Q_i(\vec{i})$ are computed by the function `calc_Q_cust()` and then stored in the array `m_Q_cust[][]`. These values are not computed for DELAY centres.

5.2.9 Step 6b — Estimating W_{ir} for priority service centres i and customers of class r from open chains

As pointed out above, the values W_{ir} must be calculated for all classes r that belong to open chains. These values are stored in the array `m_W_cust[class][centre]`[$\Psi(\vec{i})$] for use in solving Equation 12 in later iterations over Step 4.

5.2.9.1 The mathematical basis

Sevcik and Mitrani [Sevcik 79] have shown that for $r : (i,r) \in E_k$, i a PRIORITY centre and E_k an open chain, equation 12 can be rewritten as

$$W_{ir}(\vec{i}) = \frac{\frac{1}{\mu_{ir}} + \sum_{s:P(s)=P(r)} \left\{ \frac{T_{is}(\vec{i})}{\mu_{is}} W_{is}(\vec{i}) \right\} + C_1(\vec{i})}{C_2(\vec{i})} \tag{16}$$

where

$$C_1(\vec{i}) = \sum_{s:P(s)<P(r)} T_{is}(\vec{i}) W_{is}(\vec{i}) / \mu_{is}$$

and

$$C_2(\vec{i}) = 1 - \sum_{s:P(s)<P(r)} T_{is}(\vec{i}) / \mu_{is}$$

It should be noted that equation 16 is only an approximation. If we solve equation 16 for classes in *descending* priority order, then the values $C_1(\vec{i})$ and $C_2(\vec{i})$ will always be known.

In practise, equation 16 can be more usefully written as the set of simultaneous equations:

$$C_2(\vec{i}) W_{ir}(\vec{i}) - \sum_{s:P(s)=P(r)} \frac{T_{is}(\vec{i})}{\mu_{is}} W_{is}(\vec{i}) = \frac{1}{\mu_{ir}} + C_1(\vec{i}) \tag{17}$$

This set of equations does *not* necessarily have a solution, since open class customers can arrive at a centre faster than they are being served, and therefore generate infinite queue lengths Q_{ir} and infinite waiting times W_{ir}. Since the introduction of PRIORITY classes will modify the service received by each class in ways that cannot be predetermined we are forced to check for infinite queues by checking that the solution vector contains no negative components each time equation 17 is solved.

5.2.9.2 Implementation in the CMTB

In the CMTB there are only two priority classes, and therefore we solve only two sets of simultaneous equations 17 — one for priority class customers ($C_1(\vec{i}) = 0$ and $C_2(\vec{i}) = 1$) and one for normal class customers. When solving the set of equations for normal class customers, the results from the previous solution involving only priority classes is available to calculate $C_1(\vec{i})$.

The set of linear equations 17 is solved by the function `calc_W_custs_openchains()`. This function will accept a *boolean* parameter `priors` which will dictate whether the set of equations must be solved for priority class customers (priors is TRUE) or normal class customers (priors is FALSE). The variables `priority_open_classes` and `normal_open_classes` store the number of priority classes and normal classes respectively that belong to open chains. These two numbers also represent the dimension of the set of linear equations used when solving 17 for priority and normal classes respectively.

The LU-Decomposition algorithm has been used to solve these sets of linear equations. This algorithm is discussed in Section 4.5. After each call to the LU-Decomposition algorithm, we check that the solution vector has no negative components. The reason for implementing such a check is to detect any infinite queues as described above.

It must be noted that when the matrix representation of 17 is set up, we need some mapping from each node $(i; r)$ to its allocated row in the matrix. In `calc_W_custs_openchains()` this mapping is not explicit but rather implicit. The nodes $(i; r)$ are allocated rows of this matrix in a sequential manner with the loops in the function extracting each of these nodes one by one. By using the exact same loops we can extract the results from the solution vector and insert them into the appropriate stores. The reader is referred to the code to see exactly how this implicit mapping works.

5.3 Storing the values W_{ir} for PRIORITY centres i and classes r belonging to open chains

In section 5.2.9 an equation was presented to approximate the values W_{ir}. These values were needed to calculate the relative chain waiting times W^*_{ik} at each PRIORITY centre i, in Step 4 of the MVA algorithm.

However, these calculated values W_{ir} have another important use. It will be shown in section 5.7.2 that the average queue length for class r customers belonging to an open chain E_k at a PRIORITY centre i, can not be calculated directly using the results from the MVA algorithm. Instead, we must apply Little's Law

$$Q_{ir}(\vec{L}) = W_{ir}(\vec{L}) \times T_{ir}(\vec{L})$$

to calculate the value $Q_{ir}(\vec{L})$.

Thus the values W_{ir} must be stored for all PRIORITY centres i and classes r belonging to open chains. The function `shutdown_mva()` copies these results into an array `W_open[][]`. This array is declared in the module "stat41.c" and allocated storage space in the function `check_priority_centres()` in the module "book41.c".

5.4 Interpolation of the MVA results

The MVA algorithm can only be used to solve models where each chain has an integral population level (MPL). We can, however, estimate a solution when one or more chains has a non-integral average population level by interpolation.

There are literally hundreds of interpolation algorithms that can be used to interpolate values over a multi-dimensional space. Many such methods are described in the literature [Conte 65], [Ralston 65] and [Recipes 87]. The algorithm that has been adopted in the CMTB is the same as that proposed in the earliest versions of MicroSnap.

The values that are interpolated include

- `m_Wstar_chain[centre][chain]`
- `m_Tstar_chain[chain]`
- `m_Q_centre[centre]`

These values represent the values W_{ik}^*, T_k^* and Q_i respectively. The final interpolated values are given as the solution statistics of the MVA algorithm.

5.4.1 The interpolation algorithm

Suppose we wish to calculate $R(\vec{i})$, where the population vector $\vec{i}$ has one or more non-integral components. Keeping with the adopted notation the vector $\vec{i}$ has dimension J, where J is the number of closed chains in the model.

We can calculate the values $R(\vec{x})$ for a set of m population vectors $\vec{x}$, where $\vec{x}$ has no non-integral components, and for each chain k, $i_k - 1 < x_k < i_k + 1$. The size of this set m is subject to the constraint $m \leq 2^J$. Clearly, the set of results $(R(\vec{x}_1), \ldots, R(\vec{x}_m))$ will surround the true solution $R(\vec{i})$ in a m-dimensional space, and therefore present a good basis for our interpolation.

Associated with each of these results $R(\vec{x}_k)$ is a weight, which will be used to estimate $R(\vec{i})$ as a weighted average of the values $R(\vec{x})$. This weight is given by the formula

$$w_{\vec{x}_k} = \left[\sum_{j=1}^{J} (i_j - x_{kj})^2 \right]^{-1/2} \tag{18}$$

thus giving more weight to those calculated values where the approximate (integral) parameters are closest to the original (non-integral) network parameters.

For simplicity, we denote $R(\vec{i})$ by R, $R(\vec{x}_k)$ by R_k, and $w_{\vec{x}_k}$ by w_k in the following equation

$$R = \frac{\sum_{k=1}^{m} R_k w_k}{\sum_{k=1}^{m} w_k} \qquad (19)$$

This equation gives the final interpolated result R (or $R(\vec{i})$).

As a refinement, we do not need to store the values R_k and the associated population vectors before applying equation 19, but can compute R by keeping a running total of the numerator and denominator shown in the above equation. Thus we merely capture the values R_k in each of the appropriate iterations of the loop $i_1, \ldots, i_J$. This algorithm can be summarised as

- initialise both w and R to zero
- for each relevant population vector $\vec{x}_k (1 \leq k \leq m)$
  1. compute R_k,
  2. compute the weight w_k using equation 18
  3. let $w = w + w_k$
  4. let $R = R + R_k$
- finally, $R = R/w$ is the interpolated result

5.4.2 Collecting the MVA results for interpolation

From the above description of the interpolation algorithm, we see that the values `m_Tstar_chain[]`, `m_Q_centre[]` and `m_Wstar_chain[][]` need to be collected for each of the population vectors $\vec{x}_k$.

The function `mva_interpol()` accepts these values in its parameter list, and uses the arrays `Tstar_chain[]`, `Q_centre[]` and `Wstar_chain[][]` to calculate the numerator of the equation 19. The global variable `sum` is used to sum the weights of the statistics used in the interpolation. These weights are calculated according to equation 18 in the function `weight()`.

The boolean `interpol` is used to tell the function `mva()` that a call to `mva_interpol()` is necessary after each iteration of step 6.

The final interpolated results are obtained in the function `mva_solution()` by dividing each of the values in `Tstar_chain[]`, `Q_centre[]` and `Wstar_chain[][]` by `sum`.

The reader should note that when the model includes PRIORITY servers and open chains, the interpolation process at this stage is still incomplete. The reason for this is discussed in Section 5.4.3.

5.4.3 Interpolating the values W_{ir} for PRIORITY centres i and classes r belonging to open chains

As pointed out in Section 5.2.9 the values W_{ir} (where i is a PRIORITY centre and class r belong to an open chain) need to solved approximately in Step 6b of the MVA algorithm using the set of linear equations 17. These results are then used directly by the function `ave_wait_time()` in the module "stat41.c" to give the average waiting time of such customers r at the PRIORITY centre i (See section 5.7.4).

5.4.3.1 The inconsistency after interpolation

However, if $\vec{i}$ possesses some non-integral components, then the final values W_{ir} that are calculated and stored in the array location `m_W_cust[r][i][m_cust_vectors-1]` do not correspond to the population vector $\vec{i}$ but rather to the vector $\vec{x}_m$ which is the vector $\vec{i}$ with every component rounded up to an integral MPL. Clearly, we must re-solve the set of equations 17 using the new *interpolated* values of T_{ir} and W_{ir}.

5.4.3.2 Re-solving

Before we can resolve equation 17 using the function `calc_W_custs_openchains()`, the interpolated results must be substituted for those in the array locations `m_T_cust[][m_cust_vectors−1]` and `m_W_cust[][][m_cust_vectors−1]`. Step 6b is then repeated in the function `solve_mva()`, and the subsequent call to `shutdown_mva()` will copy the relevant new 'interpolated' values of W_{ir} into the `W_open[][]` array.

5.5 Dynamic memory allocation in the MVA algorithm

The memory requirements of the MVA algorithm are complex. This is especially so when a model involving PRIORITY centres is solved. Consequently, memory management in the module "mva41.c" is complicated.

We will trace the steps in typical a call to the function `mva_solution()`.

5.5.1 Deleting old solution results

If the model has been solved previously, we deallocate the storage used by `mod->Q_centre[]`, `mod->Tstar_chain[]` and `mod->Wstar_chain[][]`. We cannot use this storage again, as

the number of closed chains J may have been changed by a MODIFY_WORKLOAD statement, thereby making the dimension of the above stores inappropriate.

We note also that even if the model has been solved previously, the stores `mod->Tstar_chain[]` and `mod->Wstar_chain[][]` have not necessarily been allocated any storage since we may have had no closed chains in the model. Thus we must check that the values `mod->Tstar` and `mod->Wstar` are not null before we can deallocate them.

5.5.2 Storage for interpolation

If the MVA results need to be interpolated, the arrays `Tstar_chain[]`, `Q_centre[]` and `Wstar_chain[][]`, are allocated space to keep a running total of the numerator in equation 19 for each of the solution statistics. This allocation is done in function `mva_solution()`. These arrays are never freed, but rather tagged onto the `model` data structure as the final interpolated results (after being divided by the normalising factor `sum`).

The array `avg_mpl_chain[]` is allocated space to store the non-integral average MPL levels of each chain. This vector is needed by the function `weight()` when calculating $w_{\vec{x}_k}$ defined by equation 18. This array `avg_mpl_chain[]` is allocated and freed in the function `mva_solution()`.

The array `int_pop_chain[]` is allocated space to store the final population vector needed in the MVA solution. This vector is the same as `avg_mpl_chain[]` with each component rounded up to an integral value. This array is both allocated and freed in the function `mva_solution()`.

5.5.3 The function startup mva()

This function is called by `mva()` before any of the steps described in figure 3 are carried out. `startup_mva()` initialises certain variables before the body of the MVA algorithm is entered. Also, space is allocated to store all the intermediate results of the MVA algorithm.

The arrays `m_Tstar_chain[]`, `m_Wstar_chain[][]`, `m_Q_centre[]` and `m_Q_cust[]`, which have been introduced in previous sections, are all allocated space in this function. The array `m_pop_chain[]` is also allocated, and is used to store the population vector $\vec{i}$. Of all these arrays, only `m_Q_centre[]` is definitely allocated space. The others will not be allocated if there are no closed chains in the model.

If the model includes PRIORITY centres, we must allocate more storage. These arrays include

- `m_W_cust[class][priority_centre]`$[\Psi(\vec{i})]$

  The third dimension of this store is not allocated if the value of W_{ir} is known to be zero for the particular class@centre combination. This will be the case when

1. `mod->xi_values[class][centre]` is zero, or
2. `mod->mu_class[class][centre]` is zero

- `m_T_cust[closed_chain]`$[\Psi(\vec{i})]$
  This stores the values T_k^* for each population vector $\vec{i}$.

- Arrays to store the matrix representation of the set of linear equations 17. These include
  1. the array `a[][]`
     used to represent the LHS of the set of linear equations
  2. the array `rhs[]`
     used to represent the RHS of the set of linear equations
  3. the array `indx[]`
     used to store the row permutations of the LU-Decomposition of `a[][]`

5.5.4 The function `shutdown mva()`

This function is concerned with releasing all the temporary storage used by the MVA algorithm.

All storage detailed in the above subsection is released, with one possible exception. When the MVA results do not need to be interpolated then the arrays `m_Tstar_chain[]`, `m_Wstar_chain[][]` and `m_Q_centre[]` all hold the final solution statistics. They are therefore *not* released, but rather tagged onto the `model` data structure.

5.6 The function `checkmodelsoln()` in the module "stat41.c"

Whenever a solution statistic is requested by a statement in the evaluation section of a SnapL program, the function `checkmodelsoln()` will be called by the appropriate statistic evaluation function in the module "stat41.c".

`checkmodelsoln()` is responsible for ensuring that the MVA solution statistics are up to date. A boolean `modifiedmva` is used to mark any alteration to the model definition, and if set TRUE the model must be re-solved.

5.6.1 The function `check priority centres()`

This function *must* be called before the MVA algorithm can be initiated. It will

- provide a mapping `pri_centres[]` from the model centre number to the 'priority centre number'. The 'priority centre number' is not a true nodal centre number, but

rather an enumeration over all priority centres. This enumeration is often used in loops over all PRIORITY centres. We have

`priorcentrenum = pri_centres[modelcentrenumber]`

This also allows arrays to be created with dimension `num_priority_centres` and then indexed by `priority_centre_number`

- provide a mapping `centres_pri[]` from the enumerated 'priority centre number' to the true model centre number. This is an inverse of `pri_centres[]` and gives

  `priorcentrenum = pri_centres[centres_pri[priorcentrenum]]`

- allocate suitable space for the array `W_open[][]`

5.7 Statistical measures

All the statistics that the user may wish to use can now be evaluated from the results obtained in the previous two sections.

In the CMTB these statistics are evaluated only when requested, and do not take up any additional temporary storage. They are calculated in the module "stat41.c".

When the model is modified (and when it is first defined), the solution functions are not called directly. This is because some modifications may take place before any statistics are requested, and so any time spent solving the model, before it is redefined, is wasted. Consequently each of the procedures in "stat41.c" calls a function `checkmodelsoln()` which ensures that the model solution statistics are up to date. This function was described in Section 5.6.

5.7.1 Notes on the statistical measures formulae

The formulae described in the next few subsections give performance measures for each class $1 \leq r \leq R$ in each chain $1 \leq k \leq K$ at every centre $1 \leq i \leq N$. In each of the formula the values T_k^*, W_{ik}^* and Q_i are all dependent on the state $\vec{L}$, where $\vec{L}$ is a vector which represents the integral number of customers in each of the closed chains.

To see how these formulae are implemented in the CMTB the reader may wish to note the use of the following variables:

`model->xi_values[r][i]`	stores the values ξ_{ir}
`model->Tstar_chain[k]`	stores the values T_k^*
`model->Wstar_chain[i][k]`	stores the values W_{ik}^*
`model->Q_centre[i]`	stores the values Q_i

In all the formulae E_k represents the chain to which the class r customers belong. Each of the formula below has been taken from Kritzinger [Krit 82].

5.7.2 Average queue length

The expected number of class r customers at centre i, when the network is in state $\vec{L}$, is given by

- Where E_k is a closed chain.

$$Q_{ir}(\vec{L}) = \frac{\gamma_{ir}}{\rho_{ik}} \Xi_{ik} T_k^*(\vec{L}) W_{ik}^*(\vec{L})$$

  T_k^* and W_{ik}^* are calculated in steps 5 and 4 of the MVA solution respectively.

- Where E_k is an open chain

  1. For r a PS, LCFSPR or state independent FCFS centre

  $$Q_{ir}(\vec{L}) = \frac{\gamma_{ir}}{(1-\rho_i)}(1 + Q_i(\vec{L}))$$

  2. For r a DELAY centre

  $$Q_{ir}(\vec{L}) = \gamma_{ir}$$

  3. For r a PRIORITY centre

  $$Q_{ir}(\vec{L}) = W_{ir}(\vec{L}) T_{ir}(\vec{L})$$

  from Little's law. Note that W_{ir} is calculated directly by the MVA solution algorithm.

This value is calculated by the function `ave_queue_length()` in "stat41.c".

5.7.3 Average throughput rate

The average throughput rate $T_{ir}(\vec{L})$ of customers of class r at centre i, when the network is in state $\vec{L}$, is given by

- Where E_k is a closed chain

$$T_{ir}(\vec{L}) = \xi_{ir} T_k^*(\vec{L}) \tag{20}$$

  where $T_k^*(\vec{L})$ is computed in step 5 of the MVA algorithm.

- Where E_k is an open chain

$$T_{ir}(\vec{L}) = \lambda_k \xi_{ir} \tag{21}$$

This value is calculated by the function `ave_thruput()` in "stat41.c".

5.7.4 Average waiting time

The expected waiting time $W_{ir}(\vec{L})$ of class r customers at centre i, when the network is in state $\vec{L}$, is given by

- Where E_k is a closed chain, or r is *not* a priority centre

$$W_{ir}(\vec{L}) = Q_{ir}(\vec{L})/T_{ir}(\vec{L})$$

  by Little's Law

- Where E_k is an open chain and r is a priority centre, the value W_{ir} is calculated explicitly in the MVA solution algorithm. These values must be stored.

This value is calculated by the function `ave_wait_time()` in "stat41.c".

5.7.5 Average queueing time

The expected queueing time of class r customers at centre i, when the network is in state $\vec{L}$, is by definition the waiting time W_{ir} less the service time $1/\mu_{ir}$. Thus

$$Qt_{ir}(\vec{L}) = W_{ir}(\vec{L}) - 1/\mu_{ir}$$

This value is calculated by the function `ave_queue_time()` in "stat41.c".

5.7.6 Average cycle time (Closed Chains)

The cycle time is defined as the time elapsed from the moment a customer of class r leaves a centre i until it returns to the same centre i as a customer of the same class, given that the network is in state $\vec{L}$.

$$C_{ir}(\vec{L}) = \frac{L_k - Q_{ir}(\vec{L})}{T_{ir}(\vec{L})}$$

This measure is defined only for customers from closed chains, and is calculated by the function `ave_cycle_time()` in "stat41.c".

5.7.7 Average turnaround or residence time (Open chains)

The average residence time is the time spent by class r customers in the open chain E_k when the network is in state $\vec{L}$. It is the total time inside the network for customers of that class, and is given by the expression

$$R_r(\vec{L}) = \frac{1}{\lambda_k}\left\{\sum_{i,r:(i,r)\in E_k} Q_{ir}(\vec{L})\right\}$$

The average residence time is not defined for classes belonging to closed chains, and is calculated by the function `ave_residence_time()` in "stat41.c".

5.7.8 Average utilisation

The utilisation $U_{ir}(\vec{L})$ of class r customers at centre i, when the network is in state $\vec{L}$, is given by the ratio of the average throughput rate to the capacity of server i. That is

$$U_{ir}(\vec{L}) = \begin{cases} \frac{T_{ir}(\vec{L})}{\mu_{ir}} & \text{for PS, LCFSPR, PRIORITY and state indep. FCFS centres} \\ \frac{Q_{ir}(\vec{L})}{Q_i(\vec{L})} & \text{for } E_k \text{ a closed chain and } i \text{ a DELAY centre} \\ \text{undefined} & \text{for } E_k \text{ an open chain and } i \text{ a DELAY centre} \end{cases}$$

This value is calculated by the function `ave_utilisation()` in "stat41.c".

Chapter 6

Designing the XSnap User Interface

A toolbox of solution routines (such as the CMTB) is obviously of little use unless it is integrated into some application which offers the facility to define, query and modify models that are be to solved by such routines. Of equal importance to the CMTB, therefore, is the User Interface which is offered under XSnap.

This chapter describes the GUI (Graphical User Interface) offered under XSnap, while the actual *implementation* of the interface is described in the following chapter.

Sample screens taken from XSnap are shown to complement the textual description of the interface's features.

It should be noted that since XSnap has been implemented using X-Windows, many of the colours, sizes, relative positions and actual text contained in each of the buttons or widgets inside the application can be easily changed using an *application-defaults* file. By editing this file, one can change much of the overall appearance of XSnap without resorting to coding in any way.

6.1 Requirements of the Interface

The primary purpose of developing XSnap was to produce an interactive modelling application that could be used to define, solve and manipulate queueing network models in a manner that was easy to master, practical and attractive.

The interface was to allow the modeller to define models of the greatest complexity with which the solution modules in the CMTB could actually cope, thereby using the CMTB to its full potential.

An important function of the interface is to help to ensure that the model definition is valid. Obviously, some degree of tolerance is needed whilst a model is still being defined

since part-completed models are almost always invalid. Certain constraints can be imposed by the interface, however. For example:

- the interface should not allow the modeller to define routing information or service requirements for a class of customer that has not yet been defined.
- customers in closed chains should not be allowed to route to or from the external environment.
- paths should not be allowed to route *from* exit points or *to* entry points, even for open chains.
- classes removed from the model should be automatically removed from all routing information and service requirement definitions, etc.

When a model is passed to the CMTB to be solved, the more sophisticated validation algorithms which form part of the CMTB will be used to check that the model is completely valid. The CMTB is able to report any errors which are then passed on to the modeller by XSnap. The purpose of the interface therefore is not to check that the model is valid, but rather to prevent any model that is clearly invalid from being defined.

When designing the interface, it was only natural to draw on experience gained whilst using other modelling packages with GUI's such as MACOM. Experience using MicroSnap was also valuable, and in the discussion below reference will often be made to both of these packages.

6.2 Constructing Models

When defining models to be solved in XSnap we need to consider:

- what graphical representation of the model would be best to use, if any.
- how workloads in the model and the different customer classes in each workload should be defined.
- how centres in the model, their service discipline and service time requirements for each class of customer should be defined.
- how routing information, detailing the routing of customers between centres in the model, should be defined.

6.2.1 The Canvas

Obviously, the most effective method for representing a model of queueing networks is to use a diagram. Diagrams can be used to show clearly all the centres in the model and the paths that customers take when routing between the centres. Both MACOM and XWAN build up diagrammatic representations of the model.

MicroSnap, on the other hand, uses a textual representation of the model which is defined using a specification language called SnapL. In MicroSnap, one can only establish the actual make-up of the model by scrutinizing the statements defining workloads, centres and routing information.

XSnap, like MACOM, allows the modeller to build up a picture of the model by adding centres and paths to a *canvas*. These centres and paths can be positioned or drawn using a pointing device such as a mouse. User defined names, rather than simple centre numbers, are shown beneath each centre in the model to add clarity.

A sample screen of the XSnap application is given in figure 4 showing the large central canvas and the centres and paths defined in a sample model.

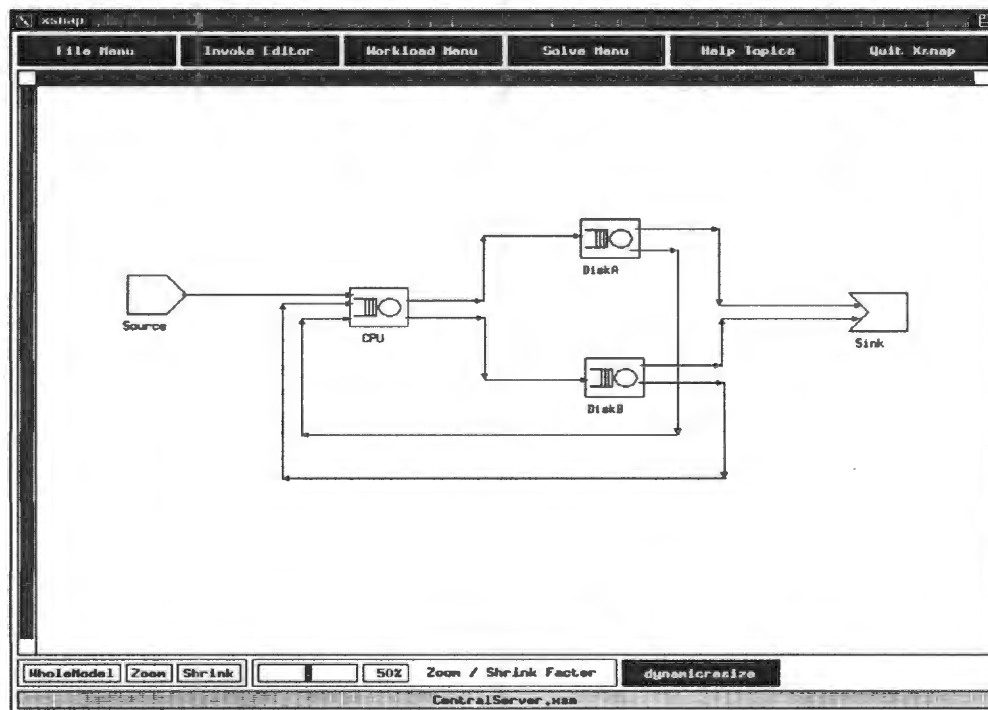


Figure 4: The *XSnap* screen with sample model

Once drawn, one can move centres or re-route (redraw) paths without losing any information about the model. This allows a modeller to rearrange the model so that any new centres or paths can be added without the model becoming cluttered or starting to appear unattractive or confusing.

When the canvas is too large to view on the screen the modeller is able to scroll the canvas appropriately, thereby showing only a part of the model through a viewport.

One of the difficulties with MACOM is that large models tend to become cumbersome and difficult to discern on the screen. It was decided therefore that in XSnap the canvas on which the model is drawn should not only scroll, but should also be allowed to grow or shrink so as to accomodate the model diagram comfortably. The modeler can also "zoom in" to certain areas of the model diagram, or view the model in its entirity. Zooming can be effected by merely clicking on a button which zooms in towards the centre of that part of the model currently viewable through the viewport, or by positioning a box around that part of the model which one would like to enlarge to fill the viewport. An additional button can be used to instruct XSnap to automatically resize and reposition the canvas so that the entire model can be viewed comfortably inside the viewport. An exmple of a 'zoomed-in' model is given in figure 5.

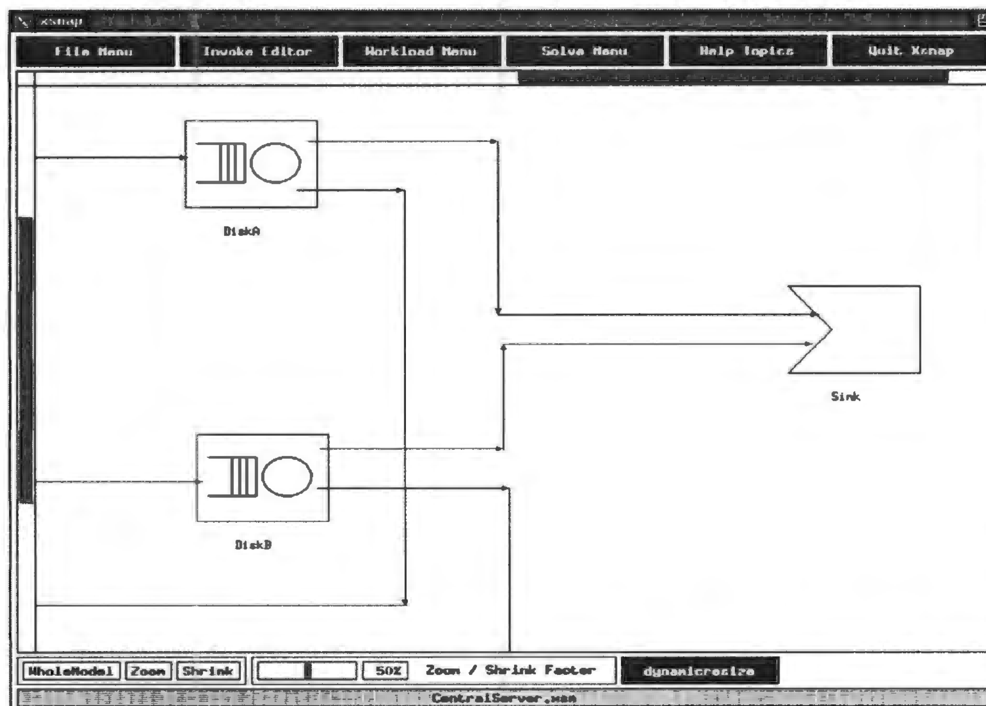


Figure 5: A 'zoomed-in' model

Attributes of the different paths or centres are set easily using the pop-up windows described in the following sections. Rather than activating these windows from a menu bar, these windows are activated by simply clicking on the path or centre in question on the canvas. This method helps to preserve the strong correlation between the model diagram and the rest of the details needed to make up the total model specification.

An attractive feature of MACOM was its sensible use of colour and different centre

icons to help add more detail and clarity to the model diagram. In XSnap this is done similarly using colours to distinguish between paths holding customers belonging to different chains, and by using different centre icons to depict the various centre service disciplines.

Sophisticated canvas management is one of the key features of XSnap. By allowing the resizing, scrolling and zooming of the canvas, repositioning of centres and the rerouting of paths XSnap has been able to offer a truly attractive alternative to the textual specification of models used in MicroSnap and other packages whilst also overcoming some of the faults of other GUI's such as MACOM.

6.2.2 Default Settings

Another problem with MACOM was that one could not copy parts of the model or specify default settings. XSnap allows the modeller to make the particular settings for any specific workload or centre default settings. Any new centres or workloads defined would then automatically have these settings. Furthermore, one can copy the default settings onto an existing centre or workload. This can therefore be used as a copy facility.

6.2.3 Variables and Expressions

XSnap allows the modeller to define variables and to use these variables when defining routing probabilities, MPL levels, arrival rates or centre service requirements. These variables can be allocated values using complex expressions such as those supported in MicroSnap. One can also use the facility to modify values across the whole model by simply changing the value of a single variable used in definitions throughout the model.

Variables are defined using the "Variables" option under the "Solve" menu. The latter is invoked from the main menu bar. This option pops up an editor in which the modeller simply lists the variables to be defined seperated by commas, and any assignment statements assigning values to the variables.

6.2.4 Workloads

Central to the definition of models is the definition of the different workloads and classes of customers that are allowed to travel between the centres in the model. These workloads are defined or modified using the Workload Manager. This pop-up window is invoked from the main menu bar of XSnap. A sample screen of the Workload Manager is given in figure 6.

No changes are actually made to the model until the modeller selects the "update" button.

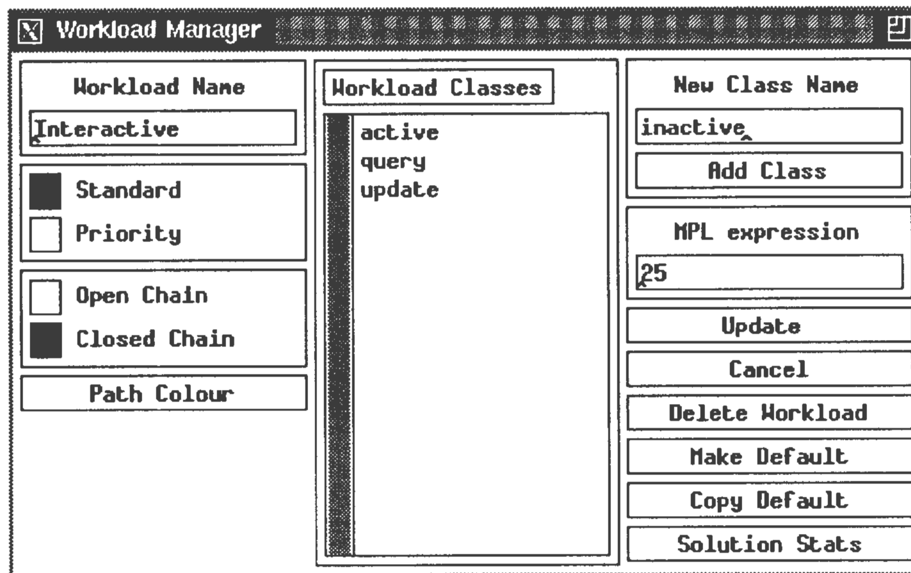


Figure 6: The Workload Manager

The Workload Manager allows the modeller to define or modify the following attributes of the workload:

- **The workload name.**
  If the name of the workload is changed then all references to classes belonging to the workload used in centre service requirement definitions are automatically ammended to reflect the new workload name.

- **Open or Closed.**
  The modeller can choose whether the workload should be open or closed by simply clicking on the appropriate toggle button. If a workload is changed to a closed chain then the routing information for the model is automatically checked to ensure that any routes to or from the external environment by classes in the chain are removed from the model.

- **List of Classes.**
  This list shows the defined classes in the workload. If the list is too long to fit in the viewport it can be scrolled. Classes can be removed from the list by simply clicking on the class to be removed. If a class list is modified, the routing information and centre service requirement information for the model is checked to ensure that any classes which are no longer defined are removed from the model.

- **New Class Name.**
  The class name dialogue allows the modeller to enter class names for the different

classes of customer belonging to the workload. The new class is added to the list by clicking on the "Done" button.

- **Workload Colour.**
  Each workload is given a colour to distinguish its paths from paths of other workloads. The "Workload Colour" button causes a list of all unallocated colours to be displayed and a new colour is chosen by clicking on the new colour name.

- **MPL expression or ARRIVAL rate.**
  This dialogue allows the modeller to enter the MPL expression for closed chains, or the arrival rate for customers belonging to open chains. The expression can include variables and normal math operators.

- **Copy Default.**
  This button allows the modeller to copy the default settings onto the workload.

- **Make Default.**
  This button allows the modeller to make the current settings for the workload the default settings.

6.2.5 Centres

6.2.5.1 Adding new centres

Centres are added to the model by simultaneously holding down the SHIFT button on the keyboard and pressing the middle mouse button. The new centre is added to the canvas at the position of the mouse pointer, and is given the default centre attributes.

6.2.5.2 Changing centre details

The attributes of each centre can be changed or queried using the Centre Manager. This pop-up window is activated by clicking on the centre, or when a centre is first added to the model.

The Centre Manager allows the modeller to set the service discipline of the centre in question and the service requirements of the classes that may visit the centre. Other attributes such as the centre name may also be changed. A sample screen of the Centre Manager is given in figure 7.

The Centre Manager maintains three lists of classes:

- classes serviced by the centre, together with their service requirement.

- classes not served by the centre and which *do not* visit the centre.

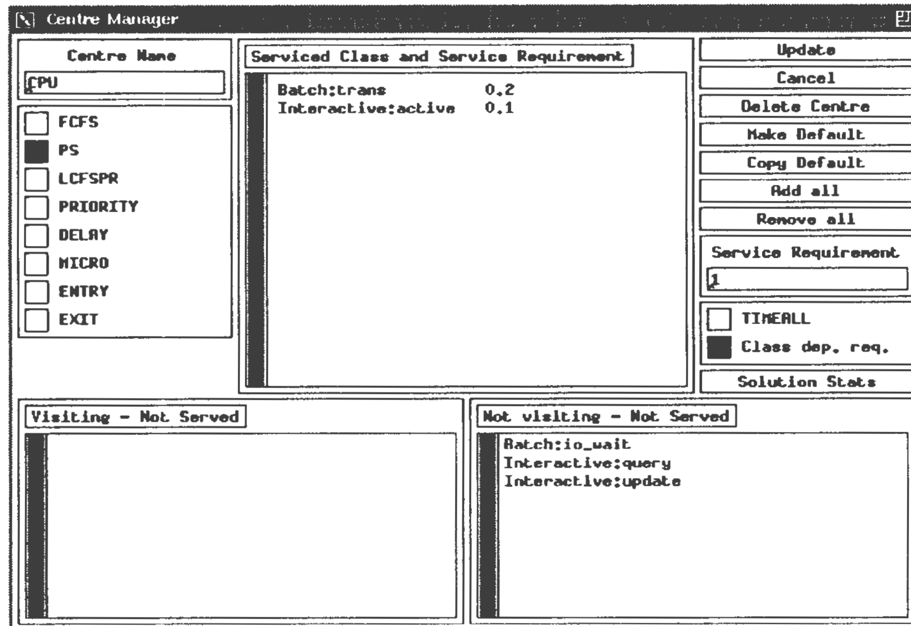


Figure 7: The Centre Manager

- classes not served by the centre even though they *do* visit the centre.

If the modeller clicks on any class name in either of the latter two lists, the class will be added to the list of classes demanding service from the centre. The service requirement for the new class is that entered in the dialogue named "Service Requirement".

By dividing those classes not serviced into those visiting the centre and those that do not visit the centre, the modeller can easily see if an error has been made. Normally all classes visiting a centre require service from the centre. Otherwise, they do not queue at the centre but rather route directly to the next centre.

The modeller can change the service requirement for any class by typing the new requirement into the "Service Requirement" dialogue and then simply clicking on the old service requirement listed to the right of the class name in the list of serviced classes. Classes are removed from the list of serviced classes by clicking on the class name to be removed.

This method of adding serviced classes ensures that only classes defined in the model can be used, and prevents unnecessary typing (and typing errors) on the part of the modeller.

Buttons also exist to allow the modeller to add all the unserviced classes to the list of serviced classes or alternatively to remove all the serviced classes.

A toggle is used to determine whether or not all classes require the same service requirement (TIMEALL), or whether the service requirement is class dependent. If the TIMEALL toggle is selected then the three class lists described above are automatically

inactivated, and all classes visiting the centre have a service requirement equal to that stated in the "Service Requirement" dialogue. FCFS centres are always TIMEALL centres.

The centre discipline is set using a toggle group, and a centre can also be defined as an entry or exit point using this toggle group. No service requirements can be defined for entry or exit points. If a centre is changed to an ENTRY or EXIT point then suitable warnings are given that the routing information may be modified automatically by the application to prevent routing errors.

The Make Default and Copy Default buttons operate in the same way as those for the Workload Manager.

6.2.5.3 Removing centres

Centres can be removed from the model by choosing the "Delete Centre" button in the Centre Manager. The user is prompted to confirm the action to prevent centres from being deleted by mistake.

When a centre is removed, all paths to and from the centre are also automatically removed from the model.

6.2.5.4 Moving centres

Centres can be moved by clicking on the centre in question using the middle mouse button. An outline of the centre then appears on the canvas and automatically moves around the screen as the user moves the pointer. The centre is finally re-affixed to the canvas when the user clicks any one of the pointer buttons.

Any paths leading to or from the centre are redrawn automatically by XSnap. If the user is unhappy with the route of the new paths then these can be redrawn manually as described in the next section. At present only a simple graph drawing algorithm is used to map out the new route to the repositioned centre. A more sophisticated algorithm could possibly be introduced later into the package to optimise not only these redrawn routes but also all other paths in the model.

6.2.6 Paths

In any model, the routing information is defined through the use of *paths.* Paths are used to carry customers belonging to any one chain from a source centre to a destination centre.

Customers leaving any one centre can route to a number of different destination centres according to different *routing probabilities.* To calculate these routing probabilities, XSnap compares the specified *relative frequencies* defined by the modeller for each class routing along each path leaving the source centre. For example, if *class a* at *centre 1* routes to

both *centre 2* and *centre 3* , then *class a* will need to be included in the routing information along the paths from centre 1 to both centres 2 and 3. If *class a* is to route to each of these centres with equal probability, then the relative frequency for this class along each of these paths will have to be identical. Conversely, if the relative frequency for this class along the path to *centre 2* was three times that along the path to *centre 3* then the probability of routing to centres 2 and 3 would be 75% and 25% respectively.

6.2.6.1 Adding new Paths

Paths can be added to the model by simply clicking on the centre from which the path is to begin using the rightmost mouse button. A pop-up then invites the user to select the workload to which the path belongs. Only classes belonging to the chosen workload will be able to route along the path.

Once the user has selected the workload, the path is then drawn using a rubber-band technique. Under this method each segment of the path is drawn in turn. The first segment is pinned at its one end to the centre from which the path leaves. The other end of the segment automatically changes position as the user moves the pointer, until this end of the segment is also pinned to the canvas by clicking one of the mouse buttons. A new segment is then automatically started with its first end pinned to the canvas at the same position as the end of the previous segment. The path is completed when the last segment drawn leads to a centre.

XSnap automatically ensures that all segments drawn are either horizontal or vertical. This makes the model neater and more attractive.

6.2.6.2 Changing Path Details

The user can modify the routing information for each path by invoking the Path Manager. The Path Manager is invoked by clicking on the path in question on the canvas. This pop-up window allows the user to specify which classes of the workload route along the path and with what relative frequency. A sample screen of the Path Manager is given in figure 8.

The Path Manager shows a list of all classes that *visit* the centre from which the path originates, as well as a seperate list holding all other classes in the chain.

The table of routing information is made up of triples. These triples show the class of the customer at the source centre, the class of the customer when it reaches the destination centre and the relative frequency of customers routing along this path. For example, for a path leading from *centre1* to *centre2*, the triple (*class1, class2, freq*) means that customers of *class1* at *centre1* will, with a relative frequency of *freq*, route to *centre2* where they will then be of *class2*.

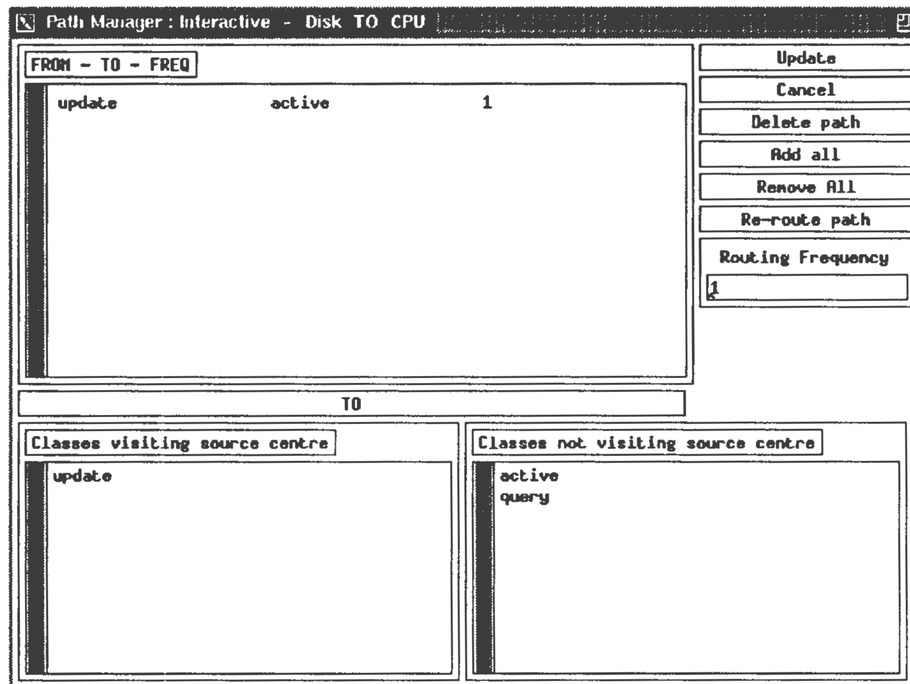


Figure 8: The Path Manager

The routing triples can be built up easily by simply clicking on the class names *class1* and *class2* in turn. The class names *class1* and *class2* will be found in either the list of classes visiting or the list of classes not visting *centre1*. The *freq* making up the final component of the triple is simply taken from the dialogue headed "Routing Frequency".

Routing triples can be removed from the path by clicking on either the first or second class name in the triple. The relative frequency can be modified by typing the new frequency into the "Routing Frequency" dialogue and then simply clicking on the relative frequency in the triple to be modified.

Although the above method may appear complicated at first, it is in fact extremely simple and efficient. There is no typing of class names or changing from "add" to "delete" modes. The lists of classes visiting and not visiting the source centre are also useful since the user can quickly see whether a class which visits a centre ever leaves the centre.

6.2.6.3 Deleting paths

Paths can be deleted by clicking on the "Delete Path" button in the Path Manager. The user will be prompted to confirm the action before the path is removed from the model.

6.2.6.4 Re-routeing paths

Paths can be redrawn from a source centre to a (possibly different) destination centre, without losing any path details, by selecting the "Re-route Path" button in the Path Manager.

The path is redrawn using the rubber-band technique described above. Redrawn paths are allowed to route to different destinations since, on occasions, the modeller may wish to insert a new centre inbetween two existing centres in a path.

6.3 Saving and Retrieving Models

Models can be saved and retrieved using model names defined by the user. The "File" button on the main menu bar activates a pop-up menu asking whether a model is to be saved or retrieved. A dialogue is then used to get the model name. All saved models have the extension '.xsn'.

6.4 Textual Representation of the Model

Sometimes a model definition can contain bugs that are difficult to find. In these cases one would want to be able to generate a textual representation of the model that can then be scrutinized later.

Such a facility can also be used to keep hard-copy definitions of models in case the electronically saved model ever becomes corrupted, overwritten or lost. The details contained in the textual version of the model can then be used to reconstruct the model using the XSnap GUI.

A textual representation of the model can be obtained by selecting the "XSnap Text" option under the "FILE" menu. The user will be prompted for the file name and the model details will then be written to the file.

When a model definition is written in textual form to a file, and the solution statistics have already been found for the model, then these statistics are automatically appended to the end of the model definition.

The CMTB also provides functions for producing a hard-copy of the model definition. The output generated by these routines is slightly different in format to that described above, which is generated by XSnap. This output can be sent to a file using the "CMTB Text" option under the "FILE" menu. The ouptut is only valid, however, where the XSnap model data has been copied successfully (i.e. without any errors being reported) to the CMTB. Otherwise the CMTB model may be incomplete and not fully reflect the model defined in the XSnap GUI. An example of the output generated is given in appendix B.

6.5 Solution Statistics

6.5.1 Generating Solutions

The model can be solved by selecting the "Go" option from the menu invoked when the mouse is clicked on the "Solve" button on the main menu bar. The model is then copied to the CMTB and solved. Any errors reported by the CMTB are displayed in pop-up windows.

A pop-up window displays a suitable message if the model is solved successfully, and also reports the solution time of the CMTB solution algorithms.

6.5.2 Viewing Solution Statistics

Solution statistics can be viewed by selecting the appropriate button in either the Workload Manager or the Centre Manager.

When viewing solution statistics from within the Workload Manager, a table is shown giving all the computed statistics for each class in the workload at each centre in the model. An example of the statistics shown is given in figure 9.

Workload Solution Details

Solution Statistics for Centre: CPU

Class Name	QUEUE	THRU	WAIT	CYCLE	QTIME	UTIL
active	0.007367	0.07278	0.1012	343.4	0.001227	0.007278
query	0	0	0	0	0	0
update	0	0	0	0	0	0

Solution Statistics for Centre: Disk

Class Name	QUEUE	THRU	WAIT	CYCLE	QTIME	UTIL
active	0	0	0	0	0	0
query	0	0	0	0	0	0
update	10.8	0.02547	424	557.4	404	0.5095

Solution Statistics for Centre: terminal

Class Name	QUEUE	THRU	WAIT	CYCLE	QTIME	UTIL
active	0	0	0	0	0	0
query	14.19	0.04731	300	228.5	0	1
update	0	0	0	0	0	0

Residence Times

Class Name	RES

Finished

Figure 9: Sample workload solution statistics

The table of solution statistics shown from within the Centre Manager gives the computed statistics for all classes in the model at the specific centre in question. An example of the statistics shown by the Centre Manager is given in figure 10.

If the model is modified then neither the Workload Manager nor the Centre Manager will allow the modeller to view the statistics calculated when the model was previously

Centre Solution Statistics

Solution Statistics for Centre: CPU

Class Name	QUEUE	THRU	WAIT	CYCLE	QTIME	UTIL
Batch:io_wait	0	0	0	0	0	0
Batch:trans	0.004966	0.02453	0.2025	407.5	0.00248	0.004905
Interactive:active	0.007367	0.07278	0.1012	343.4	0.001227	0.007278
Interactive:query	0	0	0	0	0	0
Interactive:update	0	0	0	0	0	0

Finished

Figure 10: Sample centre solution statistics

solved. This is to prevent the modeller from assuming that the statistics shown relate to the new modified model. Rather, the modeller is prompted to re-solve the modified model before the solution statistics can be viewed.

6.5.3 Saving Solution Statistics

One can save the solution statistics by selecting the "Write Solution Stats" option under the "FILE" menu.

Alternatively, solution statistics can be written to a file along with the model definition by chosing the "XSnap Text" option under the "FILE" menu.

6.6 Repetitive Model Evaluation

When modelling queueing networks, one often needs to solve a model a number of times whilst changing one or more of the model parameters each time the model is solved. Ideally, the modeller should be able to specify exactly which paramaters are to be changed, and request that the solution results are tabulated showing the effect of the changing model parameters. XSnap allows exactly such a facility by way of an Evaluation-Section which accompanies each model definition.

The Evaluation-Section can be used to request solution output in the form of tables, detailed reports, or even through the use of print statements. High level programming constructs such as loops, condition statements and complex mathematical and boolean

expressions allow the modeller to generate and manipulate solution statistics. The evaluation section code is written using a language called SnapL, which is the specification language used in MicroSnap to define and solve models. In XSnap, the model is obviously defined using the GUI, and there is therefore no need to write a SnapL *definition* of the model. None the less, a SnapL *evaluation-section* can be used to automatically modify and resolve the model, and to produce customized output.

It might be said that the fact that modellers may often find themselves involved in *textual* interaction with XSnap defeats the whole object of having a GUI in the first place. This, however, is totally untrue. Rather, the facility to automatically modify and resolve the model and to collect and manipulate the solution statistics adds tremendous power to the modelling package. Such power cannot be gained by merely clicking on buttons, and can only be truly harnessed through the use of something more dynamic such as SnapL. A GUI *is* useful in that it overcomes the traditional downfall of the wholly-textual interface which, of course, is that with textual interfaces the basic structure of the network being modelled is not at all obvious or clear. The GUI can also be used to help the modeller define the model, ensuring that no basic errors are made. Quite arguably, XSnap offers the best of both types of package.

6.6.1 Background to the SnapL language

The Stochastic Network Analysis Programming Language, or SnapL, was originally developed by M. Booyens and P.S. Kritzinger whilst working at the Institute for Applied Computer Science at the University of Stellenbosch in 1983. A paper describing the language was printed in the international journal Performance Evaluation [Krit 84] in August 1984.

The language was designed to allow a modeler to solve multi-class queueing networks and to manipulate the results of such a solution into a useful representable form.

Readers who are interested in the history of the SnapL language are referred to [Krit 84].

6.6.2 The definition and evaluation sections

A SnapL program used in MicroSnap has a definition section and an evaluation section. The former is responsible for defining the model and generates no output. The latter is responsible for generating tables, reports or any other output that is required. The evaluation section also supports loops, condition statements and expressions involving model solution statistics.

The evaluation section can also be used to modify the defined model. Any modifications

affect only the CMTB representation of the model and the model defined using the XSnap GUI remains unaffected. Obviously, under XSnap, the definition section is not needed. Rather, the definition of the model is accomplished using the Workload Manager, Centre Manager and Path Manager described above.

6.6.3 Editing Evaluation Code

Evaluation code can be edited using the pop-up editor invoked by clicking on the "Evaluation Part" button on the main menu bar.

This evaluation code is automatically appended to the end of the textual model definition produced when a user selects "XSnap Text" under the "File" menu.

The Backus Naur Form of the SnapL language is given in the following chapter regarding the implementation of XSnap, and a more detailed description of the features of the language can be obtained from the MicroSnap User Manual [MSnap 90].

6.6.4 Parsing Evaluation Code

Once an evaluation section has been edited, it can be parsed by XSnap by selecting the "Evaluation Part" option under the "Solve" menu. If the evaluation section is parsed successfully, the output produced is automatically displayed in a pop-up window. An example of an evaluation section together with the output generated is given in appendix A. If the evaluation section is not parsed successfully then a pop-up window is displayed showing any errors encountered by the parser.

The output is automatically written to a file having the same name as that of the model, but with the extension ".evl". The trace of the parse (together with any error messages) is also automatically written to a file having the same name as that of the model, but with the extension ".lst".

It should be noted that the model can be solved *without* parsing the evaluation section by selecting the "Go" option under the "Solve" menu.

Chapter 7

Implementing XSnap

This chapter describes the implementation of XSnap, as well as the hardware and software environment in which XSnap was developed.

7.1 The Development Environment

The original development of the CMTB was done on an IBM PC/AT running under DOS, whilst XSnap has been developed on a Sun SPARC 1+ workstation using the Sun Operating System version 4.1 (SunOS 4.1).

XSnap runs under *X-Windows (X11R4)* in a UNIX environment. The package was written using **C** and the Athena widget set supplied with X-Windows.

7.1.1 Choosing C as a programming langauge

The original development of the CMTB was done under DOS using Borland's TURBO C V2.0. Consequently, porting the CMTB (and MicroSnap) to a UNIX environment was a relatively pain-free excercise. When developing XSnap, **C** was a natural choice as a programming language for a number of reasons:

- There is a wide availability of **C** compilers, and this leads to the added advantage of wide portability.
- **C** offers great flexibility and a wide array of library functions that can be used by the programmer.
- There is an X-Windows system interface for **C**.
- The CMTB had already been successfully implemented in **C**.

Although there has been a lot of excitement about **C++** in recent years, a **C++** compiler was not available under UNIX or even under DOS at the start of 1990 when

the CMTB was first implemented. In any event, **C++** compilers are not yet as readily available as those for the vanilla-flavoured **C** and this is a significant disadvantage when it comes to portability.

Admittedly, LISP and ADA interfaces to the X-Windows system do exist. However, apart from the author's own inexperience with using these languages, both languages suffer from a lack of compiler availability and conformity.

7.1.2 X-Windows

The X-Windows system has been developed jointly by DEC and MIT and uses a device-independent architecture to enable the development of *portable* graphical-interface based applications.

The X-Window system is based on a client-server model, which allows programs to be run on one machine whilst displaying on another. X-Window applications are *event-driven* and the server communicates with the application by means of these events.

Figure 11 shows the architecture of the X-Windows programming environment. When

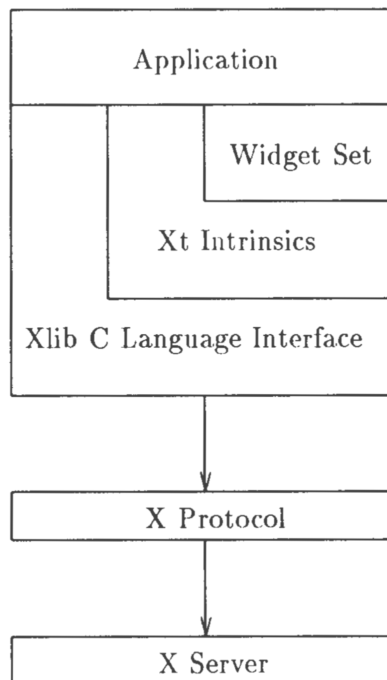


Figure 11: The X-Windows Architecture

developing applications in **C**, one uses the XLib C Language Interface to handle the lowest level interaction between the application and the X-Windows system. Higher level toolkits have been designed to be layered on top of XLib, the most standard of which used

by application programmers is the X Toolkit (*Xt* for short). Other toolkits do exist and include the Stanford *InterViews* toolkit, the Texas Instruments *CLUE* toolkit, and the Carnegie Melon *Andrew* toolkit.

The X Toolkit provides an object-orientated layer supporting the user-interface abstraction known as a *widget*. Examples of widgets include simple buttons or labels on the screen, as well as more complicated composite-widgets such as viewports which boast fully functional scroll-bars which can be used to drag a larger widget around inside the viewport. Widgets include their own event-handlers and will respond to events passed to them by the X Toolkit. Such events may require that the widget redraw or resize itself, or perhaps initiate some other user-defined function.

Widgets are made available to application programmers in *widget sets*. XSnap has been developed using the Athena widget set, which is the widget set distributed together with X-Windows. It should be noted that other more sophisticated widget sets are commercially available and include OSF's *Motif* and AT&T's *OPEN LOOK* widget sets. Although these do offer a wider array of features, the Athena widgets have been used since they are more readily available, and because the emphasis in writing XSnap has definitely not been to develop a *commercial* package. In any event, it should be relatively straight-forward to modify XSnap to use either of the other two widget sets, should this ever be deemed necessary.

7.2 Components of the XSnap package

Figure 12 shows a diagramme of the components of the XSnap package. These are:

- **The XSnap Graphical User Interface**
  This interface has been described in detail in chapter 6 of this dissertation. The interface has been written using the Athena widget set.

- **The CMTB**
  The CMTB has been described in detail in chapters 3 through to 5 of this dissertation, and is responsible for validating and solving models defined using the Xsnap GUI.

- **The SnapL Parser and Lexical Analyzer**
  The parser is responsible for parsing the SnapL evaluation-section code defined by the modeller to allow repetitive model evaluation and the manipulation of solution statistics. Mathematical expresssions used to define arrival rates, routing frequencies and customer service rates are also parsed and evaluated by the parser. The parser

is exactly the same as that implemented under MicroSnap, although obviously in XSnap there is no *definition-section* in the SnapL program.

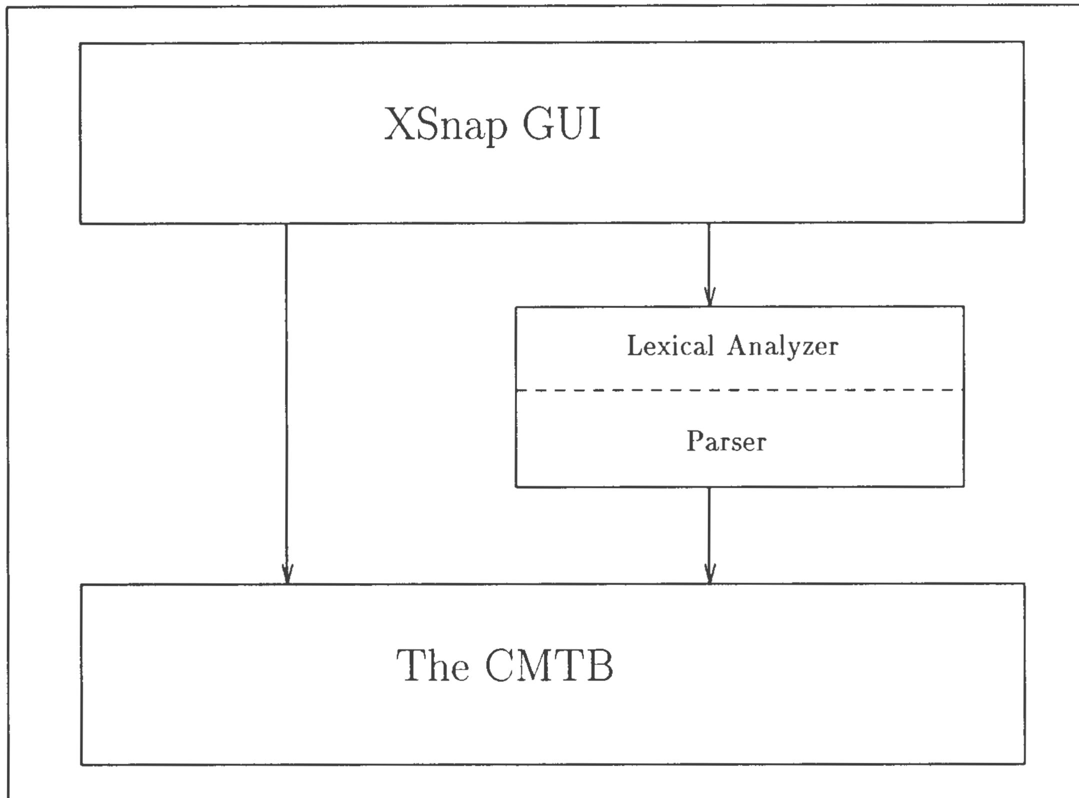


Figure 12: Components of the XSnap package

Since the implementation of the CMTB has already been discussed in this dissertation, the following sections will refer only to the XSnap GUI and the SnapL parser.

7.3 The XSnap GUI

Creating applications using *widgets* is a tedious excercise. It is, however, significantly less tedious than having to write the application without the benefit of widgets, and so all things considered, widgets are actually a *blessing* for the application programmer.

The XSnap GUI has been implemented using a number of modules:

- "xsnap.c" initializes the application and sets up the connection to the X Server. This module also sets up the XSnap screen as well as all the menu buttons and pop-up menus. Many of the widget resources are actually stored in the *application-defaults* file and are not hard-coded. This is to allow the appearance of the XSnap window to to be modified without having to re-compile XSnap.

- "xworkld.c" is repsonsible for effecting the Workload Manager described in the previous chapter and shown in figure 6.
- "xcanvas.c" is repsonsible for controlling the canvas. This includes the drawing, resizing and scrolling of models.
- "xcentre.c" is used to effect the Centre Manager described in chapter 6 and shown in figure 7.
- "xpath.c" is used to control the adding of new paths to the model and the Path Manager shown in figure 8. The Path Manager has also been described in chapter 6.
- "xfile.c" is used to write models to disk, or to read saved models into XSnap.
- "xsolve.c" is used to initiate the solution algorithms in the CMTB. A number of functions added to "parser.c" are used to copy the XSnap model to the CMTB, and are called from within "xsolve.c".
- "utils.c" includes a number of utility functions used by each of the above modules. Such utilities can be used to create pop-up windows, dialogues and toggle-sets. Furthermore, functions for creating, searching, appending and otherwise manipulating lists of strings are provided by this module. These are used extensively by the Workload, Centre and Path Managers in XSnap.

7.3.1 The XSnap GUI data structures

The data structures used by the XSnap GUI to store the model are quite straight-forward, and relatively unexciting. The information stored maps very closely onto that displayed in the Workload, Centre and Path Managers described in the previous chapter. Additional data is used to store the positions of each centre and the segments of each path in the model. Centres and workloads are stored in alphabetically sorted binary trees, whilst paths are stored in a linked list.

7.3.2 Managing the Canvas

The most complicated part of the XSnap GUI, is that which is responsible for managing the graphical representation of the model which is displayed on the canvas.

7.3.2.1 Scrolling

The Athena widget set does not provide any widget that can be used directly to manage the canvas. Instead, a 'simple' widget is used, and the model diagram is drawn directly onto the screen using the X Lib drawing primitives.

XSnap keeps a record of the size and position of the canvas, relative to origin of the viewport. Whenever the canvas is resized, XSnap 'informs' the viewport scrollbars of the new size and position of the canvas, and also redraws the model. Normally, widgets are responsible for their own geometery management, and would automatically inform their parent widget of any change in their geometery. Futhermore, widgets each include their own event-handlers which would be used to automatically redraw themselves whenever they received a 're-display' event from the X Server. Since the 'simple' widget has no knowledge of the model diagram, XSnap needs to intercept this event so that the model can be redrawn manually.

7.3.2.2 Resizing

To allow the 'zooming' and 'shrinking' of models, a *scaling factor* is used when mapping the position and size of each the model components to and from the screen.

When 'zooming' the model, XSnap needs to change the scaling factor *and* reposition the canvas. The canvas needs to be repositioned so the model diagram grows out and around the viewport, rather than just down and to the right.

Similarly, when 'shrinking' a model, XSnap needs to both rescale and reposition the canvas. There is an additional complication, however, whenever the model is shrunk so much that it no longer fills the viewport. In this case, the position of each of model components stored in the model data structures needs to be *shifted* so that the model shrinks in towards the centre of the viewport, rather than up and left towards the origin.

7.3.3 Solving the Model

Before a model can be solved, it needs to be copied to the CMTB. If a model already exists in the CMTB data structures, then this is first removed by a call to the module "scrapmod.c" in the CMTB.

The XSnap model is copied to the CMTB using functions in "parser.c". These functions have been grouped together with the parser since they make very similar calls to the CMTB as those routines used by MicroSnap to parse the *definition-section* of a SnapL program. Where a model definition includes expressions defining centre service rates, routing probabilities etc. these are interpreted and evaluated by the parser.

Once the model has been copied to the CMTB, the routing information is first validated and then the relative throughputs are solved using the appropriate functions provided by the CMTB. Thereafter, a dummy solution statistic is requested from "stat41.c" which then invokes the MVA algorithm. Once the MVA solution statistics have been found, these are automatically tagged on the model data structure used by the CMTB, where they will

be used by "stat41.c" to calculate solution statistics requested in subsequent calls to the CMTB. Such calls are made by the Workload Manager and Centre Manager whenever the modeller requests to see a table of solution statistics (see figures 10 and 9).

7.3.4 Evalution-section code

To interpret an evaluation section, XSnap simply passes the evaluation-section code to the parser. The parser automatically writes the program output and a program trace to disk. These are then viewed from within XSnap using a "AsciiText" widget.

The implementation of the parser is described in more detail in the following section.

7.4 The SnapL parser

7.4.1 Backus Naur Form of the SnapL language

This subsection gives the formal definition of the SnapL language. The productions associated with the *definition-section* of a SnapL program are not supported by XSnap. Also, plot statements are not yet supported by XSnap.

- Terminal symbols are denoted enclosed in single quotes (' ').
- An arrow is used to separate the left and right sides of a production (->).
- Alternatives are denoted using the vertical bar symbol (|).
- Optional clauses or items are enclosed in square brackets ([ ]).
- An occurrence of 0 or more of an item will be enclosed in braces ({ }).
- Groupings of items or optional items from which one must be chosen will be enclosed in round brackets ().
- Non-terminal symbols are denoted in lowercase lettering.
- Terminal symbols in English lettering may be either upper or lowercase letters. Here we use uppercase lettering for the sake of clarity.
- A range of terminal symbols over a set of well known values is linked with a series of periods (..).
- *Italicised items* are not supported in the current implementation.

program	->	label def_sec eval_sec '.'
def_sec	->	[opt_stmt] {vrb_stmt} wrkld_stmt {wrkld_stmt}

{glbl_stmt} config_para
opt_stmt -> 'OPTIONS' param '=' integer
{',' param '=' integer} ';'
param -> 'CENTRES' | 'CENTERS' | 'CLASSES' |
'MARGIN' | 'COLWIDTH' | 'LABELS' |
'NODES' | 'SYMBOLS' | 'VARIABLES' |
'WORKLOADS' | 'LABLENGTH' |
'PAGELENGTH' | 'PAGELENGTH' |
'PAGEWIDTH' | 'PLOTLENGTH'|
'PLOTWIDTH' | 'PLOTTITLE' |
'ACCURACY' | 'THRU_SOLVE'
vrb_stmt -> 'VARIABLE' defvariable {',' defvariable} ';'
defvariable -> label ['(' defexpression ')']
variable -> label ['(' expression ')']
label -> ('A'..'Z') {'0'..'9' | 'A'..'Z' | '_'}
wrkld_stmt -> 'WORKLOAD' label ['PRIORITY']
[class_clause] (mpl_clause | arriv_clause) ';'
class_clause -> 'CLASS' label_list
label_list -> label {',' label}
mpl_clause -> 'MPL' (defexpression | defexpression '#' defexpression
{','defexpression '#' defexpression})
arriv_clause -> 'ARRIVAL' defexpression ['AS' label]
config_para -> {*'SUBMODEL' label ';' [opt_stmt] config_para*
'END' ';'} centre_stmt {centre_stmt} route_stmt
{route_stmt}
centre_stmt -> ('CENTRE' | 'CENTER') defvariable
specification ';'
specification -> 'MICRO' | 'PRIORITY' time_clause |
'FCFS' *[speed_clause]* 'TIMEALL' defexpression |
pdl_clause
time_clause -> 'TIMEALL' defexpression |
{'TIME' class_list defexpression}
speed_clause -> *'SPEED' defexpr_list*
expr_list -> expression {',' expression}
defexpr_list -> *defexpression* {*','* *defexpression*}
class_list -> class {',' class}
class -> label [':' label]

```
pdl_clause       ->  ('PS' | 'DELAY' | 'LCFS') [speed_clause]
                     time_clause
route_stmt       ->  'ROUTE' label route_descr {route_descr} ';'
route_descr      ->  'FROM' (node_list 'TO' node |
                     node 'TO' node_list 'FREQ' expr_list)
node             ->  [class '@'] (variable | 'ENTRY' | 'EXIT')
node_list        ->  node {',' node}
centre_list      ->  variable {',' variable}
eval_sec         ->  'BEGIN' {eval_stmt} 'END'
eval_stmt        ->  (prnt_stmt | loop_stmt | plot_stmt |
                     tabulate_stmt | report_stmt | glbl_stmt |
                     if_stmt | while_stmt)
prnt_stmt        ->  'PRINT' (expression | string)
                     { ',' expression | string} ';'
string           ->  ' {character} '
character        ->  NUL .. DEL
loop_stmt        ->  'LOOP' variable '=' expression 'TO' expression
                     ['BY' expression] {eval_stmt}
                     'ENDLOOP' ';'
plot_stmt        ->  'PLOT' (pie_plot_stmt | bar_plot_stmt |
                     line_plot_stmt)
pie_plot_stmt    ->  'PIE' string variable 'TITLE' stringlist ';'
bar_plot_stmt    ->  'BAR' string variable 'TITLE' string
                     {',' variable 'TITLE' string}
                     'VS' stringlist 'TITLE' string ';'
line_plot_stmt   ->  'LINE' string variable 'TITLE' string
                     {',' variable 'TITLE' string}
                     'VS' variable 'TITLE' string ';'
stringlist       ->  string { ',' string}
tabulate_stmt    ->  'TABULATE' variable 'TITLE' string
                     {',' variable 'TITLE' string}
                     'VS' variable 'TITLE' string ';'
report_stmt      ->  'REPORT' (stat_list | 'ALL') (class_list | 'ALL')
                     '@' (centre_list | 'ALL') ';'
if_stmt          ->  'IF' expression 'THEN'
                     {eval_stmt}
                     ['ELSE' {eval_stmt}]
```

```
                       'ENDIF' ';'
while_stmt      ->     'WHILE' expression 'DO'
                       {eval_stmt}
                       'ENDWHILE' ';'
stat_list       ->     statistic {',' statistic}
statistic       ->     'QUEUE' | 'QTIME' | 'WAIT' | 'CYCLE' |
                       'RATE' | 'UTIL' | 'RES'
glbl_stmt       ->     let_stmt | model_stmt | mod_wrkld |
                       mod_centre | mod_route | parms_stmt |
                       incl_stmt
let_stmt        ->     'LET' variable '=' expression ';'
model_stmt      ->     'MODEL' [string] ';'
mod_wrkld       ->     'MODIFY_WORKLOAD' label ['PRIORITY']
                       [mpl_clause | 'ARRIVAL' defexpression] ';'
mod_centre      ->     'MODIFY_CENTRE' | 'MODIFY_CENTER'
                       variable specification ';'
mod_route       ->     'MODIFY_ROUTE' label ['SUBMODEL' label]
                       [route_descr] ';'
parms_stmt      ->     'PARAMS' ('WORKLOAD' [label] |
                       ('CENTRE' | 'CENTER') [label] |
                       'ROUTE' [label]) ';'
incl_stmt       ->     'INCLUDE' string ';'
expression      ->     term [relop expresssion]
term            ->     primary [addop term]
primary         ->     factor [multop primary]
factor          ->     variable | '( ' expression ')' | const | stat_sel
                       | param_sel | unary factor | 'min ( ' expr_list ' )'
                       | 'max ( ' expr_list ' )'
                       | 'range ( ' expression ',' expression ',' variable ' )'
                       | 'power ( ' expression ',' expression ' )'
defexpression   ->     primary [addop defexpression]
defprimary      ->     factor [multop defprimary]
deffactor       ->     defvariable | '( ' defexpression ')' | const
                       | unary factor | 'min ( ' expr_list ' )'
                       | 'max ( ' expr_list ' )'
                       | 'range ( ' defexpression ',' defexpression ',' defvariable ' )'
                       | 'power ( ' defexpression ',' defexpression ' )'
```

unary -> '+' | '–' | 'NOT'

relop -> '<' | '>' | '=' | '<=' | '>=' | '<>' | 'AND' | 'OR'

addop -> '+' | '–'

multop -> '*' | '/'

const -> real | integer

integer -> ('0'..'9') {'0'..'9'}

real -> ('0'..'9') {'0'..'9'} '.' {'0'..'9'} ['e'['-'] ('0'..'9') [('0'..'9')]]

stat_sel -> (statistic | 'ALL') ((class_list | 'ALL')
'@' (centre_list | 'ALL')) | 'RES' label

param_sel -> 'ARRIVAL' label |
'MPL' label [',' defexpression] |
'TIME' node |
'FREQ' node 'TO' node |
'SPEED' variable [',' defexpression]

7.4.2 Lexical Analyzer

The lexical analyzer is responsible for reading the SnapL program and converting the stream of input characters into tokens. These tokens are then passed to the parser, to be used in the syntax-directed translation of the SnapL program.

Readers interested in the design and implementation of lexical analyzers are referred to the excellent book by Aho, Sethi and Ullman [Aho 86].

Save for the following specialised features, the lexical analyzer is quite standard:

- A panic mode is provided where each line read is prefixed in the output with the words "LINE IGNORED :". This mode can be used whenever the parser has become totally confused and is searching for the next useful token, such as a semi-colon.
- A literal string mode is supported where each character is returned without any case modification. This is necessary for reading in strings to be printed as headings in a TABULATE statement, or strings in a PRINT statement.
- No bookkeeping or symbol table manipulation is done by the lexical analyzer, but is rather left to the parser.

7.4.3 The Parser

The parser used by XSnap actually functions as an *interpreter* since the SnapL code is not translated into any other form before being executed. Rather the code is 'executed' as it is being parsed. The SnapL code is not compiled into any lower level language since it

is almost certain that the improvement in execution time that could be gained from such an excercise would be insignificant when compared to the solution time of the CMTB solution modules. The parser has been shown to run very quickly, and any delay caused by the direct interpretation of the evaluation section has been unnoticable.

XSnap uses a simple recursive-decent parser to parse the SnapL *evaluation-section* code defined by the modeller. The procedures used in the code map closely onto the productions given in the formal definition of the SnapL language (ie. each production is represented by a procedure having the same name). One can extend the language by merely changing the productions given in section 7.4.1 above, and by modifying the corresponding procedures in the source code. Readers unfamiliar with recursive-decent parsers are once again referred to the "Dragon Book" written by Aho, Sethi and Ullman [Aho 86].

When first implemented in MicroSnap, the parser was split into two modules:

1. the module "parser.c", which parses the *definition section* and the MODIFY_WORKLOAD, MODIFY_CENTRE and MODIFY_ROUTE statements.

2. the module "eval41.c", which parses the evaluation section of the SnapL language.

In XSnap, there is no need for a definition section. However, many of the functions used when first defining a model are also used when the model is being modified with a MODIFY_WORKLOAD, MODIFY_CENTRE or MODIFY_ROUTE statement. Consequently, both "parser.c" and "eval41.c" have been included into XSnap.

7.4.3.1 Overview of passes

The SnapL code is parsed once to check for correct syntax. Once this has been done, the evaluation code is interpreted. For this reason, MicroSnap has sometimes been described as a 'two-pass' interpreter. However, it would be incorrect to call the *interpretation* of the evaluation section as a 'pass', since any loop statement would cause the interpreter to travel over the code more than once.

7.4.3.2 Reporting errors – the `error()` function

This function reports any errors to the user, and has been only slightly modified in XSnap to present errors in a pop-up window rather than via standard output. If an errors occur whilst the evaluation section is being interpreted XSnap will abort the batch run.

The `error()` function accepts a textual string describing the error, and this makes for very readable source code since a textual explanation of each error that may arise is given in each function where the error could be detected. This is obviously far more helpful than

error codes. Admittedly, some textual error descriptions are duplicated, thereby wasting a small amount of data space. However, using error codes rather than strings would make a miniscule dent in the data storage problems presented by the CMTB solution algorithms.

The function `error()` also keeps a tally of the number of errors that have been reported.

7.4.3.3 Panic mode and aligning tokens

The parser uses a very simple token alignment system, which is used when trying to recover from syntax errors. The parser recognises the semi-colon as the only useful 'solid' token. This is a logical choice as all statements in SnapL are right-delimited by a semi-colon.

The function `align_semicolon()` will check that the next token is indeed a semi-colon. If not, an error is reported and all tokens discarded until a semi-colon is reached. Whilst discarding tokens the lexical analyzer is set in 'panic' mode where each line echoed is prefixed "Line ignored: ".

The method of error recovery adopted by the XSnap parser is by no means state-of-the-art, and could be significantly improved through the use of FIRST and FOLLOW sets, and more sophisticated matching functions. Nonetheless, the parser is robust, and will report correctly all errors found.

7.4.3.4 Declaring variables – the VARIABLES statement

The function `vrb_stmt()` is responsible for parsing the VARIABLES statement. It calls the function `addtovartree()` in the module "book41.c" to add the variable declaration to the variables tree. The variable name and subscript (if any) are passed as parameters.

This function used to parse variable definitions is used by XSnap, even where there may be no evaluation section for the model. The defined variables can then be used in mathematical expressions defining arrival rates, centre service rates or routing frequencies.

7.4.3.5 Parsing expressions

Expressions are parsed as defined by the SnapL productions, with the precedence of operators enforced by the structure of the productions themselves. A distinction is made, however, between 'defexpressions' and 'expressions'.

- defexpressions are expressions that can be used when *defining* the model, and exclude the use of relation or logical operators such as $<$, $>$, $\leq$, 'NOT', 'OR' etc. as well as statistic or 'params' selectors.

- normal expressions allow the use of all the operators and selectors mentioned above.

Chapter 8

Performance and Testing

This chapter will give a report on the performance of XSnap, and will also describe the procedures that have been used to test XSnap.

Much of the testing of the CMTB was tackled when MicroSnap was first being written. Further integration testing has obviously being needed to ensure that the CMTB has been integrated successfully into XSnap. In this chapter, both the original testing and the subsequent "integration testing" will be described.

8.1 Performance Analysis

When talking about the performance of XSnap, we are mainly refering to the space and time requirements for solving different types and sizes of models. This performance is not a function of the implementation of the XSnap GUI, but rather the CMTB.

Admittedly, one *could* talk about the performance of the package whilst refering to things such as "ease of use" or special features offered by the XSnap GUI. However, the XSnap GUI has already been described is detail in chapter 6, and will therefore not be included in the discussion presented in this section.

Throughout this section simple formulae have been presented to illustrate the effect that certain model parameters have on the time and space requirements for solving models. These formulae are not necessarily *exact*, and are intended only as 'in the order of' approximations. They do, none the less, provide valuable insight into many performance issues in XSnap.

8.1.1 Performance criteria

The most important performance criteria in the implementation of the CMTB has been space efficiency. One should remember that the CMTB was originally developed as part of MicroSnap in a PC environment where space constraints are all too real.

Admittedly, in a UNIX environment run-time storage is not nearly as dear as under DOS, and therefore much larger models can be solved by XSnap than by a DOS-based version[1] of MicroSnap. Even in XSnap, the storage requirements of the CMTB can *still* limit the size of models that can be solved, although admittedly solution time would probably be a more significant factor in the larger models.

XSnap has been designed as an *interactive* modelling tool, and therefore much of the success of the package depends on the speed with which it can solve models.

The two procedures that require the most memory in the CMTB are the MVA algorithm and solving relative throughputs. The memory requirements for the MVA algorithm are affected quite significantly when PRIORITY servers are introduced into the model, and are also affected slightly by the use of the interpolation algorithm.

8.1.2 MVA Memory Requirements

Table 1 gives a summary of the memory requirements of the MVA algorithm implemented in the CMTB. The notation adopted in the table is as follows:

- N is the total number of centres in the model.
- P is the number of PRIORITY centres in the model.
- J is the number of closed chains.
- r is the total number of classes in the model.
- x is the total number of classes in the model belonging to *open* chains.
- s is the number of classes belonging to *closed* chains in the model.
- ψ is defined in equation 11
  (i.e. $\psi = \prod_{k=2}^{J}(L_k + 1)$, where L_k is the population of closed chain k).

8.1.2.1 MPL levels

The memory requirement of the MVA algorithm is proportional to the product of the populations of all but one of the closed chains in the model. The total memory requirement therefore rises exponentially with the addition of each new workload in the model.

We note that increasing the MPL level of any *one* of the chains in a given model would not increase the storage requirement, but would only increase the solution time by the same proportion. This is because the value ψ is a product over all but one of the

[1] MicroSnap has also been ported to UNIX.

Table 1: MVA memory requirements

Memory Item	Number of elements	Space per element
BASIC MVA		
rho_chain	$N \times J$	8
xi_chain	$N \times J$	8
rho_centre	J	8
avg_inpl_chain	J	8
m_Tstar_chain	J	8
m_Wstar_chain	$J \times N$	8
m_Q_centre	$J \times N$	8
m_Q_cust	$N \times \psi$	8
m_pop_chain	J	2
INTERPOLATION		
m_Tstar_chain	J	8
m_Wstar_chain	$J \times N$	8
m_Q_centre	$J \times N$	8
PRIORITY CENTRES		
m_W_cust	$r \times P \times \psi$	8
m_T_cust	$J \times \psi$	8
W_open	$P \times x$	8
equation 16	s^2	8

chain's MPL levels. The function `put_wrklds_into_array()` of "book41.c" ensures that the chain with the highest MPL level will be that one excluded from the calculation of ψ (see Chapter 5).

8.1.2.2 No PRIORITY servers

From table 1 is can be seen that the total memory requirement for the MVA algorithm is

$$J(26 + 32N) + 8N \times \psi \text{ bytes}$$

assuming that there is no interpolation necessary and that there are no PRIORITY servers in the model. Where interpolation is necessary, the memory requirement increases to

$$J(34 + 48N) + 8N \times \psi \text{ bytes}$$

In the above expressions, the most important factor is obviously ψ, since this value increases exponentially with the additional of each new closed chain to the model.

8.1.2.3 PRIORITY servers

When a model includes PRIORITY servers we need to allocate an *additional* amount of storage equal to

$$8Px + (rP + J) \times \psi + 8s^2 \text{ bytes}$$

In other words, for the larger models, the storage requirements for solutions involving PRIORITY centres is proportional to

$$\begin{aligned} \psi \quad \times \quad & [\text{the number of centres in the model} + \\ & \text{the number of closed chains} + \\ & \sum_{\text{priority centres}} (\text{ number of classes that visit that centre })] \end{aligned}$$

where each line of the above equations represents storage for the arrays `m_Q_cust[][]`, `m_T_cust[][]` and `m_W_cust[][][]` respectively. Clearly these models require significantly more storage than similar models without PRIORITY servers.

8.1.2.4 Maximum number of chains

Each chain in the model must have a positive MPL level which, after rounding up all MPL levels for interpolation, will be greater than or equal to 1. For each k, $(L_k + 1)$ therefore has a minimum value of 2, and ψ must be greater than or equal to 2^{J-1}.

It is interesting to consider the maximum number of chains that could be included in any one model. Given that there must be at least one centre in the model, and also that

ψ is always greater than or equal to 2^{J-1}, we can derive the absolute minimum memory requirement for a model with J closed chains to be

$$58J + 8 \times 2^{J-1} \text{ bytes}$$

In a DOS system with an available heap of roughly 400 KBytes, J would then have to be less than or equal to 16.

The Sun SPARC 1+ station on which XSnap was developed reported a maximum available memory area of 511 MBytes, yet even on such a machine J could be at most equal to 26. Obviously, solving a model which required over 500 MBytes of intermediate solution results would in most cases take an awfully long time!

If the MPL levels of two or more chains are greater than 1, or if there is more than one centre in the model, then the model would have to contain far fewer closed chains than that derived above.

We turn now to time requirements for the MVA algorithm.

8.1.3 MVA Time Requirements

We can work out the time requirements of the MVA algorithm by looking at each of the steps given in figure 3 in chapter 5.

8.1.3.1 No PRIORITY centres

The main loop over all closed chain population vectors involves $\prod_{j=1}^{J}(L_j + 1)$ iterations, and steps 4 and 5 are carried out J times for *each* of these outer loop iterations. Since the waiting time $W_{ik}^*(\vec{i})$ is then computed for *each* of the N centres, the total time taken by step 4 is proportional to Γ, where

$$\Gamma = JN \prod_{j=1}^{J}(L_j + 1) \tag{22}$$

Admittedly, the equation defining $W_{ik}^*(\vec{i})$ is very much simpler to evaluate for DELAY centres than it is for the other service disciplines. The effect of this is not likely to be significant and will therefore be ignored.

Step 5 is completed J times in every outer loop, and since the calculation of $T_k^*(\vec{i})$ involves a summation over all N centres, we can see that the total time spent executing this step is also proportional to Γ.

Step 6 involves calculating $Q_i(\vec{i})$ for each of the N centres, and each of these calculations includes a summation over all J chains in the model. Once again, therefore, the execution time for this step is proportional to Γ.

8.1.3.2 PRIORITY servers

Time requirements for solving models with PRIORITY servers is very much more complicated to derive.

To solve $W^*_{ik}(\vec{i})$ for PRIORITY servers i, we need to sum the values W_{ir} for all classes in the closed chain k (see figure 3). Calculating each W_{ir} by equation 13 involves an iteration over all classes in the model (or, if the chain k is a priority chain, an iteration over all classes in the model belonging to priority chains). However, due to the optimisations described in section 5.2.6.3, the iterations involved in finding W_{ir} only need to be performed once for each centre. This set of iterations will therefore be ignored in the following equations.

The total time spent solving $W^*_{ik}(\vec{i})$ (step 4) in a model with P PRIORITY servers is proportional to

$$[J(N-P)+PNs]\times\prod_{j=1}^{J}(L_j+1)$$

using the notation introduced in the sections above.

As was described in section 5.2.9, there is also an additional overhead in step 6 with the calculation of W_{ir} for all r classes belonging to *open* chains. To find these values we need to solve two sets of simultaneous linear equations. Since such sets of equations must be solved for each centre during each loop over the different population vectors, the *total* time spent solving these equations is proportional to

$$[a^3+b^3]\times N\times\prod_{j=1}^{J}(L_j+1)$$

where a is the number of classes belonging to open chains which have PRIORITY status, and b is the number of classes belonging to open chains which do not have PRIORITY status.

8.1.3.3 Summary of time requirements

All steps considered, the solution time for a model with no PRIORITY servers is proportion to the product of

- the number of centres in the model.
- the number of closed chains in the model.
- the product of the customer population levels (plus 1) of each of the closed chains in the model.

The solution time for models which include PRIORITY servers is very much more difficult to represent clearly, and depends on very many different factors. A significant

Table 2: Time and space requirements for solving a number of simple models

	Memory Requirement (bytes)			Solution Time (seconds)	
Chains	XSnap GUI	CMTB	MVA	Sun SPARC 1+	IBM PC/AT
2	8016	3744	612	0.11	3.75
3	11340	4176	1998	0.27	9.31
4	14664	4944	9384	1.49	51.02
5	17988	5376	52770	10.62	365.07
6	21312	6096	312156	77.00	2640.11

difference between a model with and without PRIORITY servers, is that the MVA solution time of the former is affected by the number of *classes* in the model (in both open and closed chains), while the latter is not. Another important difference is that it clearly takes *longer* to solve models with PRIORITY servers than it does to solve models of similar size which have no such PRIORITY servers.

8.1.4 Some sample models

As an indication of the performance of the MVA algorithm, table 2 has been compiled to illustrate the space and time requirements for solving 5 simple models having 2,3,4,5 and 6 closed chains respectively. Each of the models has no open chains, and each of the closed chains has an MPL level (closed chain population) of 5.

The Sun SPARC-Station 1+ used in the test had 8 MBytes of memory and no floating point accelerator. The IBM PC/AT had 640 KBytes of memory and no math co-processor, and ran at a clock speed of 16 MHz.

8.1.5 Adding Open Chains to the model

Adding one or more *open* chains to any of the above models would not affect the memory requirements of the MVA algorithm. In fact, the number of open chains in the model is constrained only by the storage space available to store the definition parameters and routing information for such chains, and those extra statistics ρ_{ik} and Ξ_{ik} calculated in the module "param.c". Even in a DOS environment, we might expect to be able to store the data for at least 100 such open chains. Clearly then, this places no real constraint on the model size.

8.1.5.1 Speeding up MVA

It should be noted that step 4 of the MVA algorithm given in figure 3 has to be carried out

$$JN \prod_{k=1}^{J} (L_k + 1)$$

times, every time the model is solved.

At present this step is implemented by the function `calc_W_chain()` in "mva41.c" and the time taken to execute this function is normally responsible for a significant proportion of the total solution time. In the simple model with four closed chains used in the example above, this function is called over 25000 times, while the model with five closed chains needs 194400 calls!. Such function calls are quite expensive, and this suggests that one of the easiest ways to speed up the MVA algorithm would be to move the function `calc_W_chain()` inline into `mva()`. This would, however, sacrifice the readability of the `mva()` function.

8.1.6 Relationship between memory and time requirements of the MVA algorithm

It was pointed out in section 8.1.2.4 that the maximum number of chains that could be included in a model would be affected by the available memory *and* the time taken to solve a model. We might wish to consider which one of these performance criteria can be expected to be more prohibitive when it comes to solving large models. To establish this, we will attempt to derive a relationship between the solution time and the memory requirement of large models.

If we consider the value of Γ for each of the sample models used in table 2 and compare this value to the actual solution time, we see that for the larger models Γ is roughly 18000 times the solution time in seconds (on the Sun SPARC 1+). Given that the memory requirement for large models is roughly equal to $8N \times \psi$ bytes, and that $\prod_{j=1}^{J}(L_j + 1)$ is at least equal to 2ψ, we can derive an equation which gives the *minimum* solution time for models as a function of the MVA memory requirement for solving the model. We have

$$\text{Solution time} \geq \frac{2NJ \times \psi}{18000} \text{ seconds}$$

and therefore

$$\text{Solution time per MByte of storage} > 14J \text{ seconds}$$

For ψ to be greater than 1, there must be at least two chains in the model, and so in 'large' models the right hand side of the above equation must be at least 28.

Admittedly, any model that used more than 8MBytes of storage would require memory-paging by the operating system, which would slow down the application. The above minimum solution time of 28 seconds per MByte of storage is therefore very conservative.

Another consideration is that $\prod_{j=1}^{J}(L_j + 1)$ can be very much larger than ψ times 2, and is in fact ψ times the *largest* closed chain population level in the model. With this in mind, the solution time would more likely be as much as 10 minutes per MByte of storage used, ignoring the additional overhead involved with paging.

In summary then, it is difficult to establish a simple relationship between the MVA solution time and MVA memory requirements which would be applicable to *all* models. We might expect, however, that on a Sun SPARC 1+ workstation the solution time will be somewhere between 1 and 20 minutes per MByte of storage used. A more acturate estimate of the relationship can only be given where both the number of chains and the largest closed chain population are known.

8.1.7 Solving Relative Throughputs

8.1.7.1 Memory requirements for LU-Decomposition

When solving the relative workload throughputs, each chain is solved separately and only those nodes that are actually *visited* are included in the matrix representation of equation 7 given in chapter 4.

If w represents the total number of visited nodes in the chain, then this set of equations will take up

$$8w(w + 2) \text{ bytes}$$

of memory. Admittedly, the routing probabilities also need to appear in the CMTB data structures where they will take up additional storage space. However, the CMTB uses a sparse matrix representation of the data which, for large models, is likely to take up very much less space than that given by the equation above. The CMTB data structures will therefore be ignored in the rest of this analysis.

Using the equation above, we can see that in a DOS environment with a heap of approximately 400KBytes we could represent the matrix for a chain with up to 225 visited nodes.

Under UNIX, very much larger matrices could be used. For example, 8MBytes would be sufficient to hold a matrix representing 1000 visited nodes in a single chain!

Practically then. memory requirements place no real constraint on the solution of relative throughputs using LU-Decomposition.

8.1.7.2 Memory requirements for the Conjugate Gradient method

The above estimates have been derived assuming that the LU-Decomposition method is to be used to solve the relative throughputs. When using the Conjugate-Gradient method we do not need to store the matrix representation of this set of linear equations, and instead use the routing frequencies directly as they appear in the model data structures. This is a significant saving for larger models.

Unfortunately, as has been pointed out in section 4.5.1, the Conjugate Gradient method has proved to be inaccurate when used with larger models, and more work would be necessary to improve the accuracy of the algorithm. This is beyond the scope of this dissertation.

8.1.7.3 Time Requirements

Quite apart from the problem of space, we have the problem of time efficiency.

Unlike with the MVA algorithm, the time requirement for solving sets of simultaneous linear equations increases exponentially faster than the memory requirements. Using standard notation, the memory requirement for representing the set of simultaneous linear equations is $O(n^2)$ while the time requirement for solving these is typically $O(n^3)$. With this in mind, one can expect the time taken to solve the relative throughputs to become prohibitive long before one runs into any memory problems.

It has been pointed out that the solution of a set of *sparse* linear equations can be done in less than $O(n^3)$ using specialized algorithms. As demonstrated in the CMTB, however, these algorithms are not always as accurate as their $O(n^3)$ counterparts and therefore cannot always be used on the larger models.

8.2 Testing XSnap

The CMTB has been tested thoroughly at each stage during its development. Furthermore, the parser and lexical analyzer have been used extensively as part of MicroSnap since the end of 1990.

8.2.1 Testing the CMTB solution results

The code for each of the statistical solution modules ("mva41.c", "stat41.c" and "through41.c") has been checked thoroughly by inspection. However, since such inspection cannot be guaranteed to find every error, a number of test models have been used to demonstrate that the CMTB works correctly.

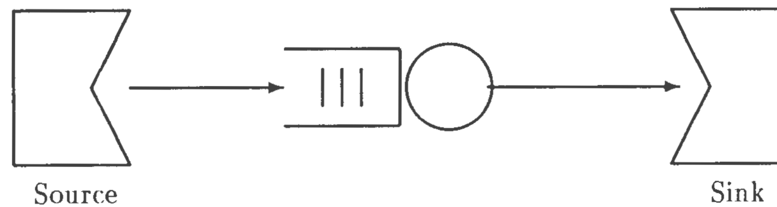


Figure 13: M/M/1 Queue.

In December 1990, Andrew Hutchison of the Computer Science Department (UCT) wrote a paper comparing the solution results of a number of modelling packages [Hutch 90]. The packages investigated included (amongst others) MicroSnap, RESQ, HIT and MACOM.

In Andrew's study, the MACOM, RESQ and MicroSnap experiments were run on a Sun SPARC 1+ Workstation with 8 MBytes of memory and no floating point accelerator. The HIT results were obtained through collaboration with the University of Dortmund, Germany, which provided their own results for the models tested using the HIT package; these HIT experiments were run on a Sun 4. Both of the HIT solution methods used (DOQ4 and MARKOV) fall into the class of analytical methods and are not simulations.

In the following sections, the results produced by XSnap will be compared with those tabulated by Andrew. The XSnap models were solved on the same machine as the MACOM, RESQ and MicroSnap models.

8.2.1.1 A simple M/M/1 queue

The first model used is quite simple, and was developed only to compare the output from the various tools with the know theoretical results.

Figure 13 depicts the model of an isolated M/M/1 queue with FCFS scheduling. A mean arrival rate of 10 customers per unit time ($\lambda = 10$) and a mean service rate of 20 customers per unit time ($\mu = 20$) was used in the model. In the case of RESQ and the HIT Simulator a total of 10 000 events was used.

The results are given in Table 3. It can be seen from this table that the results for MicroSnap and XSnap are both exact in terms of queueing theory.

8.2.1.2 Central server model

The second model used was the Central Server model, which is considered to be the classical example of a queueing network.

In every case except RESQ, it was modelled as an open network with external arrivals at the rate of $\lambda = 1/3$ per second. In the case of RESQ it was modelled as a closed

Statistic	RESQ	Micro-Snap	MACOM	HIT DOQ4	HIT MARKOV	XSnap	Theory
Utilisation	0.50	0.50	0.50	0.50	0.50	0.5	0.50
Throughput	9.80	10.00	10.00	10.0	10.0	10.00	10.0
Queue Length	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Queueing Time	0.051	0.050	0.050	0.050	0.050	0.050	0.050
Waiting Time	0.101	0.100	0.100	0.100	0.100	0.100	0.100
Solution time	1.0s	0.1s	35.0s	1.5s	6.2s	0.1s	n/a

Table 3: Results for the M/M/1 Queue

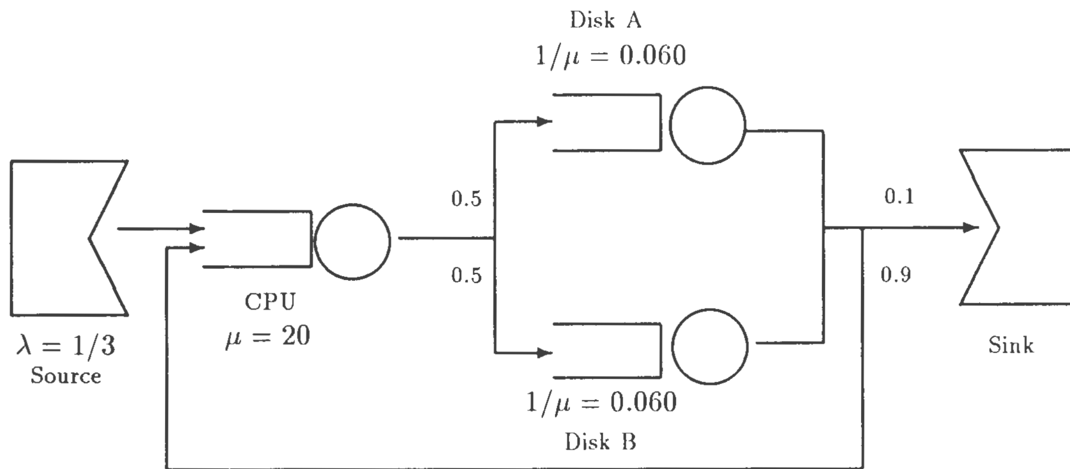


Figure 14: The Central Server Model.

network with a population of 11 customers corresponding to a throughput of about 1/3 on the feedback loop.

Figure 14 shows the central server model, whilst a picture of the model implemented under XSnap has already been shown as the sample model used in figure 4.

In the cases of Sandy, RESQ and the HIT Simulator a total of 100 000 events was used. Since open chains are not implemented in the version of RESQ which Andrew used, no response time was computed in that case. In all four cases the CPU was given a Processor Sharing service discipline while the disks were each given a FCFS service discipline.

Tables 4, 5 and 6 show the solution results and solution times for this model.

An interesting point is that for each of the models tested, both MicroSnap and XSnap have been shown to be significantly faster than *any* of the other packages used.

Statistic	RESQ	MicroSnap	MACOM	HIT DOQ4	HIT MARKOV	XSnap
Utilisation	0.165	0.167	0.160	0.167	0.167	0.167
Throughput	3.230	3.333	3.201	3.333	3.333	3.333
Queue Length	0.189	0.200	0.186	0.200	0.200	0.200
Queueing Time	0.009	0.010	0.008	0.010	0.010	0.010
Waiting Time	0.059	0.060	0.058	0.060	0.060	0.060

Table 4: CPU Statistics in the Central Server model

Statistic	RESQ	MicroSnap	MACOM	HIT DOQ4	HIT MARKOV	XSnap
Utilisation	0.100	0.100	0.096	0.100	0.100	0.100
Throughput	1.625	1.667	1.601	1.667	1.667	1.667
Queue Length	0.105	0.111	0.105	0.111	0.111	0.111
Queueing Time	0.005	0.007	0.006	0.007	0.007	0.007
Waiting Time	0.065	0.067	0.066	0.067	0.067	0.067

Table 5: Statistics for Disk A (or B) in the Central Server model

Statistic	RESQ	MicroSnap	MACOM	HIT DOQ4	HIT MARKOV	XSnap
Response Time	n/a	1.267	1.189	1.267	1.267	1.267
Solution Time	22.0s	0.2s	46.0s	1.9s	51.5s	0.2s

Table 6: Throughput and Solution Times for the Central Server model

8.2.1.3 Three Node Computer Network

Andrew also compared the solution results of a model of a large 3-node computer network. Unfortunately, the model used does not seem to give a realistic representation of the operation of the 'timeout' on each of the trunks. Therefore, it has been decided not to reproduce this model in this dissertation. Suffice it to say, however, that realistic or not, MicroSnap once again gave results which were directly in line with each of the other packages.

8.2.1.4 A simple closed chain model

Each of the above examples uses only open chains in the model. One more sample model will therefore be presented consisting of two *closed* chains.

The sample model chosen is called 'ex.xsn', and the MicroSnap equivalent of this model has been used often as a teaching example. Since the model is relatively small it has been possible to check the solution results manually. The output produced is also identical to that produced by MicroSnap.

There are two workloads in the model, namely "batch" and "interactive". The former workload has two classes "trans" and "io_wait", while the latter has three classes "active", "query" and "update". The number of customers in workload "batch" is 10 whilst the number in "interactive" is 25. The sample screen given in figure 6 actually shows the definition of the workload "interactive" using the XSnap Workload Manager.

Figure 15 gives a simple picture of the model. Unfortunately, since the model includes a number of classes it is not possible to show simply the service rates and routing probabilities on the model diagramme. Rather these would be queried using the Centre Manager and Path Manager in XSnap. Rather than having to list all the model parameters in this section, the reader is referred to appendix B which shows the output produced using the "CMTB Text" option in the XSnap GUI. This output was generated using "ex.xsn" as the sample model.

8.2.1.5 Models with PRIORITY servers

In the case of PRIORITY centres, an interesting exercise is to set a number of centres as PRIORITY centres and ensure that all customers that visit these centres have PRIORITY status (or alternatively we ensure they all have normal status). The solution for such a model should be very similar to that where the centres are all FCFS and the customers have no PRIORITY status. The results of these two models will not be identical since solutions involving PRIORITY centres can only be found by approximation (see Section 5.2.6). A bug was actually found in the CMTB using the above test, and this has been removed

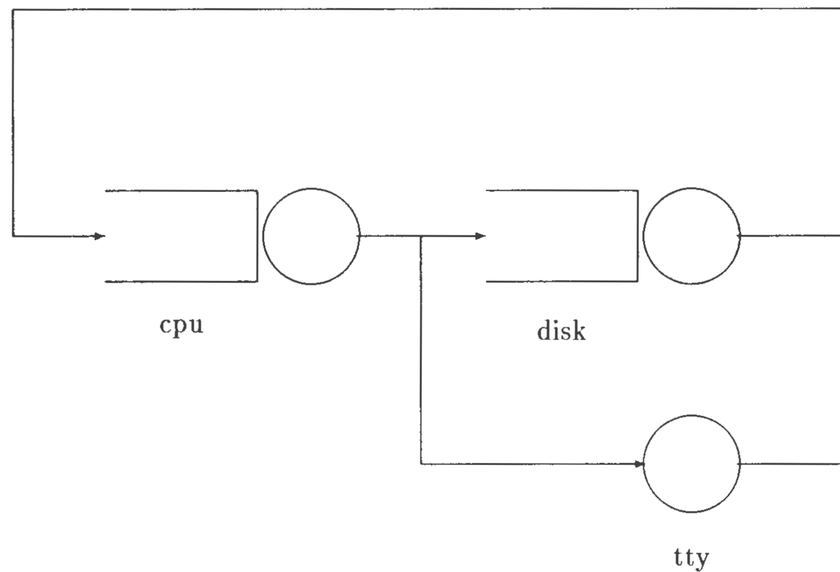


Figure 15: Sample Model 'ex.xsn'

successfully.

8.2.2 Testing the XSnap GUI

A graphical user interface is much easier to test than mathematical solution modules since it is normally immediately obvious whenever anything is wrong. Furthermore, each of the widgets on the screen either works or does not work, and is very rarely affected by any other part of the program.

A graphical user interface can only be tested interactively, with the modeller attempting to use as many features of the interface as possible, and ensuring that they all work correctly. Each of the features of the XSnap GUI described in chapter 6 has been tested using a number of dummy models.

8.2.3 Testing the other modules

The parser and lexical analyzer have been checked using numerous test programs, thereby ensuring that each of the productions of the SnapL language (see section 7.4.1) is parsed properly.

The bookkeeping functions have been checked using the PARAMS statement to interrogate the model data structures to ensure that all the parameters have been entered correctly into the CMTB data structures. Code inspection has also been useful in searching for bugs in the bookkeeping functions.

The LU-Decomposition and Conjugate-Gradient method algorithms have been validated by a simple check on the solution sum of square errors. Obviously, a large sum of

square errors would imply that the solution vector was inaccurate. These matrix algorithms have also been tested on simple sets of linear equations and the results checked by hand.

8.2.4 Results of the testing procedures

Obviously we cannot hope to expect that there are *no* bugs left in XSnap. However, the program has been thoroughly tested and both the CMTB and the SnapL parser have been complete and in use since the end of 1990. The few bugs that have surfaced since that time have been completely removed. Furthermore, it would be fair to say that any bugs that do remain are minor bugs that could be easily removed with the help of the MicroSnap Programmer's Manual and the source code comments.

It has been mentioned before that while programming with *widgets* is perhaps somewhat tedious, it is not necessarily complicated or difficult. Fixing any bugs that may appear in the XSnap GUI should therefore be relatively straight-forward.

XSnap has been only recently completed and has not yet enjoyed as much usage as MicroSnap. The scope for errors in the integration between the XSnap GUI and the CMTB, therefore, cannot be ignored. None the less, XSnap has been shown to work correctly on all models used thus far, and any errors found should be relatively easy to fix with the help of the source-code comments provided with the XSnap code.

Chapter 9

Conclusion

This chapter offers a summary of the main successes and any shortcomings of the project. A number of areas of possible future study are also suggested which may be undertaken to extend the package and to improve the overall quality of the final product.

9.1 Successes

An obvious success of the project has been the development of the XSnap package. Hopefully this package will be used productively for some years to come.

The package is user friendly, attractive and practical, and runs under X Windows which is a uniform and widely accepted platform. The model validation aspects are quite good in that XSnap will report any internal inconsistencies within a model, so as not to provide incorrect solutions based on such a model.

The package also usefully integrates both exact and approximation results. This is useful for modelling workloads with different priorities. Non-integral multiprogramming levels are also catered for automatically using interpolation approximations.

Sufficient flexibility is also offered by the tool to define complex model experiments through the use of the SnapL model specification and evaluation language. Very often, tools which offer a Graphical User Interface do so at the expense of system flexibility and power.

The dissertation also represents a useful reference which brings together much of the theory surrounding product-form queueing network solvers, and provides a practical example of the implementation of these somewhat complex mathematical algorithms.

Another significant success factor is the apparant speed with which XSnap is able to solve models. This has been highlighted by the comparison with similar tools in the previous chapter. Having a fast tool is always an advantage when doing interactive modelling.

Finally, and most importantly, XSnap has been shown to produce accurate solution statistics which are consistent with those produced by all the other tools with which it has been compared. The tool can therefore be used confidently when analyzing models.

9.2 Shortcomings

The main shortcoming of the XSnap package is the fact that the models which can be solved are largely restricted to that subset of models which can be represented using product-form queueing networks. Admittedly, a tool which calculated results using direct Markovian Analysis would be able to encompass a much wider domain of models. Such tools do, however, have their own shortcomings which have already been described in chapter 2.

There are perhaps additional features that could have been included in the package and which would have improved the package. Examples of such extensions are given in the next section. However, such features are probably best added to later versions after sufficient feedback has been gained regarding the use of the package; this way, one would be able to determine more clearly which additional features would add the greatest value.

One final shortcoming, perhaps, is that the package has not yet been extensively used. Some bugs in the packge may therefore still surface over time.

9.3 Further Research

There are many areas of further research that could be undertaken to improving the package. Some of the more obvious of these are listed below.

- The package could be extended to allow for the inclusion of Sub-Models. These would allow complicated models to be presented far more attractively and could also speed solution times (see [Krit 81]).

- The package could be extended to allow for the inclusion of load dependant servers. The CMTB already supports the data structures needed to store the information necessary to define servers of this type. The solution of such models is described in detail in [Krit 82]. Admittedly this would introduce another degree of complexity to the implementation of the MVA algorithm in the CMTB.

- The parser, used to parse the Evaluation Code used to define complex model experiments, could be improved through the use of FIRST and FOLLOW sets. This would allow more sophisticated error recovery (see [Aho 86]).

- The accuracy of the Conjugate Gradient method for solving simultaneous linear equations might be improved by manipulating the original matrix (see [Recipes 87]). This algorithm could then be used to calculate the relative throughputs of even the larger models, thereby speeding up the solution time.

- Estimation and Extrapolation algorithms could be integrated into the package to cater for closed workloads with very high MPL levels. A number of algorithms which may be adapted suitably for this purpose are described in [Recipes 87].

Appendix A

Sample Evaluation-Code Output

This appendix gives the sample output from the evaluation section of a model. The parameters of the model are the same as that given in appendix B.

A.1 Evaluation Code

```
testprog

VARIABLE v1(10), v2(10), v3(10), lop;

BEGIN   | the evaluation section |

report all all@all;

LOOP lop = 1 TO 10
     MODIFY_CENTRE disk fcfs timeall (lop*lop);
     | customer service time exponentially |
     PRINT 'Queue length@disk = ',  QUEUE all@disk;
     LET v1(lop)=lop*lop;
     LET v2(lop)= QTIME all@disk;
     LET v3(lop)= WAIT batcha:io_wait@disk;
ENDLOOP;

TABULATE v2 TITLE 'Queuing Time',
 v3 TITLE 'Waiting Time'
 VS v1 TITLE 'Service Time at Disk';
```

END.

A.2 The output produced by XSnap

```
XSnap V1.0           Model name: sample       Page:1
Sun Sep 20 16:56:00 1992


REPORT: Average Queue Length

                         batcha        batcha        interb
                          trans       io_wait        active
cpu                  0.00496619             0    0.00736734
disk                          0       9.99503             0
tty                           0             0             0

                         interb        interb
                          query        update
cpu                           0             0
disk                          0       10.8004
tty                     14.1922             0
```

```
XSnap V1.0           Model name: sample       Page:2
Sun Sep 20 16:56:03 1992

REPORT: Average Queue Time
                      batcha         batcha         interb
                       trans        io_wait         active
cpu               0.00247994              0     0.00122698
disk                       0        387.514              0
tty                        0              0              0

                      interb         interb
                       query         update
cpu                        0              0
disk                       0        403.993
tty                        0              0


XSnap V1.0           Model name: sample       Page:3
Sun Sep 20 16:56:03 1992

REPORT: Average Waiting Time
                      batcha         batcha         interb
                       trans        io_wait         active
cpu                  0.20248              0       0.101227
disk                       0        407.514              0
tty                        0              0              0

                      interb         interb
                       query         update
cpu                        0              0
disk                       0        423.993
tty                      300              0
```

```
XSnap V1.0            Model name: sample       Page:4
Sun Sep 20 16:56:04 1992

REPORT: Average Cycle Time
                      batcha        batcha        interb
                       trans       io_wait        active
cpu                  407.514             0       343.398
disk                       0       0.20248             0
tty                        0             0             0

                      interb        interb
                       query        update
cpu                        0             0
disk                       0       557.432
tty                   228.46             0


XSnap V1.0            Model name: sample       Page:5
Sun Sep 20 16:56:04 1992


REPORT: Average Throughput Rate
                      batcha        batcha        interb
                       trans       io_wait        active
cpu                0.0245268             0     0.0727804
disk                       0     0.0245268             0
tty                        0             0             0

                      interb        interb
                       query        update
cpu                        0             0
disk                       0     0.0254732
tty                0.0473073             0
```

```
XSnap V1.0          Model name: sample       Page:6
Sun Sep 20 16:56:04 1992

REPORT: Average Utilisation
                       batcha        batcha        interb
                        trans       io_wait        active
cpu                0.00490537             0    0.00727804
disk                        0      0.490537             0
tty                         0             0             0

                       interb        interb
                        query        update
cpu                         0             0
disk                        0      0.509463
tty                         1             0


XSnap V1.0          Model name: sample       Page:7
Sun Sep 20 16:56:04 1992

REPORT: Average Residence Time
                       batcha        batcha        interb
                        trans       io_wait        active

  Total             Undefined     Undefined     Undefined

                       interb        interb
                        query        update

  Total             Undefined     Undefined
```

```
XSnap V1.0          Model name: sample       Page:8
Sun Sep 20 16:56:42 1992

Queue length@disk = 10.2536
Queue length@disk = 12.087
Queue length@disk = 15.065
Queue length@disk = 18.9509
Queue length@disk = 22.6919
Queue length@disk = 25.6272
Queue length@disk = 27.7555
Queue length@disk = 29.282
Queue length@disk = 30.3924
Queue length@disk = 31.2174


       ---------------------------------
       | Service Time  | Queuing Time  |
       ---------------------------------
       |     1.0000    |    19.4412    |
       |     4.0000    |    91.5901    |
       |     9.0000    |   256.6337    |
       |    16.0000    |   575.6065    |
       |    25.0000    |  1082.7795    |
       |    36.0000    |  1769.3383    |
       |    49.0000    |  2617.0880    |
       |    64.0000    |  3614.4978    |
       |    81.0000    |  4755.5845    |
       |   100.0000    |  6037.2424    |
       ---------------------------------
```

XSnap V1.0 Model name: sample Page:9
Sun Sep 20 16:56:42 1992

Service Time	Waiting Time
1.0000	10.2086
4.0000	47.6412
9.0000	132.4616
16.0000	296.3471
25.0000	557.6317
36.0000	911.5831
49.0000	1348.4848
64.0000	1862.3007
81.0000	2449.9586
100.0000	3109.8865

Appendix B

Sample output generated using "CMTB Text"

This appendix gives sample output generated using the "CMTB Text" option under the "File" menu in XSnap.

```
XSnap V1.0          Model name: sample       Page:1
Sun Sep 20 16:56:04 1992

Params:  Workload details
Workload name  :interb

Classes : active
          query
          update

Closed Chain                Average mpl :25

MPL 25#1
```

```
XSnap V1.0            Model name: sample       Page:2
Sun Sep 20 16:56:04 1992

Params:  Workload details

Workload name  :batcha
Classes : trans
          io_wait

Closed Chain                Average mpl :10

MPL 10#1
```

```
XSnap V1.0            Model name: sample       Page:3
Sun Sep 20 16:56:04 1992

Params:  Centre details

Centre name  :cpu

Service Details:
Processor sharing
   time batcha:trans   0.2
   time interb:active   0.1
```

```
XSnap V1.0           Model name: sample       Page:4
Sun Sep 20 16:56:04 1992

Params:  Centre details
Centre name  :disk

Service Details:
FCFS Centre     Service time :20
```

```
XSnap V1.0           Model name: sample       Page:5
Sun Sep 20 16:56:04 1992

Params:  Centre details
Centre name  :tty

Service Details:
Delay centre     Timeall :300
```

```
XSnap V1.0           Model name: sample       Page:6
Sun Sep 20 16:56:04 1992

Params:  Route details
Workload name  :interb

from active@cpu to update@disk,
                   query@tty
                   freq 0.35,0.65
from update@disk to active@cpu freq 1
from query@tty to active@cpu freq 1
```

```
XSnap V1.0           Model name: sample       Page:7
Sun Sep 20 16:56:04 1992

Params:  Route details
Workload name  :batcha

from trans@cpu to io_wait@disk freq 1
from io_wait@disk to trans@cpu freq 1
```

Bibliography

[Aho 86] Aho A.V., Sethi R. and Ullman J.D., *Compilers – Principles, Techniques and Tools*, Addison-Wesley Publishing Company, Massachusetts, 1986, ISBN 0-201-10194-7.

[Balbo 92] Balbo G., Serazzi G. (eds), *Computer Performance Evaluation – Modelling Techniques and Tools*, Elsevier 1992

[Baskett 75] Baskett F. et al. "Open, Closed and mixed Networks of Queues with Different Classes of Customers." *Journal of the ACM*, 22, 2, pp 248-260, April 1975.

[Beilner 90] Beilner H., "Structured Modelling and Tool Support", *Performance 1990* Johannesburg, 1990.

[Buzen 73] Buzen J.P., Computational algorithms for closed queueing networks with exponential servers, *Communications of the ACM, 16(9)*, September 1973, pp 527-531

[Chandy 80] Chandy K.M. and Sauer C.H., Computational algorithms for product-form queueing networks, *Communications of the ACM 23*, 10 (October 1980), pp 573-583

[Conte 65] Conte S.D. and De Boor C., *Elementary Numerical Analysis – An algorithmic approach*, McGraw-Hill, New York, 1965.

[Conway 86] Conway A.E. and Georganas N.D. RECAL – a new efficient algorithm for the exact analysis of multiple-chain closed queueing networks. *Journal of the ACM 33*, 4 (October 1986), pp 768-791

[Grassmann 87] Grassmann W., Kumar S. and Billinton R., "A stable algorithm to calculate the steady-state probability and frequency of a Markov system", *IEEE Transactions on Reliability*, **36**, 198, pp. 58-61.

[Hutch 90] Hutchison A., *Solution Methods and Tools for Performance Models*, Computer Science Department, University of Cape Town, CS90-03-01, December 1990.

[King 90] King P.J., *Computer and Communication Systems Performance Modelling*, Prentice Hall, New York, 1990.

[Klein 75] Kleinrock L., *Queueing systems Volume I: Theory.* J.Wiley & Sons, New York, 1975.

[Krit 81] P.S.Kritzinger, S. Van Wyk and A.E.Krezsinski, *A Generalisation of Norton's Theorem for Multiclass Queueing Networks*, Institute for Applied Computer Science, University of Stellenbosch, February 1982.

[Krit 82] P.S.Kritzinger and S.Van Wyk, *MEAN VALUE ANALYSIS, A Collection of the Results*, ITR 82-14-00, April 1982.

[Krit 84] M.Booyens and P.S.Kritzinger, *SNAPL/1: A Langauge to Describe and Evaluate Queuing Network Models*, Printed in Performance Evaluation, Volume 4, No 3, August 1984.

[Lam 83] Lam S.S., A simple derivation of the MVA and LBANC algorithms from the Convolution algorithm, *IEEE Transactions on Computers*, C-32(11):1062-1064, November 1983.

[Laz 84] Lazowska E.D., Zahorjan J., Graham G.S., and Sevcik K.C., *Quantitive System Performance - Computer System Analysis using Queueing Network models*, Prentice-Hall, Englewood Cliffs, N.J., 1984.

[MSnap 90] *MicroSnap User Manual*, 3rd Edition, Department of Computer Science, University of Cape Town, August 1990.

[Niji 78] Nijenhuis A. and Herbert S.W., *Combinatorial Algorithms for Computers and Calculators.*, Academic Press Inc., New York, 1978.

[Pooley 92] Pooley R.J., Hillston J. (eds), *Computer Performance Evaluation '92 - Modelling Techniques and Tools*, Conference Copy 1992

[Puig 89] Puigjaner R., Potier D. (eds), *Modelling Techniques and Tools for Computer Performance Evaluation*, Plenum 1989

[Ralston 65] Ralston A., *A first course in Numerical Analysis*, McGraw-Hill, New York, 1965.

[Recipes 87] W.Press, B.Flannery, S.Teukolsky and W.Vetterling, *Numerical Recipes - The art of Scientific Computing*, Cambridge University Press 1987, ISBN 0 521 30811 9.

[Reiser 81] Reiser M., Mean Value Analysis and Convolution method for queue-dependent servers in closed queueing networks, *Performance Evalaution Vol 1, No 1*, January 1981, pp 7-18.

[Sauer 81] Sauer C.H. and Chandy K.M., *Computer Systems Performance Modelling*, Englewood Cliffs, NJ: Prentice Hall, 1981.

[Schmidt 89] Renate A. Schmidt, *Analysis and comparison of several algorithms for the solution of closed multichain product-form queueing networks*, Department of Computer Science, University of Cape Town, February 1989.

[Sczittnick 90] Sczittnick M. and Muller-Clostermann B., "MACOM - A Tool for the Markovian Analysis of Communication Systems", *Proceedings on the Fourth International Conference on Data Communication Systems and their Performance* Barcelona, 1990.

[Sevcik 79] Sevcik K.C., and Mitrani I., The Distribution of Queueing Network states at input and output instants, *Proceedings of the 4th International Symposium on Modelling and Performance Evaluation of Computer Systems*, Vienna, February 1979.