TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

PET, a performance evaluation tool for flexible modeling and analysis of computer systems

Citation for published version (APA):
Veth, de, R. (1988). *PET, a performance evaluation tool for flexible modeling and analysis of computer systems*. (Memorandum COSOR; Vol. 8823). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/1988

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 08. Feb. 2024

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computing Science

Memorandum COSOR 88-23

PET, a performance evaluation tool for flexible modeling and analysis of computer systems

by

Robbert de Veth

Eindhoven, the Netherlands

October 1988

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computing Science

MASTER'S THESIS

PET, a Performance Evaluation Tool
for flexible modeling and analysis
of computer systems

by
Robbert de Veth

Supervisor	:	dr. ir. J. van der Wal
Advisor	:	dr. W. Z. Venema

August 1988

Table of Contents

1. Introduction 1

1.1 A growing interest in performance evaluation of computer systems 1

1.2 Computer systems and queuing networks 1

1.3 The PET software package 1

2. Queuing network models 3

2.1 Description of a queuing network model 3

2.2 Assumptions and notations 3

2.3 An Example 5

2.4 Hierarchical modeling 6

3. The algorithms 9

3.1 The performance characteristics of interest 9

3.2 Product form networks 9

3.3 Mean Value Analysis (MVA) 10

3.3.1 Approximations for non-product form networks 13

3.3.2 Reducing the complexity of the MVA-algorithm 14

3.4 Row by row analysis 15

3.4.1 Row by row with multi programming 15

4. PET, Performance Evaluation Tool 17

4.1 Purpose of PET 17

4.2 Decomposing models and algorithms 18

4.3 The modules 18

4.4 How to use the PET package 21

4.4.1 Defining the process tree 22

4.4.2 Setting the parameters 23

4.4.3 Computing the results 23

4.4.4 Other facilities 24

5. The VAX-cluster at the E.U.T., a case study 25

5.1 Purpose of this case study 25

5.2 Description of the VAX cluster 25
5.3 The decision support system for the VAX cluster 26
5.4 Using PET to analyze the VAX cluster 27
5.4.1 The process tree for the VAX cluster 27
5.4.2 Setting the parameters of the VAX cluster 28
5.4.3 Computing the results for the VAX cluster 29
5.5 Conclusions of the case study 31

6. PET in more detail 32
6.1 Description of a module 32
6.1.1 The name.cap file 32
6.1.2 The name.c file 36
6.2 Data flow 37
6.2.1 The monitor 37
6.2.2 Communication between the user and a process 38
6.2.3 Communication between the processes 39

7. Summary, conclusions and suggestions 41
7.1 Summary and conclusions 41
7.2 Suggestions for further development 42

Appendix A: Theory 44

1. Definitions and notations for a queuing network 44
1.1 The parameters of a queuing network 44
1.2 Performance characteristics for a queuing network 45

2. Mean Value Analysis 47
2.1 MVA-algorithm 47
2.1.1 First Come First Served 49
2.1.1.1 Client type independent workloads 49
2.1.1.2 Client type dependent workloads 50
2.1.1.3 Non exponential distributed workloads 51
2.1.2 Processor Sharing 52
2.1.3 Infinite Server 53
2.1.4 First Come First Served with Preemptive Resume Priority 53
2.1.4.1 The Shadow Approximation 53
2.1.4.2 The Completion Time Approximation (CTA) 56

2.1.5 Summary for the computation of the sojourn times 59

3. Schweitzer and Schweitzer-FODI 62
3.1 The Schweitzer approximation algorithm 62
3.2 The Schweitzer-FODI algorithm 64

4. Row By Row Analysis 65
4.1 The Row By Row algorithm 65
4.2 Row By Row with multi programming 70

Appendix B: Introductory manual 71

1. Introduction 71
2. Getting started 72
3. Defining the process tree 75
4. Setting the parameters 77
5. Computations 80
6. Making some changes 82
7. Leaving PET, entering Unix 82
8. And finally.. 83

Appendix C: Writing new modules 85

1. Introduction 85
2. Writing the module 85
3. The name.cap file 86
4. The name.d file 87

References 88

Glossary of notations 90

1. Introduction

1.1. A growing interest in performance evaluation of computer systems

Only a few decades ago a whole new era began with the introduction of the computer. It started with big slow computing machines, for special purposes only, but they seemed to get faster and smaller almost every day, and now they can be found not only in nearly all companies, but also at schools, households, etc.
Another interesting development is the connection of computers (and devices) to other computers, thus forming complex computer systems and networks.

Related to all this is the growing interest in the evaluation of the performance of computer systems. In most cases this performance evaluation is a useful tool if decisions are to be made.
If for example the performance of a computer system is getting worse because of a growing number of users, or a change in the users behavior, then it would be appropriate to know in which part of the system the bottleneck can be found, and how the performance of the system can be improved.
In both cases performance evaluation (of the current computer system and some alternative computer systems) can be of great help.

1.2. Computer systems and queuing networks

One way of obtaining the desired information is by modeling the computer system (which on its own can already give some insight) and then investigate the model. A frequently used strategy is to model the system as a queuing network, and then use mathematical analysis to determine the performance of the modeled system. Up till now exact analysis within a reasonable amount of time is only possible for a small class of queuing networks, the so called *product form networks*. The algorithms based on this kind of networks offer results for the system in equilibrium. Some progress has been made in the development of approximation methods and heuristics that can handle a larger class of networks. But there is still a lot of work to be done in this field of research.

1.3. The PET software package

To use the algorithms, they have to be implemented in some kind of software package. Recently an initial implementation of such a package was made by A. Koopman [6]. The package is called PET: Performance Evaluation Tool. Also a short note on the PET package (as well as a detailed description of several algorithms) can be found in Wijbrands[14].
The strength of PET is that it can not only support the performance evaluation of computer systems by practical users (system managers, students, etc) but also the testing of new developed algorithms. That's why a very flexible design for the package has been made.
The starting point of PET is the hierarchical modeling approach. As we shall see later

the model (and the algorithms) can be decomposed in components that can be analyzed separately. The results of this analysis then can be combined to obtain results for larger components, and finally for the whole network model.
The decomposition approach has lead to a package consisting of a set of individual modules. This set can easily be changed or extended to satisfy the needs of the user or developer.
The model can be entered by defining its components. For every component an algorithm (i.e. a module) has to be specified. It is always possible to replace such an algorithm by another, without the need to change the whole model. In this way an environment is created where new algorithms can be tested, added and used in combination with the already existing ones.

In the past few months the PET package is extended and has now reached the stage that it can be used by e.g. students to test if there are any improvements to be made.

To describe the PET package one has to know how to model a computer system as a queuing network, and what algorithms there are available for mathematical analysis of this network. So we will first describe the queuing network model (Chapter 2) and the algorithms (Chapter 3) before we turn our attention to the PET package in Chapter 4. This Chapter is mainly an introduction in how to use the PET package. The way of entering a model and choosing the algorithms will be discussed here.
After that a case study about the performance evaluation of the VAX-cluster at the Eindhoven University of Technology (E.U.T.) will show the advantages and disadvantages of the PET package.
In Chapter 6 we will describe the PET package in more detail. This Chapter is meant for those users who intend to write their own modules, or who are interested in the design of the package and the modules.
Conclusions, suggestions and some remarks can be found in the last Chapter of this thesis. Three appendices are added: A detailed description of the theory for analyzing queuing network models, a tutorial for the PET package, and a short description of how to write a module.

2. Queuing network models

2.1. Description of a queuing network model.

A queuing network consists of a number of stations, where in every station one or more servers wait for clients to arrive. The clients in the network travel from station to station. At each station they offer the server(s) a certain amount of work (the *workload*), and then they wait until the server has carried out the job. Let us consider such a client, arriving at a station. Maybe it is his first visit to the network, or maybe he has just left another station. The client joins the queue of clients already waiting at the station. The order in which the server serves the clients waiting in the queue is defined by the *service discipline* at the station. It is for example possible to serve the clients in order of arrival, in order of priority, simultaneously, etc. If the server has completed the service of a client, this client continues his *route* by leaving the station to join another station or to leave the system. Note that the clients waiting in the queue also include those clients the server is serving at the moment.
For convenience we introduce the following definitions. An *open* client is a client who arrives at the system, visits a number of stations, and then leaves the system. A *closed* client on the other hand is a client who never arrives or leaves, but always stays in the system. An open network is a network with only open clients. Closed and mixed networks are defined in an analogous way. Clients of the same *client type* are clients with stochastically the same routing, priority and workload.

2.2. Assumptions and notations

First we consider the stations in the network. The number of stations will be denoted by M. One can think of many possible service disciplines, but we consider only the following disciplines:

- FCFS. The simplest service discipline is the discipline where the clients are served in order of arrival. This discipline is called First Come First Served (FCFS).
- LCFS. Another (rarely used) discipline is the so called Last Come First Served (LCFS) discipline, where the server is always serving the client who arrived last.
- IS. If there are enough (e.g. infinite) identical servers to serve all clients at the same time, the station acts under an Infinite Server (IS) discipline.
- PS. It is also possible for a server to serve the clients one by one, all during a small amount of time. If the service of a client is not completed by then, the client is placed at the end of the queue to wait for another amount of time. If this amount gets infinitely small, the service discipline is called Processor Sharing (PS).

- PRIOR-PR. Sometimes the client types have different priorities. The clients with the same priority are handled in order of arrival. But if a client of a higher priority enters the queue, the service of the lower priority client is interrupted, and is resumed only if there are no more clients of a higher priority in the queue. This type of discipline is called preemptive resume priority scheduling (PRIOR-PR). Other possibilities are non preemptive resume (a service is completed before another starts) and preemptive repeat (after an interruption the service is started all over again).

Now we consider the clients in the system. It's easiest to distinguish between closed and open client types. The closed client types are numbered from 1 to R and the open client types from $R+1$ to $R+L$.

For the closed clients the population in the system is constant and can be written as a population vector $\underline{K} = (K_1,..,K_R)$, where K_r is the number of closed clients of type r in the system.

For the open clients an arrival process has to be specified. We assume that the open clients of type r arrive at the system according to a Poisson process with parameter λ_r, and join station m with probability p_m^r. Furthermore it is impossible for the open clients to stay in the system forever.

If a client of type r leaves station m, he will join station n with probability $p_{m,n}^r$. The way of jumping in the network defined by these probabilities is called *Markov routing*.

In this thesis we assume that the clients do not change type, although analysis would be possible if they did.

Another parameter of interest is the mean total number of visits to a station. As the closed clients never leave the system this mean number of visits will either be infinite or zero (if the probability of visiting this station equals zero). It is however possible to determine the *relative* visiting frequency $f_{m,r}$, defined as the mean number of visits of a client of type r at station m during a *cycle*. For a client of type r ($r = 1,..,R$) this can be done by solving the following linear equation system

$$f_{m,r} = \sum_{n=1}^{M} f_{n,r}\, p_{n,m}^r \qquad m = 1,..,M \tag{2.1}$$

with the additional constraint (where C is some constant)

$$\sum_{m=1}^{M} f_{m,r} = C \tag{2.2}$$

Note that the mean cycle time depends on the value of C, so by choosing C the relative visiting frequencies and a cycle are defined. Note also that $f_{m,r} / f_{n,r}$ gives us the

mean number of visits of a client of type r to station m per visit to station n.

For an open client of type r ($r = R+1,..,R+L$) the mean number of visits will be finite because such a client always leaves the system after a certain number of visits. It's easy to verify that these visiting frequencies, also denoted by $f_{m,r}$, can be obtained by solving

$$f_{m,r} = \lambda_r p_m^r + \sum_{n=1}^{M} f_{n,r} p_{n,m}^r \qquad m = 1,..,M \tag{2.3}$$

Finally we will discuss the workload a client offers at a station. We assume that the mean and variance of the workload for a client of type r, arriving at station m only depend on the kind of station m and the client type r (although in some networks the model would be closer to reality if the workload should also depend on e.g. the population). The average workload will be denoted by $w_{m,r}$ and the variance of this workload by $\sigma_{m,r}^2$.
It is also possible to introduce a service rate at each station as the rate at which a server can serve its client. We assume that this rate is equal to 1.

The notations introduced in this section can also be found in the glossary of notations at the end of this thesis.

2.3. An Example

A simple example is used to illustrate how a computer system can be modeled.

Consider a computer with some terminals connected to it. The computer consists of a central processing unit (CPU) and two disks. Suppose there are two kinds of jobs: batch jobs and interactive jobs. Batch jobs are started by the computer while interactive jobs are generated at the terminals. If an interactive job is finished the generation of a new job starts at the same terminal. A job 'in' the system will alternately 'visit' the CPU and one of the disks. We assume that a job arriving at a disk has to wait until the jobs waiting in front of him are served. At the CPU a so called *Round Robin* scheduling mechanism is used in order to pay attention to the jobs in the queue in a more fair way. With a Round Robin discipline the first job in the queue is served for a small amount of time, and if that job is not finished after that amount of time it is placed at the end of the queue to wait for a new service.

This system can be modeled in many ways. One obvious way is the following.
We have four stations: CPU, two disks (D_1 and D_2) and a terminal station (T). At the CPU we choose a PS service discipline, although FCFS would also be possible, especially if the amounts of time the CPU dedicates to a job are not to small. The service

discipline at the disks is FCFS and for the terminal station it's IS.
Furthermore there are two client types: the batch jobs and the interactive jobs. If we assume that the generation of a new interactive job start after the completion of the previous job, then there are as many interactive jobs as there are terminals (in use of course). For the batch jobs we assume that there is a limited number of jobs in the system. If such a job is finished another batch job is started immediately, so also the number of batch jobs stays the same. This makes it possible to model the system as a closed queuing network. For the determination of the workload distribution and the visiting frequencies some kind of measuring has to be done.

The queuing network model as described above is depicted in figure 2.1.

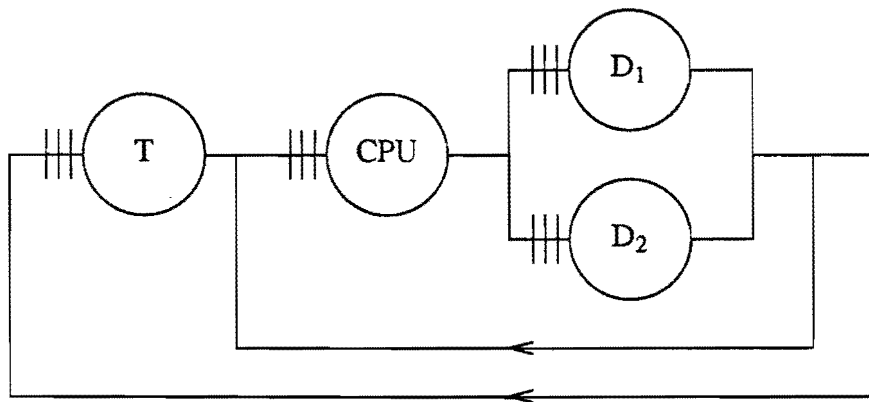

figure 2.1. A simple computer system modeled as a queuing network

2.4. Hierarchical modeling

Decomposing the network in parts which are easy to handle is a commonly used strategy if the system is too complex or too detailed for analyzing. Usually the structure of the network implies already a logical decomposition. An example will illustrate this. It is possible to decompose the network of figure 2.1 in a terminal part and a computer part, where the computer component consists of the CPU and the two disks. The results, obtained by analyzing the computer part can be used as input for the simplified network, where the CPU and the disks are replaced by a single "Comp" station.

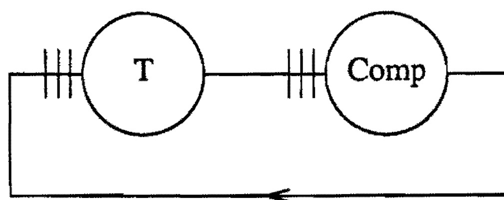

figure 2.2. The network decomposed in a terminal and computer station

The hierarchical structure, obtained in this way, can be represented in a *model tree*.

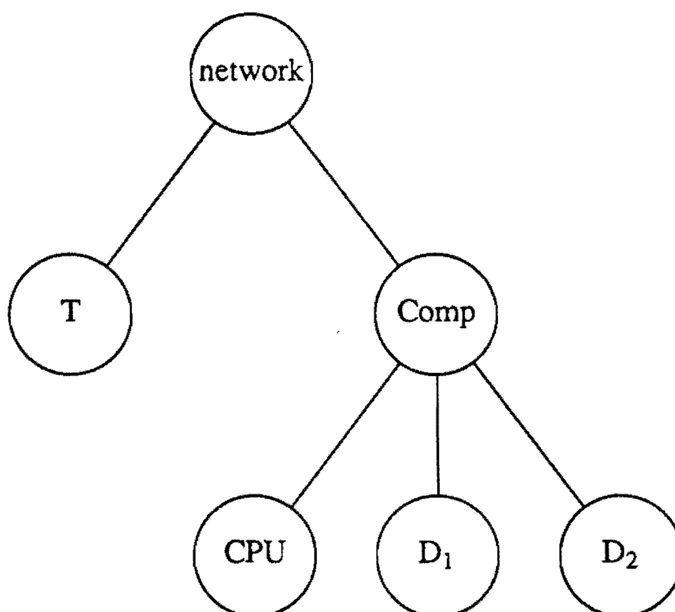

figure 2.3. tree representation of a queuing network model

Of course, it is also possible to analyze the model without decomposing it. In that case the model tree will look like this:

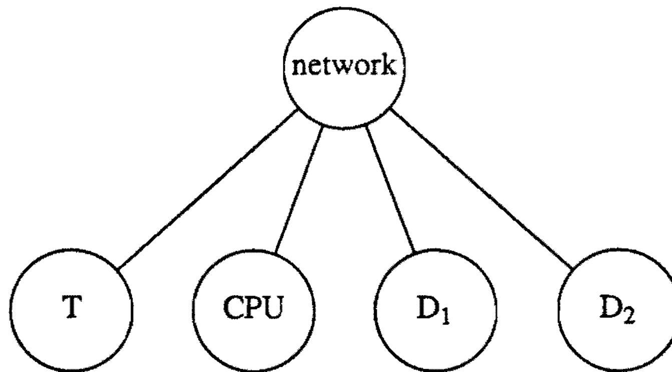

figure 2.4. alternative tree representation of a queuing network model

Decomposition proves to be a useful analyzing tool. It provides better insight in the problem, and permits a hierarchical analysis, from a detailed level up to a global level. We will return to the tree representation of queuing networks when discussing the PET package.

3. The algorithms

3.1. The performance characteristics of interest

After modeling a computer system as a queuing network, it is possible to obtain some performance characteristics by using mathematical analysis. The performance characteristics we will discuss first are the mean values, such as

- The mean number of clients (per client type) waiting in the queue at a station (including the ones being served).
- The mean throughput in the stations per client type, i.e. the mean number of clients that are served per unit of time.
- The mean time a client has to wait at a station, including his service time. This time is called the *sojourn time*.
- The mean utilization of the station per client type. I.e. that part of the time that there are clients (in service) at that particular station.

Usually these mean values will give a good idea about the performance of the system, and they can be used for answering questions like: What response times can be expected, how much time is spent with waiting, what is the bottle neck, etc.
A class of networks for which these mean values are relatively simple to calculate are the product form networks. We will first describe these networks, because most algorithms use the theorems on product form networks as a starting point. After that we will introduce the algorithms.

Of course the mean values do not answer all questions. Other interesting questions one could ask are: What are the variances of the queue lengths and sojourn times? What is the probability that there are more than a certain number of clients in a station? etc. Unfortunately, answering these questions usually turns out to be rather difficult, because it means you have to know something about the distribution of the sojourn times or the queue lengths. We remark that it is possible to determine the steady state distribution by using the theory on Markov chains. In practice however the number of states can be enormously big, even for small models, so usually it will take far too much computation time to calculate this distribution.

3.2. Product form networks

The most important class of networks in queuing theory are the so called *product form networks* or *separable networks*. For these networks the steady state distribution can be written as the product of the state distributions of the independent queues. Baskett, Chandy, Muntz and Palacios [1] have described the *BCMP-networks*. These product form networks are defined as queuing networks satisfying the following restrictions:

- The clients jump according to a Markov routing, with state independent transition probabilities. It is however possible for a client to change type, also in a Markovian way.
- If a station operates with a FCFS discipline the workload must have an exponential distribution, and must be independent of the client type.
  For the other service disciplines (LCFS, IS and PS) the client types may have different workloads. The distribution of these workloads has to have a rational Laplace transform. This last restriction is not a strong one, because every distribution can be approximated arbitrary close by a distribution that has a rational Laplace transform.

The BCMP-networks include closed, open and mixed networks. It is also possible (under certain conditions) to vary the service rate at a station, but since we only consider constant service rates this possibility is not used.

3.3. Mean Value Analysis (MVA)

In queuing theory Mean Value Analysis can be used to obtain some information about the mean values of the system, such as the mean number of clients waiting in a queue, the mean time spent in the network, etc. Especially if the network satisfies the product form conditions it is possible to formulate some interesting relations between the mean values of the network. The algorithm that uses these relations to compute some performance characteristics is called the MVA-algorithm. This algorithm can entirely be expressed in the mean number of clients at a station per client type, the mean throughput at the stations, and the mean time spent in a queue during a visit, also per client type.

Before we formulate the MVA-algorithm for mixed queuing networks we introduce some notations:

$\underline{k}$ population for the closed clients in vector notation, $\underline{k} = (k_1, k_2, .., k_R)$. The maximum population will be denoted by $\underline{K}$.

$\underline{e}_r$ vector denoting a population of a single client of type r.

And the performance characteristics of interest:

$S_{m,r}[\underline{k}]$ mean time a client of type r spends in the queue during a visit at station m given population $\underline{k}$. This time is also called the sojourn time.

$\Lambda_r[\underline{k}]$ mean throughput rate in the network, measured in cycles per unit of time, for a *closed* client of type r, given population $\underline{k}$.

$\Lambda_{m,r}[\underline{k}]$ mean throughput rate at station m for clients of type r, given population $\underline{k}$. For open clients this throughput rate is independent of the population, and will also be written as $\Lambda_{m,r}$.

$N_{m,r}[\underline{k}]$ mean number of clients at station m of type r, given population $\underline{k}$.
$\rho_{m,r}[\underline{k}]$ mean utilization for a client of type r at station m, given population $\underline{k}$.

There are two theorems on which the MVA-algorithm is based. First of all we have Little's Formula [8]. This general applicable theorem gives us a simple relation between the mean number of clients (N) waiting at a station, the mean throughput (Λ) at that station and the mean amount of time (S) a client spends in the queue during a visit

$$N = \Lambda S \tag{3.1}$$

The second theorem (Reiser and Lavenberg [9]) is based on the product form of the network. This *arrival theorem* can be stated as follows

> A closed client of type r arriving at a station observes the system in the steady state distribution with one client of type r removed.

For the open clients the arrival theorem is even easier:

> An open client arriving at a station observes the system as if it is in equilibrium.

The arrival theorem can be used to obtain a relation between the sojourn times of the systems with population $\underline{k}-\underline{e}_r$ and the system with population $\underline{k}$. This implies an algorithm, recursive in the population vector $\underline{k}$.
To compute the performance characteristics for a system with population $\underline{K}$ one has to compute these characteristics for all populations $\underline{k}$ in the range from $\underline{0}$ to $\underline{K}$. This leads to an exact algorithm (for BCMP-networks) for which we will describe how the performance characteristics can be computed in every iteration step. We consider the iteration step for the system with population $\underline{k}$, assuming that we've already performed these steps for the system with populations $\underline{k}-\underline{e}_r$, $r = 1,..,R$.

For all closed client types r ($r = 1,..,R$), and all stations m ($m = 1,..,M$) compute

$$S_{m,r}[\underline{k}] = \begin{cases} \sum_{s=1}^{R+L} N_{m,s}[\underline{k}-\underline{e}_r] w_m + w_m & , m = FCFS \\ \sum_{s=1}^{R+L} N_{m,s}[\underline{k}-\underline{e}_r] w_{m,r} + w_{m,r} & , m = PS \\ w_{m,r} & , m = IS \end{cases} \tag{3.2}$$

$$\Lambda_r[\underline{k}] = \frac{k_r}{\sum_{m=1}^{M} f_{m,r}\, S_{m,r}[\underline{k}]} \tag{3.3}$$

$$N_{m,r}[\underline{k}] = f_{m,r}\, \Lambda_r[\underline{k}]\, S_{m,r}[\underline{k}] \tag{3.4}$$

The first equation is a consequence of the arrival theorem. The last two equations are applications of Little's Formula. For a more detailed description see the appendix on the theory of the MVA-algorithm.

For the open client types the situation is a little different. First of all the mean throughput at the stations is constant, no matter how many closed clients there are in the network. This throughput $\Lambda_{m,r}$ is defined as

$$\Lambda_{m,r} \;=\; \lambda_r\, f_{m,r} \tag{3.5}$$

The arrival theorem for open clients doesn't imply a recursive algorithm, but in combination with Little's Formula it can be used to obtain the performance characteristics for a system with population $\underline{k}$, if these characteristics are known for the closed client types. For the BCMP-networks the computations of the performance characteristics are exact and can be obtained by solving the system of linear equations (3.6) and (3.7)

$$S_{m,r}[\underline{k}] = \begin{cases} \sum_{s=1}^{R+L} N_{m,s}[\underline{k}]\, w_m + w_m & , \; m = FCFS \\ \sum_{s=1}^{R+L} N_{m,s}[\underline{k}]\, w_{m,r} + w_{m,r} & , \; m = PS \\ w_{m,r} & , \; m = IS \end{cases} \tag{3.6}$$

$$N_{m,r}[\underline{k}] = \Lambda_{m,r}\, S_{m,r}[\underline{k}] \tag{3.7}$$

So an iteration step of the MVA-algorithm consists of two parts. First the performance characteristics for the closed clients for the system with population $\underline{k}$ are computed, using the characteristics (for closed and open clients) for the system with populations $\underline{k}-\underline{e}_r,\ r = 1,..,R$.
After that the performance characteristics for the open clients for the system with population $\underline{k}$ can be obtained by solving a system of linear equations, where the

performance characteristics for the closed client types are assumed to be known.

Disadvantages of the MVA-algorithm are the great computational complexity and the strong restrictions on the network for the computations to be exact.

3.3.1. Approximations for non-product form networks

Other workloads at a FCFS stations

If the workloads at a FCFS station depend on the type of client, the computations are no longer exact. The simplest solution is to replace the workload in equation (3.2) by the client type dependent workload. This leads to the following sojourn time for a closed client of type r at such a FCFS station

$$S_{m,r}[\underline{k}] = \sum_{s=1}^{R+L} N_{m,s}[\underline{k}-\underline{e}_r] w_{m,s} + w_{m,r} \tag{3.8}$$

For the open client types an analogous formula can be obtained. This leads to an approximation which is not too bad if the workloads do not differ too much.

It is also possible that the workload for a client at a FCFS station is not distributed according to a negative exponential distribution. In that case the product form conditions aren't satisfied either. Now consider a client of type r arriving at the station. The probability of finding a client of type s in service equals the utilization of this client type s. This utilization $\rho_{m,s}[\underline{k}]$ is given by

$$\rho_{m,s}[\underline{k}] = \Lambda_{m,r}[\underline{k}]\, w_{m,r} \tag{3.9}$$

The average amount of work that still has to be done for the client of type s, at the moment of arrival of the type r client, is called the mean *residual workload* $R_{m,s}$. If the workload would have an exponential distribution the mean residual workload $R_{m,s}$ would equal $w_{m,s}$. This is not the case for a non-exponential distributed workload. It is however possible to use an approximation for the mean residual workload, which should be exact if the client of type r should arrive according to a Poisson process (this is the so called PASTA property, Poisson Arrivals See Time Average). This approximation is

$$R_{m,s} = \frac{\sigma_{m,s}^2 + w_{m,s}^2}{2\, w_{m,s}} \tag{3.10}$$

The formula for the sojourn time then becomes

$$S_{m,r}[\underline{k}] = \sum_{s=1}^{R+L} N_{m,s}[\underline{k}-\underline{e}_r]w_{m,s} + w_{m,r} + \sum_{s=1}^{R+L} \rho_{m,s}[\underline{k}-\underline{e}_r]\left[R_{m,s} - w_{m,s}\right] \tag{3.11}$$

For the open clients an analogous formula can be obtained. Note that we obtain formula (3.8) if the workloads are distributed according to an exponential distribution.

Priority scheduling at a FCFS station

If the station acts under a priority schedule, it is possible to approximate the sojourn times by using the shadow approximation or the completion time approximation. For the shadow approximation the network is transformed into a network where clients do not have to wait for clients of a higher priority. The workloads however are adjusted to slow down the progress of the clients. This transformed network satisfies the product form conditions. This is not the case if the CTA algorithm is used. This last approximation however considers also the clients of a higher priority waiting in the queue if a client arrives. Both approximations are based on a preemptive resume schedule. For the computation of the sojourn times and a more detailed description we refer to the appendix on the theory of the MVA-algorithm.

3.3.2. Reducing the complexity of the MVA-algorithm

A disadvantage of the MVA-algorithm is its computational complexity. For all populations in the range from (0,..,0) to $(K_1,..,K_R)$ an iteration step has to be done. So the total number of iterations is

$$(K_1 + 1)(K_2 + 1) \cdots (K_R + 1) \tag{3.12}$$

Especially for larger populations and more client types this will lead to a lot of computation time.
Schweitzer [10] suggested to replace the recursive algorithm by an iterative approximation that computes the performance characteristics only for the maximum population $\underline{K}$, because usually this is the population of interest. The approximation starts with an initial guess for the performance characteristics of the system with population $\underline{K}$. These characteristics are then improved by iteration. In this iteration the equations of the (last) iteration step of the MVA-algorithm are used. The approximation proves to be fast in relation to the errors (about 5%).

A more accurate approximation method is a First Order Depth Improvement of the Schweitzer algorithm, the Schweitzer-FODI algorithm. Instead of using the

Schweitzer algorithm for the system with population $\underline{K}$, the Schweitzer algorithm is used for the system with populations $\underline{K} - \underline{e}_r$, $r = 1,..,R$. The approximated performance characteristics for these populations then can be used to perform the last step of the MVA-algorithm, thus obtaining the performance characteristics for the system with population $\underline{K}$. The computation time for this algorithm will be larger, because there are R Schweitzer approximations, instead of 1. The errors however are reduced to about 1%.

3.4. Row by row analysis

The row by row analysis can only be used for a *closed* queuing network, consisting of *two* stations. As shown in the example of sections 2.3. and 2.4. a network with two stations can always be obtained by a proper decomposition. So assume we have such a network. In that case it is possible to transform the network so that a client leaving one station always joins the other station. This type of network can be considered as a single queue, where one of the stations takes care of the arrival process, and the other of the service process. By using the theory on single queues analyzing is possible. The (iterative) algorithm based on this analysis is called the row by row algorithm (RBR-algorithm). The name of the algorithm refers to the fact that the client types are considered one by one (or row by row). The algorithm was presented independently by Brandwajn [2] and by Lazowska & Zahorjan [7].

The special form of the network is not the only difference between MVA-algorithm and the RBR-algorithm. For the RBR-algorithm it is possible to use service rates depending on the number of clients in the station. Furthermore the marginal probabilities can be obtained. The marginal probability $p_{m,r}[k]$ is the probability that there are k clients of type r at station m.
Finally we remark that the computations are exact if there is only one client type ($R = 1$), otherwise the results are approximations.
For a more detailed description we refer to the appendix on the theory of the RBR-analysis.

3.4.1. Row by row with multi programming.

The RBR-algorithm can easily be adjusted so that it can be used if the number of clients at one of the two stations is limited. The maximum number of clients allowed at the station is called the *multi programming level*. Clients arriving at a full station have to wait in a buffer.
In computer systems a multi programming level at the CPU is often used by the system manager, to improve the performance of the system. Note that a multi programming level is only useful if the service rate depends on the population at the station, otherwise it makes no difference if you are waiting in the queue or in the buffer.

The RBR-algorithm (with multi programming) is often used in the situation where the stations are in fact aggregated parts of the network. The population dependent service rates then can be obtained by computing the (population dependent) throughput in those parts of the network (this can be done by e.g. the MVA-algorithm).

4. PET, Performance Evaluation Tool

4.1. Purpose of PET

To get some insight in the performance of a computer system, such a system is often modeled as a queuing network, and then analyzed. The PET package is designed to support both the modeling and analysis of the computer system, and it has already proved to be a useful and time saving tool.
This Chapter is mainly an introduction to the PET package. For those who intend to use PET, an introductory manual is added as an appendix, and for those who are interested in the design of PET a more technical description is given in Chapter 6.

The PET package is intended for several situations in which performance evaluation plays a role. Of course it can be used to model and analyze practical situations, but PET is also a useful tool in evaluating newly developed heuristics and approximation methods.

Therefore there will be also several kinds of users. PET can be used by students, so they can learn how to model small computer systems, and what algorithms there are available to analyze a model.
It can also be used by for instance computer system managers, for evaluating the performance of larger computer systems. An example of such a situation is described in Chapter 5, where the VAX-cluster at the E.U.T. is modeled and analyzed.
And finally PET can be used by researchers in a theoretical environment, where it can support the development of new analyzing methods, because these methods can be tested against, and in combination with the already existing ones. In Chapter 6 we will describe how such a newly developed algorithm can be added to the set of algorithms.

Because PET is intended for several kinds of users, it must be easy to learn and to use, but it also has to be flexible and easy to extend. Therefore the design of PET is so that:

- It is easy to model a computer system, by specifying the components of the system. It is also easy to add, delete or replace such components.
- It is easy to choose the algorithms that are used to analyze the model. It has to be possible to replace these algorithms by other algorithms, without the need to change the whole model. And it has to be easy to add new algorithms to the PET package, and to use algorithms in combination with each other.

4.2. Decomposing models and algorithms

The starting point of PET is the hierarchical modeling approach. As described in Chapter 2 it is possible to decompose a network into several "sub networks", the so called *components*. Eventually these components are further decomposed into smaller parts. The smallest component is called a station. Decomposition is a useful modeling tool if the network is too big or complex to analyze it at once. Usually the individual components of the decomposed queuing network are chosen in a way that they are easy to analyze, and the results then can be combined to obtain results for the whole network.

Because every component is analyzed separately there is a strong relation between the way the network is decomposed, and the algorithms that are used to solve the model. In fact one could say that the decomposition of the model also implies a decomposition of the algorithm that analyzes the model. This observation forms the basis of the design of the PET package. In the package each part of an algorithm that can be used to analyze a component of a network is implemented in an individual *module*. These modules are in fact individual programs, and they are only connected to eachother if the user says so, when he defines how the model has to be solved. By implementing PET as a set of individual modules we came to a flexible and easy to understand software package.

4.3. The modules

In Chapter 3 we have discussed some algorithms that can be used to analyze a queuing network model. We will first describe how the algorithms are decomposed, and what modules there are available up till now. After that an example will illustrate how the modules can be combined to solve a model.

First of all there are the algorithms based on the Mean Value Analysis. We introduced the MVA-algorithm, and two approximation methods: the Schweitzer approximation, and the Schweitzer-FODI approximation. These algorithms can solve a queuing network model consisting of one or more stations. For the stations, as well as for the whole network, some parameters have to be specified, such as the workloads (for each station) and the population (in the whole network).
A decomposition of these algorithms suggests itself. All three of the algorithms use the same way of calculating the mean sojourn times for each part (i.e. each station) of the network. Because it depends on the service discipline at a station how these sojourn times are to be calculated, it is obvious that the computation of the sojourn times is implemented in several individual modules. And it depends on the service discipline at a station what module should be used for that station.
The computation of the mean queue lengths and the mean throughput rates however can be implemented in a module for the whole network.

So for each of the three algorithms we implemented a module that computes the mean queue lengths and the mean throughput rates for all stations in the network. For each individual station however the computation of the mean sojourn times is done by a "station level" module. Which module is used depends on the service discipline at that station. one advantage of this approach is that all three algorithms can use the same modules for the computation of the sojourn times. Another advantage is that it becomes very easy to add modules for other service discipline.

Up till now the following modules, based on the Mean Value Analysis, are available:

model component	modules available
network	mva, schweitzer, schweitzer-fodi
FCFS-station	mva-station
PS-station	mva-station
IS-station	mva-station
FCFS-station with non exp. distributed workloads	mva-nonexp
PR-PRIOR-station	mva-prior-cta, mva-prior-shadow

Table 4.1. Available MVA-based modules.

For the algorithms based on the row by row (rbr) analysis the situation is analogous. The two algorithms that we have discussed are the row by row algorithm, and the row by row algorithm with multi programming. For both algorithms the mean service rates for each of the two stations have to be known, so the computation of these service rates can be done in an individual module. This leads to the following modules.

model component	modules available
network	rbr, rbr-multi-prog
FCFS-station	rbr-station
PS-station	rbr-station
IS-station	rbr-station

Table 4.2. Available RBR based modules.

As an example we will use the model of section 2.3 and 2.4., where a computer system consists of a computer with some terminals connected to it. The computer itself is further specified as a CPU with two disks. The terminals are modeled as a single IS-station, the CPU has a PS service discipline, and the disks use a FCFS discipline. The model tree for this network represents the way it is decomposed. In section 2.4. the following decomposition was presented.

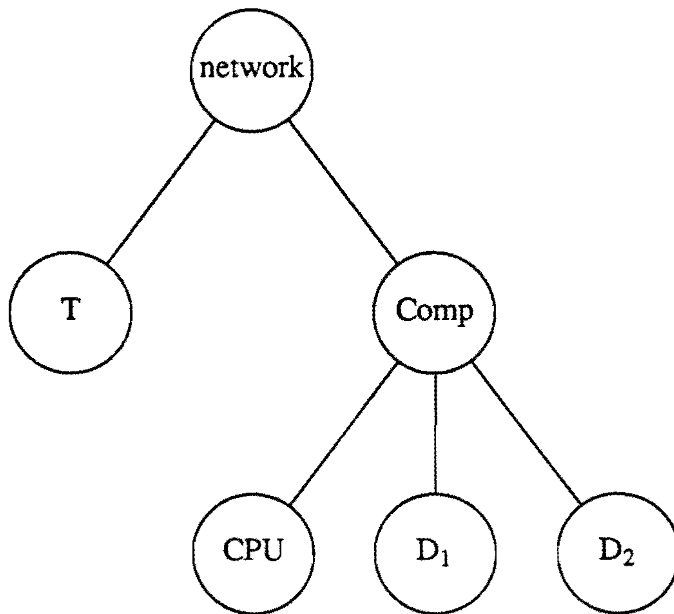

figure 4.1. model tree of a queuing network model

Suppose we want to use the RBR-algorithm to analyze the model. It is possible to use this algorithm because there are only closed clients, and the model consists of two stations (Comp and T). For both stations the service rates (for all populations) have to be available.
One way of obtaining these values for the Comp station is by using the MVA-algorithm to analyze the computer part of the network. The results of this analysis then can be used as input for the row by row algorithm. The algorithm tree for this way of solving the model is depicted in the following figure.

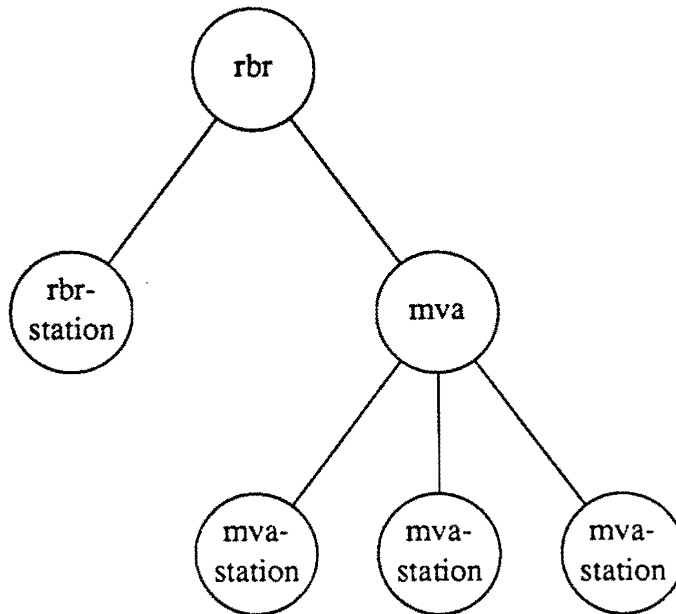

figure 4.2. algorithm tree for the queuing network model

The relation between the model tree and the algorithm tree is obvious. Every node of the model tree corresponds with a node of the algorithm tree. In fact one could say that there is only one tree, defining the model and the algorithms used to solve it. The nodes of this model and algorithm tree are the so called *processes*, and the model and algorithm tree is called the *process tree*.

It is clear that it is easy to replace an algorithm (or part of an algorithm) by another one, simply by replacing a process. One could for instance propose to use a priority station at the CPU (replace mva-station by e.g. mva-prior-cta), or to use a multi programming level at the computer (replace rbr by rbr-multi-prog). Also the addition or deletion of a station is no problem, because it is easy to add or delete a process. Furthermore it is very easy to specify a station in more detail. If for instance the disks are further decomposed, the only thing you have to do is to replace the process for the disk station by a process with some subordinate processes.

How the modeling and analyzing with the PET package is done will be described in the next paragraph.

4.4. How to use the PET package

One can distinguish three phases when using the PET package. These phases are depicted in figure 4.3.

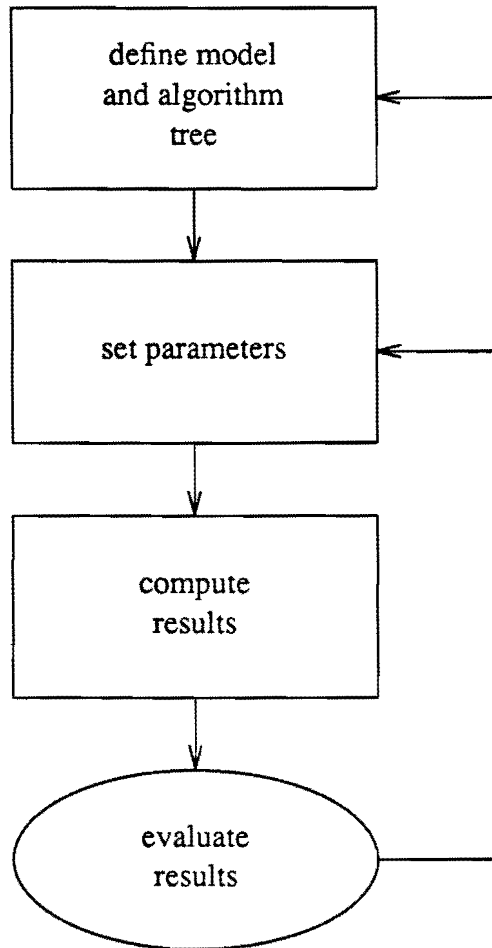

figure 4.3. The three phases when using PET

4.4.1. Defining the process tree

The first phase of defining the process tree (the model and algorithm tree) is very important. In fact this is the phase where the computer system is modeled. Here one has to decide how to decompose the model, and what algorithms one wants to use to solve the model. Of course it is, during the analysis of the model, always possible to change the model and algorithm tree by replacing the processes (i.e. replacing algorithms), by further specifying a station, or by adding or deleting a station (add or delete one or more processes).

A *process* of the model and algorithm tree can be defined by specifying its place in the process tree, its name and the program it uses. First the place of the process has to be given by specifying its *parent*. After that the name of the process is asked. This name refers to the part of the network the process stands for. In figure 4.1 for example, the names as given in the model tree can be used as names for the processes.
Finally the name of (a part of) an algorithm is asked. Here one has the choice between

the modules available in the PET package. They are listed in the tables given in the section about the modules. The program permits only those modules that technically fit at that place of the model tree (by checking the input of the module against the output of the ones connected to it and vice versa). Usually these are the only modules that make sense.

One process is always present, because there has to be a parent available if the user starts defining the process tree. This process is called the *root*, because it forms the root of the model and algorithm tree.
So the parent of the process *network* of the example of figure 4.1 is the *root*.
The processes with the same parent are the so called *slaves* of that parent. So the slaves of the process *network* are *T* and *Comp*.

4.4.2. Setting the parameters

After defining the process tree the parameters of the different processes have to be set. Every process has its own parameters to be set. Usually it is obvious which parameters "belong" to which process. For the process root for instance, the number of clients for the closed client types and the arrival rates for the open client types have to be specified, while at station level for example one has to specify the mean workloads.
The order in which the processes are passed through doesn't matter, because the modules do not "know" yet that they are connected to each other (although the number of slaves usually has to be known).

A special kind of parameters are the *options*. These parameters already have a default value, but they can be reset by the user during the setting of the other parameters. Options never contain parameters of the network model, but they are used for instance to set the way of reporting, the number of iterations, the rate of convergence, etc.

4.4.3. Computing the results

If the model and algorithm tree is defined in a proper way, and if all parameters are set, it is possible to compute some results. For the user this means that he has to give the compute command, and then sit back and wait until the computation is finished. After that he can ask each process to report its results. Usually the results are also written to a file, called processname.report.

The computation uses the structure of the algorithm tree. Remember that the processes (the nodes) of this tree are implemented as individual programs. Such a process is started (i.e starts computing) only if the parent of that process asks the process for results. The computation therefore is as follows. First the process root is started. This process asks his slave for results, so the slave process is started. At the moment that this slave also needs results from *his* slave(s), those processes are started, etc. In this

way the algorithm tree is passed through from the top (root) to the bottom (stations). A more detailed description of the computation is given in Chapter 6, where we take a closer look at the PET package.

4.4.4. Other facilities

In the previous paragraphs we discussed how to model and analyze a computer system with the PET package, by *defining* the model, *setting* the parameters and *computing* the results. Of course one can also *show*, *save* and *print* the input parameters and the results, but that are not the only facilities of the package. It is for instance possible to show the times spend with computing and transporting data. One can also edit the parameters of the model or a process, instead of setting the parameters by answering the questions of the computer. And some help can be obtained if one needs it.
We are also working on a facility that will give some information on the complexity of (a part of) an algorithm, so one doesn't need to do some computations to find out what this complexity will be.

Of course there's still a lot to be done to improve the PET package, but we think we've provided a good basis with enough facilities that can be used as a starting point of a useful software package.

5. The VAX-cluster at the E.U.T., a case study

5.1. Purpose of this case study

A substantial phase in the development of a software package is the testing phase. The first tests for the PET package consisted of only very small problems, because these were the only problems we could check by recalculation of the results with pen and paper. These tests however had some important disadvantages. First of all the models were not realistic, so we didn't know what troubles there would be if a more realistic situation had to be modeled and analyzed.
It was also unknown how fast PET should be if bigger problems had to be solved. Particularly this question was very interesting, since we didn't have any experience with a software package where several programs are running at the same time and where the exchange of data between the programs is done via the system I/O channels.
Therefore we decided to analyze a bigger and more realistic situation to test the PET package.

The problem we have chosen concerns the VAX-cluster at the Eindhoven University of Technology. For this computer network a decision support system, called VAMP is developed that can be used by the system manager to get some insight in the performance of the cluster. The decision support system also uses a queuing network to model the computer system. Therefore this problem is very suitable as a test problem for the PET package, because for this problem it is possible to compare the results and computation time with the results and computation time obtained with the especially for this problem designed decision support system. For a more detailed description of VAMP we refer to the master's theses of De Grient Dreux [3] (in dutch) and Hoogendoorn[4], and a memorandum about this subject [5].

We will first describe the VAX cluster and the decision support system VAMP, before we discuss how the PET package has "passed" the test.

5.2. Description of the VAX cluster

The VAX cluster is a computer network consisting of three VAX computers, nine disk units, and several terminals. It can be modeled as a queuing network in a similar way as the example described in sections 2.3 and 2.4. The model is depicted in figure 5.1.

The terminals are modeled as a single IS-station, at the VAXes there's a priority scheduling (that will be discussed later), and the disks use a FCFS service discipline. The workloads at the disks however do not satisfy an exponential distribution, because during the observation of the disk units it turned out that the variances of the workloads would be too large if we should use exponential distributed workloads. Therefore the variances are taken three times as small as the average workload to obtain a more realistic model.

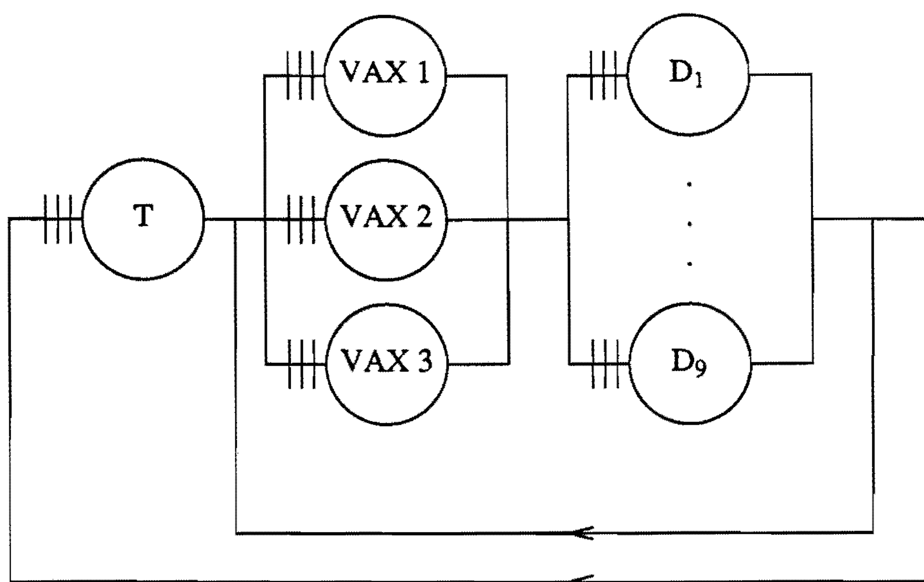

figure 5.1. The VAX cluster modeled as a queuing network

There are two kinds of jobs: the batch jobs and the interactive jobs. The interactive jobs are generated by (the users at) the terminals, while the batch jobs are started by the VAXes. A job is always assigned to one of the VAXes, so after a "visit" to a disk the job will either be finished, or it will "visit" the same VAX as it did before the visit to the disk.
Therefore its easiest to distinguish six different client types. For each VAX we have two client types: the batch jobs and the interactive jobs, where at a VAX station the interactive clients have a higher priority than the batch clients.
A batch job only visits the VAXes and the disks. If such a job is finished, a new job of the same type is immediately generated, so the number of batch jobs stays the same.
If an interactive job is finished, the user who generated the job at one of the terminals starts to generate a new job. The time it takes to generate this job (the thinking time of the user) can be modeled as the time an interactive job spends at the terminal. In that case also the number of interactive jobs is constant. This makes it possible to model the computer system as a closed queuing network.

5.3. The decision support system for the VAX cluster

The decision support system (VAMP) for the VAX cluster is especially made for the analysis of networks, as the one described in the previous paragraph. System managers of a VAX cluster could use it to obtain useful information if decisions are to be made. Therefore VAMP has to be very easy to understand, and it must not bother the user with questions like how to model the system, which algorithm is suitable, etc. So the user interface, taking care of the input as well as the output, is an important part of VAMP.

Another part of VAMP is the part that collects all data, such as the mean workloads, the relative visiting frequencies, the population, etc. Most of these values are obtained by observing the system for a certain period (days, months) and measuring the characteristics of interest (number of users, workloads, etc). Collecting all values that are needed usually turns out to be a difficult and time-consuming job.

If all parameters are available, it is possible to compute some results. As the users of VAMP are likely to be unfamiliar with the algorithms that can be used, it seems favorable to choose an algorithm with a reasonable computation time, and with acceptable errors. Here the Schweitzer-FODI approximation is used for this computation, because the MVA-algorithm for bigger problems usually costs too much computation time, and the accuracy of the ordinary Schweitzer approximation is not good enough.

5.4. Using PET to analyze the VAX cluster

As described in Chapter 4, there are three phases one can distinguish while using PET: defining the process tree, setting the parameters and computing the results. We will shortly discuss the problems that arose during each of these phases. After that some general remarks are made.

5.4.1. The process tree for the VAX cluster

Since the whole model is analyzed with the Schweitzer-FODI algorithm, the network has to be "decomposed" as depicted in the process tree (figure 5.2). For the network the Schweitzer-FODI module has to be used, for the terminal station the mva-station module, for the VAXes the mva-prior-shadow module (although mva-prior-cta is also possible) and for the disks the mva-nonexp module.

Here the first problems of a big model arose.

Although the disks are all identical and use the same program, they have to be modeled as nine different processes, which means that the user has to type in the same values several times. For nine disks however this is still a lot faster than writing your own program.

A more serious problem arose when the computation aborted because there where too many programs running. Actually the number of programs still was allowed, but the number of I/O channels exceeded the maximum number. Fortunately we could overcome this problem by changing some parameters of the operating system that ran the PET package. Still the maximum number of I/O channels, or the maximum number of programs that can run at the same time, is a strong restriction on the size of the problems. Especially because these numbers strongly depend on the machine that is used to run PET, and on the way the operating system on that machine is initialized.

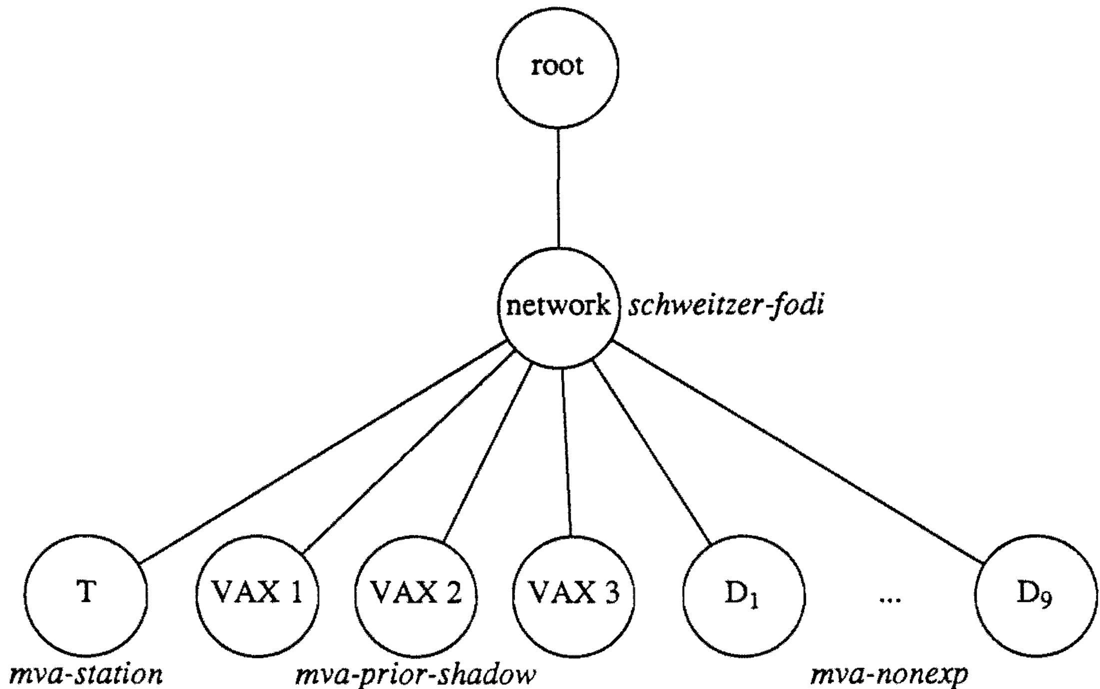

figure 5.2. Process tree of the VAX cluster.

Besides these problems there are also some advantages of using the PET package instead of VAMP. It is for instance very easy to compare different algorithms, like the MVA-algorithm, the Schweitzer algorithm and Schweitzer-FODI, simply by replacing the program schweitzer-fodi of the process network by mva or schweitzer. In this way it is possible to compare the speed and the accuracy of the algorithms and to decide which program is suited best.

It is also no problem to add or delete a disk or a VAX, or even another kind of station. This last possibility is very difficult to realize if VAMP should be used.

5.4.2. Setting the parameters of the VAX cluster

The parameters as collected by VAMP, were essentially the same as the ones we used for our Schweitzer-FODI algorithm. We only had to adjust some values by multiplying them with each other because the formulation of the algorithm used by VAMP was slightly different from the one used by the PET package.
We adjusted and typed in the parameters ourselves, since the number of test problems was too small to consider other ways of entering the input data.
However it should have been possible to transform the input file as generated by VAMP into an input file suitable for PET, because PET uses very simple input files.

5.4.3. Computing the results for the VAX cluster

The computation time of the general applicable PET package was expected to be larger as the computation time of the special for this purpose designed VAMP package. In fact in literature a factor 10 is mentioned to be fully normal. But as we started the computation we had to wait for several minutes before the Schweitzer-FODI algorithm was finished.
If VAMP is used, the results are almost immediately available, so for the VAMP package the computation time is neglectable.
A first cause of this very long computation time can be found in the speed of the machine on which the PET package was running. So it was decided to transport the package from the VAX, on which it was developed to another machine: the SUN. This SUN proved to be about three times as fast as the VAX. Another advantage of transporting the package was the reduced influence of other users.

Still the computation time was too long compared with the computation time of VAMP. To find out where improvements had to be made we measured the time every process spent with computing and the time it was busy doing system calls (mainly I/O time because of the exchange of data). Remember that the PET package has the possibility of showing these times. It turned out that the I/O times were the main reason for the long computation time. Therefore we took a closer look at the data communication. For sending only one series of data an I/O channel had to be used six times. For instance the name of the data had to be send, just as the type of data, the number of records and the data itself. By sending some of these messages as one "aggregated" message, instead of one by one, the number of times an I/O channel had to be opened, could be reduced to three.
An example of the output that is generated if one asks PET for the computation times if given in the following table. Here the process *monitor* is the process that takes care of the exchanging of data between the programs mutually, and between the programs and the user. In fact it is a kind of traffic manager. *User* refers to the time the program is computing, and *sys* refers to the time the system (the SUN) performs system calls (mainly I/O time).

process	6 messages		3 messages	
	user	sys	user	sys
monitor	2.47	23.05	1.55	14.97
root	0.03	0.13	0.03	0.17
network	2.58	13.82	2.27	7.25
vax1	0.23	0.57	0.17	0.53
vax2	0.17	0.55	0.22	0.38
vax3	0.22	0.67	0.17	0.47
disk1	0.18	1.37	0.15	1.07
disk2	0.27	1.28	0.48	1.17
disk3	0.43	1.65	0.42	0.82
disk4	0.27	1.55	0.28	0.88
disk5	0.17	1.47	0.42	0.95
disk6	0.33	1.93	0.13	1.07
disk7	0.35	1.52	0.28	1.08
disk8	0.05	0.15	0.02	0.13
disk9	0.07	0.67	0.07	0.48
term	0.22	1.43	0.23	0.92
total	8.04	51.81	6.89	32.34

Table 5.1. Computation times as reported by the old and by the adjusted PET package.

From the table of computation times we learn that by halving the number of messages the system times (I/O times) are also (nearly) halved. Still these I/O times form the main part of computation times. Furthermore we see that, especially for the smaller values, the computation times are not very accurate, since one should expect the user times almost to be equal for twice solving the problem with the same programs.
However the computation times, as reported by the system, are not very accurate and besides they depend strongly on the number of users, etc. So the differences, as found in table 5.1., are fully normal.

So by two simple actions, namely transporting PET to another machine and adjusting the way of exchanging data between the programs, the computation time could be reduced by about a factor 6.

We expect to improve this computation time even further in the future. It is for example possible to reduce the actual computation time (the user time in the table). This can be done by leaving out some checks that are made during the execution of an algorithm (for instance all vector indices are tested if they are valid). Of course during the development of a new module these tests can be very useful.

We think that it is also possible to reduce the computation time considerably by using the same memory for each module (shared memory). In that case the I/O channels are no longer necessary. This however would involve a major redesign of the way data communication is handled in the PET package.

5.5. Conclusions of the case study

Testing PET with a bigger problem has proved to be a necessary and useful phase of the development of the PET package. First of all we found out that the size of the problem gave some troubles, and that the computation time was far too long. We adjusted PET so that the modeling of larger problems is possible now, but still the size of the model can be a restriction on the possibilities of PET. We also reduced the computation time, but we think that some more improvements can be and have to be made to achieve a more satisfying response time.

Also the user interface of the PET package is not what one should want it to be. Maybe the improvement of this user interface could be the next phase in the development of the PET package.

What's more important is that PET proved to be the tool it was meant for. If VAMP should be still in development, then PET could be used to test several algorithms. If a part of an algorithm was not available, then only that part had to be made, and added to the set of algorithms. In fact we did so ourselves by replacing the mva-nonexp module by a module that calculated the residual workloads by using the special form of the variances of the workloads (see the description of the VAX cluster). But PET could not only be used for the development of the VAMP package. It could also easily replace the part of the VAMP package where the calculations are done (especially if the computation time is further reduced).

6. PET in more detail

In this Chapter a more technical description of the PET package will be presented. It is mainly intended for those readers who want to write their own modules, or who are interested in the way things are arranged in PET.
For those who are only interested in how to use the PET package, the description of PET of Chapter 4 will do.

In the following sections we will describe the contents of a module and the data flow in more detail, but first some general information about PET is given.

We chose to develop PET in a Unix environment, because Unix is an operating system that is well suited for developing new software packages. It also allows more programs (called *processes* in Unix language) to run simultaneously.
Since Unix is written in the programming language C, it is obvious that the support of this language by Unix is on a high level. Therefore we chose to write the modules in C. However, it is also possible to use other programming languages to write one or more modules.

6.1. Description of a module

In a module (a part of) an algorithm is implemented. Although such an algorithm is compiled to an individual program, it has to be possible for other algorithms to use the results of this algorithm. And the algorithm itself may also ask another algorithm for results.
Therefore some things have to be the same for all modules, such as the names of the parameters, the way of communicating with other modules, the names of some functions in the module, etc.
In this section we will describe what a module should look like.

A module consists of two files: the *name.cap* file and the *name.c* file. Here *name* refers to the name of the module, *cap* stands for capability (what results can be computed, etc) and the extension *c* is always used for a source file written in the programming language C. These two files are now discussed in more detail.

6.1.1. The name.cap file

In this file some information about the relation of the module with the outer world is given. This information mainly consists of the parameters the module needs as input, and the results it can deliver. As an example we consider the mva module for which the input parameters are given in the following table. We refer to the glossary of notations (or Chapter 3) for the meaning of the symbols.

from master	*from slave*	*from user*
R, L $(K_1,..,K_R)$ $\lambda_1,..,\lambda_L$	$S_{m,r}[\underline{k}]$	$f_{m,r}$

Table 6.1. Input parameters for the mva module.

The output parameters can be listed in the same way as the input parameters.
The mva.cap file, as given in figure 6.1, contains these input and output parameters also, but now the standard names for the parameters are used. For example nrclosed refers to the number of closed client types R, and visitfreqstattype is the name of the matrix of visiting frequencies $f_{m,r}$, for $m = 1,..,M$ and for $r = 1,..,R+L$.
The names of all parameters are listed in a special file, and each module has to use these names, so that the different programs can communicate with each other.

In this mva.cap file the words fm, tm, fs, ts and fu stand for from master, to master, to slave, from slave and from user. The word ns is an abbreviation for the number of slaves.

mva.cap

fm	nrclosed	fl_complexity|fl_compute
fm	nropen	fl_complexity|fl_compute
fm	popclosed	fl_complexity|fl_compute
fm	arrivalrateopen	fl_complexity|fl_compute
tm	truputpoptype	fl_compute
tm	complexity	fl_complexity
ts	nrclosed	
ts	nropen	
ts	arrivalrateopen	
ts	queuelengthtype	
ts	truputclosed	
ts	visitfreqtype	
ts	clienttype	
fs	responstime,truputtype	
fu	visitfreqstattype	
ns	> 0	

figure 6.1. The mva.cap file.

One has to specify if the parameters are used in the computation (add fl_compute), or if the data are used for complexity calculations (add fl_complexity), or both. If a zero is specified for the to-master data, then these data are assumed to be available without calculation. If two parameter names are separated by a comma then only one of them has to be available.

The data flow as given in the name.cap file can be depicted as follows.

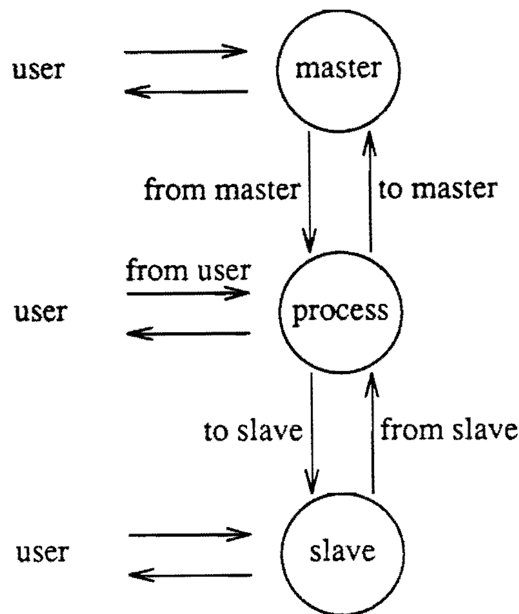

figure 6.2. data flow for a process.

In the name.cap file also the restriction on the number of slaves has to be specified. For the mva module for example the number of slaves (ns) has to be positive (ns > 0).

The name.cap file is used by the PET package to check if the input and output of the process corresponds with the input and output of the master and slave processes. Also the number of slaves can be checked.

The name.cap file is also transformed into a table of messages, called the *name.h* file. This file has to be a part of the source file (the name.c file), because also the program has to know the names of the input and output parameters. This can be done by including the name.h file in the name.c file (see next section). The *h* is an extension that is used for files that are included in source files written in C.
For the modules written in C a special program, called *msgtables,* is available for transforming the name.cap file into the name.h file. One only has to type in

```
msgtables <name.cap >name.h
```

Especially for the builder of a new module the name.cap files of the already existing modules are very important. Because if he wants his module to cooperate with the other modules, then the input and output of data should match. This is easily checked by comparing the name.cap files.

6.1.2. The name.c file

The name.c file consists of the name.h file and some standard functions. Therefore the framework of the name.c file will always look like this.

```
#include name.h

start_up()
{ ... }

ask_user()
{ ... }

show_user()
{ ... }

save()
{ ... }

complexity()
{ ... }

compute()
{ ... }

report()
{ ... }
```

figure 6.3. Framework of the name.c file of a module.

The start_up() function is invoked only once, when the process starts running. Usually this is before the first time the parameters are set. It provides the process with its number of slaves, and it can also be used to initialize and check some things.

The ask_user() function asks the user for the parameters it needs from the user. These parameters can be typed in by the user or they can be written from an input file. These parameters should not be modified during the computation.

Show_user() shows the parameters as given by the user.

Save() has the same output as show_user(), but now the parameters are written to a file. This file can be used as input file for the ask_user() function, simply by typing *<filename*.

Complexity() calculates the complexity of the algorithm.

The compute() function is the most important function. Here the calculation is done for which the module is written. This function may use the functions message(..) and request(..) to send information to its slaves and to ask for results from its slaves. This way of exchanging data will be discussed in the next section.

Finally the report() function can be used to report the results obtained while computing.

Of course the user may add some more functions if he wants to. But the functions listed above always have to be present.
Therefore it is easiest to take an already existing module as a starting point for a new one, because all function names are present and in most cases also the contents of the functions doesn't have to be changed that much.

6.2. Data flow

6.2.1. The monitor

Since each part of an algorithm is implemented in an individual module, the exchange of data has to be done via the I/O-channels of the operating system the PET package runs on. In our case this is the Unix system. A program can write some data to such a channel. These data are placed in a buffer. Another (or the same) program reads the data from the buffer in the order they were written to it (first in first out). The operating system takes care of all this.

One can think of the I/O channels as the arrows as depicted in figure 6.2, but this is not exactly the way these channels are used. To save on the number of channels and to simplify the exchanging of data, all data transport is done via a sort of traffic manager: the *monitor*. So the situation of figure 6.2 can be depicted as:

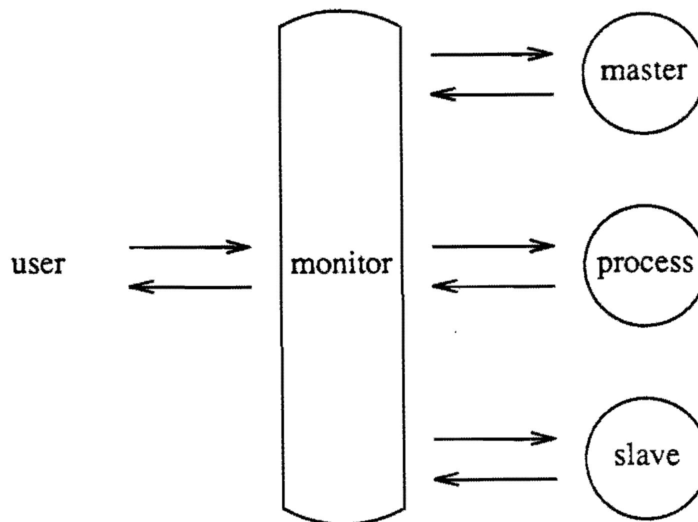

figure 6.4. All data exchange goes via the monitor.

Directing the data flows is only one of the tasks of the monitor. Some other tasks are:

- Let the user build the process tree, and memorize all the information about this tree. Also check if the algorithms in the tree do match.
- Let the user set the parameters, and check if the parameters for all processes are set if the computation starts.
- Start the computation, and take care of the communication of the processes during that computation.
- Supply each process with its number of slaves.

Some examples of how information is exchanged between the user and the processes, and between the processes in the process tree, are discussed in the following two paragraphs.

6.2.2. Communication between the user and a process

If the user defines the process tree, then this information is memorized by the monitor. It is also the monitor who checks if the algorithms fit in the tree by comparing the different name.cap files.
If the user asks to show the model then the monitor will show what the process tree looks like. So in the defining phase it is not necessary for the processes to run.

If the user wants to set the parameters of a process however, he (or she) has to communicate with that process. This is done in the following way. First the user has to tell the monitor for which process he wants to set the parameters. Then the monitor checks if the process is already running. If this is not the case, the process is started,

and the start_up() function is called.
Now that the process is running, it is possible to start the ask_user() function of the process. This function asks the user to type in all parameters that the process needs from the user. It is not possible to set only some of the parameters of the process, because the ask_user() function has to tell the monitor if the setting was successful, so the monitor will know that it is possible to use the show_user(), save() or compute() function of that process.
If the user wants to see the parameters of a process, he first has to specify the name of the process to the monitor, and the monitor then starts the show_user() function of that process. The saving and reporting is done in a similar way.

In the computation phase the processes have to communicate with each other, instead of communicate with the user. In the next section this type of communication will be discussed.

6.2.3. Communication between the processes

If the user tells the monitor to start the computation, then the monitor starts the compute() function of the root process. This root process will tell its slaves (via the monitor of course) when they have to start their part of the computation. These slaves can do the same thing with *their* slaves, etc. So by starting the computation at the root, the whole model will be solved, because each process can tell each of its slaves to start computing the results for the part of the network such a slave represents.
Because of the special structure of the process tree (each process has only one parent, and cycles are impossible), and because of the fact that a process can only ask results from its slaves, processes will never wait for eachother or something like that. So if the compute() functions are implemented correctly, the model will always be solved.

There are two functions a process can use to communicate with its slaves. These functions can only appear in the compute() part of the module. They are the message(..) function and the request(..) function. The message(..) function is used to give the slave process the data it needs for his computation, and the request(..) function is used to ask the slave for some kind of data. If these data are not available, and if the module has all the parameter values to compute the requested data, then the compute() function of the slave process is started to calculate the requested data. Note that the request(..) function is the only function that can start a compute() function (besides the monitor, who can start the compute() function of the root).
The use of the message(..) and request(..) functions during a computation are depicted in figure 6.5.

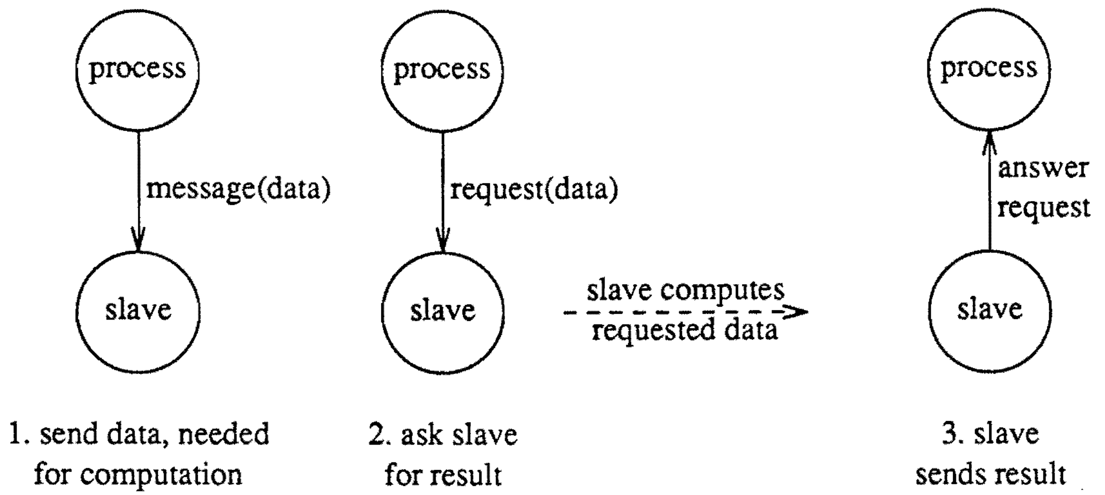

figure 6.5. Asking a slave for information during a computation.

Note that the dashed arrow of figure 6.5 indicates the time that slave process is computing. Also note that the figure suggests that a process is "talking" directly to its slave. This however is always done via the monitor.

7. Summary, conclusions and suggestions

7.1. Summary and conclusions

In the first Chapter of this thesis PET (Performance Evaluation Tool) is introduced as a software package that can be used to obtain some information about the performance of a computer system.
The package is designed to support both the modeling of practical situations as well as the evaluation of new approximation methods.

To model a computer system PET uses queuing network models. One way of investigating such a model is by decomposing it into smaller components that can be analyzed separately. The results of this analysis then can be combined to obtain results for larger components, and finally for the whole network.
The decomposition of the model also implies a decomposition of the algorithm, with components (implemented in modules) that can analyze a part of the network.
This hierarchical modeling and analyzing approach forms the starting point of the design of the PET package.

The basic characteristic of PET is that it consists of a library of individual modules. Each module can be seen as an algorithm component. So the user may build his own algorithm by using these components. He may even write his own components.

An initial implementation of the PET package was developed by Koopman [6] . We continued this development by adding some new modules to the library and by adjusting the other modules, so that also open client types are allowed in the queuing network model.
Furthermore we added the name.cap file to each module. These files can be used if information about the data flow has to be available.

Of course we also tested the PET package. One of these tests concerned the VAX cluster at the Eindhoven University of Technology. This test was particularly interesting because it was the first time we had to solve a big and realistic model. There appeared to be some problems, especially about the computation time, but the PET package also offered some possibilities that facilitated the modeling and analyzing of the problem. The results of this test are given in Chapter 5, where it is concluded that PET proves to be the package it is designed for.

From the other chapters the second Chapter is used to describe the queuing network model, and how it can be decomposed, and in Chapter 3 some algorithms are presented. A more detailed description of the theory for these algorithms can be found in Appendix A.

Chapter 4 is where the PET package is described. This section is mainly a global

introduction to PET, and it gives some information about how the package can be used. A more technical description, for those users who want to build their own modules, is given in Chapter 6. And an introductory manual is added as an appendix.

We note that the use of PET is not restricted to the performance evaluation of computer systems only. It is also very easy to analyze other situations that can be modeled as a queuing network. It is even possible analyze problems that have a tree structured model, and that can be analyzed in an hierarchical way, by creating a new module library for such problems.

Of course there is still a lot to be done, but we think we have provided a useful, flexible and time saving tool, that will help several kinds of users to solve various kinds of problems.

7.2. Suggestions for further development

The library of modules is still very modest. It is for example only possible to build process trees up to three levels in height. Therefore this library should be extended. Here we can think of *aggregation/disaggregation* methods, or a module that represents a number of identical stations, etc. Maybe it is also possible to build a simulation module, so that the results of a mathematical analysis can be compared with the results of a simulation.

As we noted in the analysis of the VAX cluster (Chapter 5) the performance of PET is not yet what one should want it to be. There are several suggestions to improve this performance. One possibility is to use *shared memory* (all modules use the same memory). In that case the I/O channels are no longer necessary.
Another suggestion is to let the processes communicate directly with each other instead of via the monitor. This would halve the I/O time, but a lot of complex arrangements have to be made to realize this kind of communication.
One could also consider to let a process ask for results from one of its slaves, even before it has received an answer from another slave. In that way the processes can do their computations simultaneously. We presume however that this will only improve the performance of PET if the computations are done on different machines.

Especially the user interface needs some improvement. For example it would be very nice if the process tree could be shown in a graphical way. Maybe it is even possible to let the user define a queuing network by "drawing" it, without the need to know that there is also a tree representation of the model.
Also a the setting of the parameters should be improved. Up till now it is not possible to set only some of the parameters of a module (or you have to edit a file that can be used as input file). It is also unpleasant that the modules know nothing about the parameters of the other modules. Especially the parameters that define the size of the

model (and thus the number of other parameters) should be available for the other modules.

Our last suggestion is about the output of the modules. Up till now the results are reported per module, and one has only little influence in the way the output is generated. In some cases however one would like to be able to combine results of one or more processes to obtain new results. Therefore we think of some kind of report writer, that will read its information from the processes, and that will report the things, like tables and maybe even graphics, the user wants to see.

Appendix A: Theory

1. Definitions and notations for a queuing network

We will first summarize the parameters that are used to describe a queuing network, and the performance characteristics we are interested in.
After that the Mean Value Analysis and the Row By Row analysis are discussed.

1.1. The parameters of a queuing network

In Chapter 2 we have introduced the parameters that describe a queuing network model. We will first summarize these parameters.

The number of stations will be denoted by M. The service discipline at a station can be one of the following: First Come First Served (FCFS), Processor Sharing (PS), Infinite Server (IS) or Preemptive Resume Priority scheduling.
The amount of work a client of type r offers at station m, the *workload*, will be denoted by $w_{m,r}$ (or by w_m, if the workload does not depend on the client type).

For the closed clients we have

R	The number of closed client types.
$(K_1,..,K_R)$	The maximum population in the network in vector notation, where K_r denotes the number of closed clients of type r. This vector is also denoted as $\underline{K}$.
$f_{m,r}$	The mean number of visits to station m during a cycle for a client of type r ($r = 1,..,R$). This number is also called the *relative visiting frequency*, since the magnitude of the visiting frequencies depends on the choice of the cycle.

And for the open clients

L	The number of open client types.
$\lambda_1,..,\lambda_L$	The arrival rates for the open client types
$f_{m,r}$	The mean total number of visits to station m for an open client of type r ($r = R+1,..,R+L$) during the time this client is in the network.

Maybe the visiting frequencies are known, but it is also possible to calculate them if the transition probabilities are known. These probabilities are defined as

p_m^r The probability that an *open* client of type r joins station m when he arrives at the network.

$p_{m,n}^r$ The probability that a client of type r joins station n after leaving station m.

To obtain the visiting frequencies for a closed client of type r we have to solve the following system of linear equations

$$f_{m,r} = \sum_{n=1}^{M} f_{n,r} \, p_{n,m}^r \qquad m = 1,..,M \tag{A.1}$$

with the additional constraint (where C is some constant)

$$\sum_{m=1}^{M} f_{m,r} = C \tag{A.2}$$

By choosing C the visiting frequencies and the cycle are defined.
For an open client of type r, the following system has to be solved

$$f_{m,r} = \lambda_r p_m^r + \sum_{n=1}^{M} f_{n,r} \, p_{n,m}^r \qquad m = 1,..,M \tag{A.3}$$

1.2. Performance characteristics for a queuing network

In Chapter 3 we introduced some performance characteristics. The characteristics we will use in the algorithms to be discussed are

$S_{m,r}[\underline{k}]$ mean time a client of type r spends in the queue during a visit at station m, given population $\underline{k}$. This time is also called the *sojourn time*.

$\Lambda_r[\underline{k}]$ mean throughput rate in the network, measured in cycles per unit of time, for a *closed* client of type r, given population $\underline{k}$.

$\Lambda_{m,r}[\underline{k}]$ mean throughput rate at station m for clients of type r, given population $\underline{k}$. For open clients ($r = R+1,..,R+L$) this throughput rate is independent of the population, and will also be written as $\Lambda_{m,r}$.

$N_{m,r}[\underline{k}]$ mean number of clients at station m of type r, given population $\underline{k}$.

$\rho_{m,r}[\underline{k}]$ mean utilization for a client of type r at station m, given population $\underline{k}$. The mean utilization can be calculated as the product of the mean throughput rate and the average workload. Again, for the open client types ($r = R+1,..,R+L$) this utilization is population independent, and will also be written as $\rho_{m,r}$.

For the open client types the mean throughput rate $\Lambda_{m,r}$ and the utilization $\rho_{m,r}$ can immediately be obtained

$$\Lambda_{m,r} = \lambda_r f_{m,r} \tag{A.4}$$

$$\rho_{m,r} = \Lambda_{m,r} w_{m,r} \tag{A.5}$$

This means that it is possible to detect if the system can handle the open clients even before the computation of the performance characteristics is started. We denote the total utilization of open clients at station m by ρ_m, with

$$\rho_m = \sum_{r=R+1}^{R+L} \rho_{m,r} \tag{A.6}$$

This utilization can also be seen as the mean number of open clients *in service* at the station. So the open clients arrive faster than they can be served if this total utilization is greater than or equal to the number of servers at the station.

2. Mean Value Analysis

One way of analyzing a queuing network model is by using Mean Value Analysis (MVA). This analysis gives us some relations between the mean values of the performance characteristics of the network, such as mean queue lengths, mean sojourn times, mean throughputs, etc.

The algorithm that is based on the Mean Value Analysis is called the MVA-algorithm. In this section we will shortly discuss some theoretical aspects of this algorithm.

2.1. MVA-algorithm

The MVA-algorithm is based on two theorems. First we have Little's Formula (Little [8]), which gives us a relation between the mean number of clients (N), the mean sojourn time (S) and the mean throughput (Λ)

$$N = \Lambda S \tag{A.7}$$

Secondly we have the arrival theorem (Reiser and Lavenberg [9]) for BCMP-networks (a class of product form networks, refer Baskett, Chandy, Muntz and Palacios [1]), which we will use in the following form

> In a system with population $\underline{k}$ a *closed* client of type r arriving at a station observes an average number of clients waiting in the queue which equals the mean number of clients at that station in equilibrium for a system with population $\underline{k}-\underline{e}_r$.
> For an open client the average number of clients at an arrival instant equals the mean number of clients at that station in equilibrium for a system with population $\underline{k}$.

The arrival theorem leads to an algorithm, recursive in the population vector $\underline{k}$:

MVA-algorithm

For all populations $\underline{k}$ in the range from $\underline{0}$ to $\underline{K}$, compute the performance characteristics in the following way

For the closed client types

$$S_{m,r}[\underline{k}] = f \text{ (the characteristics of the system with populations } \underline{k}-\underline{e}_r) \tag{A.8}$$

$$\Lambda_r[\underline{k}] = \frac{k_r}{\sum_{m=1}^{M} f_{m,r} S_{m,r}[\underline{k}]} \tag{A.9}$$

$$\Lambda_{m,r}[\underline{k}] = f_{m,r} \Lambda_r[\underline{k}] \tag{A.10}$$

$$N_{m,r}[\underline{k}] = \Lambda_{m,r}[\underline{k}] S_{m,r}[\underline{k}] \tag{A.11}$$

And for the open client types solve for every station the system

$$S_{m,r}[\underline{k}] = f \text{ (the characteristics of the system with population } \underline{k}) \tag{A.12}$$

$$N_{m,r}[\underline{k}] = \Lambda_{m,r} S_{m,r}[\underline{k}] \tag{A.13}$$

Note that the performance characteristics for the closed clients have to be calculated first, because these characteristics are needed in the calculation of the performance characteristics for the open client types.
It is also necessary that the iteration steps for populations $\underline{k}-\underline{e}_r$ $(r = 1,..,R)$ are done before the beginning of the iteration step for population $\underline{k}$. One way of ensuring this is to pass through the populations in a lexicographical order, starting of course with population $\underline{0}$. This and other enumerations are discussed by Wijbrands [13].

We will now discuss the iteration step of the MVA-algorithm as described by (A.8) - (A.13) in more detail.

Equations (A.8) and (A.12) indicate that the mean sojourn time is a function of the other performance characteristics. The computation of this sojourn time is based on the arrival theorem. How the sojourn time can be obtained depends on the service discipline at the station and will be discussed in sections 1.3.1. - 1.3.4. and summarized

in section 1.3.5.

Equation (A.9) is an application of Little's Formula for a closed client type r in the network. The total number of clients in the network equals k_r, the mean throughput rate in the network, measured in cycles is given by $\Lambda_r[\underline{k}]$, and it is easy to verify that the mean cycle time $C_r[\underline{k}]$ can be expressed by

$$C_r[\underline{k}] = \sum_{m=1}^{M} f_{m,r} S_{m,r}[\underline{k}] \tag{A.14}$$

So Little's Formula becomes

$$k_r = \Lambda_{m,r}[\underline{k}] C_r[\underline{k}] \tag{A.15}$$

By combining the last two results we obtain equation (A.9).

Equation (A.10) is obvious and both (A.11) and (A.13) use Little's Formula for a single station.

In each of the following paragraphs we will consider a service discipline. The calculation of the sojourn time for this service discipline will be discussed. For the open clients this means we have to solve a system of linear equations.

2.1.1. First Come First Served

2.1.1.1. Client type independent workloads

The sojourn time of a client of type r, arriving at station m, consists of two parts: the time he has to wait until the clients in front of him are served, and his own service time. If the workload w_m is independent of the client type and has an exponential distribution, then the mean sojourn time for a closed client of type r, $r = 1,..,R$ is given by

$$S_{m,r}[\underline{k}] = w_m + \sum_{s=1}^{R+L} N_{m,s}[\underline{k}-\underline{e}_r] w_m \tag{A.16}$$

For the calculation of the mean sojourn time at station m, for open clients of type r ($r = R+1,..,R+L$) we have to solve the system (consisting of a number of $2L$ equations)

$$\begin{cases} S_{m,r}[\underline{k}] = w_m + \sum_{s=1}^{R+L} N_{m,s}[\underline{k}] w_m \\ N_{m,r}[\underline{k}] = \Lambda_{m,r} S_{m,r}[\underline{k}] \end{cases} \tag{A.17}$$

By substituting the second equation in the first, and observing that the mean sojourn time is equal for all open client types, we can solve the system, leading to

$$S_{m,r}[\underline{k}] = \frac{w_m + \sum_{s=1}^{R} N_{m,s}[\underline{k}] w_m}{1 - \sum_{s=R+1}^{R+L} \Lambda_{m,s} w_m} = \frac{w_m + \sum_{s=1}^{R} N_{m,s}[\underline{k}] w_m}{1 - \rho_m} \tag{A.18}$$

One can think of cases were the workload does depend on the client type and even might have a non exponential distribution. In that case the product form conditions are no longer satisfied, and since the arrival theorem is based on those conditions, we may not use this theorem. It is however possible to approximate the mean sojourn time.

2.1.1.2. Client type dependent workloads

If the workload $w_{m,r}$ does depend on the client type r, but still has an exponential distribution, the approximation for the closed clients is obvious

$$S_{m,r}[\underline{k}] = w_{m,r} + \sum_{s=1}^{R+L} N_{m,s}[\underline{k} - \underline{e}_r] w_{m,s} \tag{A.19}$$

For the open clients the linear equation system is a little more difficult to solve. However by observing that the sojourn time of a type s client can be expressed in terms of the sojourn time of the open type r client, we obtain the following system

$$\begin{cases} S_{m,r}[\underline{k}] = w_{m,r} + \sum_{s=1}^{R+L} N_{m,s}[\underline{k}] w_{m,s} \\ N_{m,s}[\underline{k}] = \Lambda_{m,s} S_{m,s}[\underline{k}] \\ S_{m,s}[\underline{k}] = S_{m,r}[\underline{k}] + w_{m,s} - w_{m,r} \end{cases} \tag{A.20}$$

Note that type s in the second and last equations has to be an open client type. By substituting the last equation in the second, and then the second in the first the mean sojourn time for an open client of type r can be obtained

$$S_{m,r}[\underline{k}] = \frac{w_{m,r} + \sum_{s=1}^{R} N_{m,s}[\underline{k}] w_{m,s} + \sum_{s=R+1}^{R+L} \Lambda_{m,s} w_{m,s}(w_{m,s} - w_{m,r})}{1 - \sum_{s=R+1}^{R+L} \Lambda_{m,s} w_{m,s}} \tag{A.21}$$

If we use (A.5)-(A.6) for the utilization of the open clients then the mean sojourn time for an open client can be rewritten as

$$S_{m,r}[\underline{k}] = w_{m,r} + \frac{\sum_{s=1}^{R} N_{m,s}[\underline{k}] w_{m,s} + \sum_{s=R+1}^{R+L} \rho_{m,s} w_{m,s}}{1 - \rho_m} \tag{A.22}$$

2.1.1.3. Non exponential distributed workloads

The situation becomes even more complicated if the workloads do not have an exponential distribution. In that case an arriving client of type r can find a client of type s in service, with a residual workload $R_{m,s}$ that doesn't have to equal the average workload of that client (for exponential distributions it does however).
Although the arrival theorem is not valid for these workload distributions, the best approximation we can think of is based on another 'arrival' theorem, the Pollaczek-Khintchine formula. This formula states that at a station, where clients arrive according to a Poisson process, the mean residual workload of the client in service upon an arrival instant equals

$$R_{m,r} = \left[\frac{\sigma_{m,r}^2 + w_{m,r}^2}{2\, w_{m,r}} \right] \tag{A.23}$$

Furthermore the probability that a closed client finds a client of type s in service equals the mean utilization $\rho_{m,s}$

$$\rho_{m,s}[\underline{k}-\underline{e}_r] = \Lambda_{m,s}[\underline{k}-\underline{e}_r] w_{m,s} \tag{A.24}$$

So an approximation for the mean sojourn time of a closed client of type r would be

$$S_{m,r}[\underline{k}] = w_{m,r} + \sum_{s=1}^{R+L} N_{m,s}[\underline{k}-\underline{e}_r] w_{m,s} + \sum_{s=1}^{R+L} \rho_{m,s}[\underline{k}-\underline{e}_r](R_{m,s} - w_{m,s}) \tag{A.25}$$

For the open client types we find a system of linear equations analogous to (A.20)

$$\begin{cases} S_{m,r}[\underline{k}] = w_{m,r} + \sum_{s=1}^{R+L} N_{m,s}[\underline{k}] w_{m,s} + \sum_{s=1}^{R+L} \rho_{m,s}[\underline{k}](R_{m,s} - w_{m,s}) \\ N_{m,s}[\underline{k}] = \Lambda_{m,s} S_{m,s}[\underline{k}] \\ S_{m,s}[\underline{k}] = S_{m,r}[\underline{k}] - w_{m,s} + w_{m,r} \end{cases} \tag{A.26}$$

The mean sojourn time can also be obtained by substitution, which leads to

$$S_{m,r}[\underline{k}] = w_{m,r} + \frac{\sum_{s=1}^{R} N_{m,s}[\underline{k}] w_{m,s} + \sum_{s=R+1}^{R+L} \rho_{m,s} w_{m,s} + \sum_{s=1}^{R+L} \rho_{m,s}[\underline{k}](R_{m,s} - w_{m,s})}{1 - \rho_m} \tag{A.27}$$

Note that if the workloads are distributed according to an exponential distribution (then $R_{m,s} = w_{m,s}$), we find the sojourn time given by (A.22). If in addition the workloads do not depend on the client type ($w_{m,r} = w_m$) we obtain equation (A.18).

2.1.2. Processor Sharing

At a PS-station the workload may depend on the client type and may even have a non exponential distribution to satisfy the product form conditions. By again applying the arrival theorem we find the mean sojourn time for the closed client types

$$S_{m,r}[\underline{k}] = \left[1 + \sum_{s=1}^{R+L} N_{m,s}[\underline{k} - \underline{e}_r] \right] w_{m,r} \tag{A.28}$$

For solving the system for an open client of type r we will again add an equation, with the mean sojourn time of an open client of type s expressed in terms of the mean sojourn time of the open client of type r

$$\begin{cases} S_{m,r}[\underline{k}] = \left[1 + \sum_{s=1}^{R+L} N_{m,s}[\underline{k}] \right] w_{m,r} \\ N_{m,s}[\underline{k}] = \Lambda_{m,s} S_{m,s}[\underline{k}] \\ S_{m,s}[\underline{k}] = S_{m,r}[\underline{k}] \dfrac{w_{m,s}}{w_{m,r}} \end{cases} \tag{A.29}$$

By substitution, this results in a mean sojourn time given by

$$S_{m,r}[\underline{k}] = \frac{\left[1 + \sum_{s=1}^{R} N_{m,s}[\underline{k}]\right] w_{m,r}}{1 - \rho_m} \qquad \text{(A.30)}$$

2.1.3. Infinite Server

This is the simplest case, for all client types r we have

$$S_{m,r}[\underline{k}] = w_{m,r} \qquad \text{(A.31)}$$

2.1.4. First Come First Served with Preemptive Resume Priority

At a station with a PRIOR-PR schedule each client type has a priority level. The client type with level 1 has the highest priority, and the type with level N the lowest. The priority level of a type r client will be denoted by $pr(r)$.
At the station client types with the same priority are served in order of arrival. If a client of a higher priority enters the station, the service of the client with the lower priority is interrupted, and resumed only if there are no more clients of a higher priority at the station.
If the network contains a station with this service discipline, the product form conditions are no longer valid, and hence exact analysis is prohibited. The priority scheduling however is an often used service discipline in computer systems, so an approximation of the mean sojourn time is desirable.

Two approximation methods are considered in this section: the *Shadow Approximation* and the *Completion Time Approximation*.

2.1.4.1. The Shadow Approximation

Sevcik [11] proposed to transform the network, so that the product form conditions are valid again (the attentive reader however will notice that this is not the case if client types with the same priority have different workloads). The transformation of the network is done by replacing the priority station by N FCFS-stations (the *shadow* stations), one for each priority level.

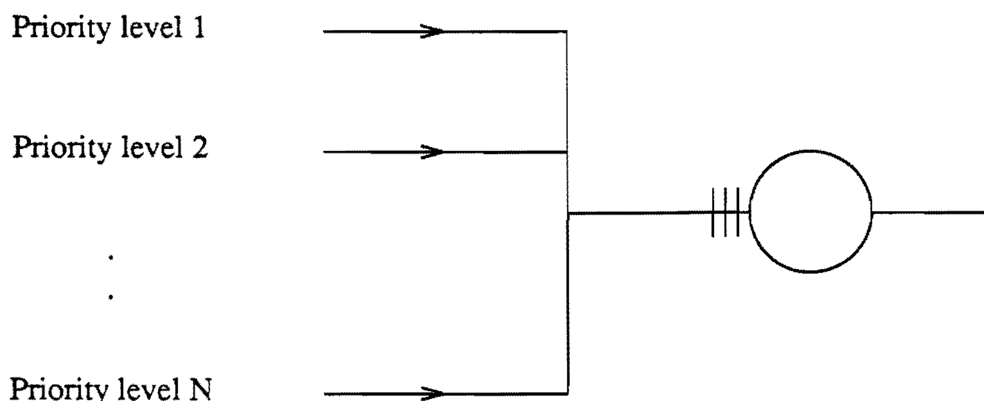

figure A1. A Priority station

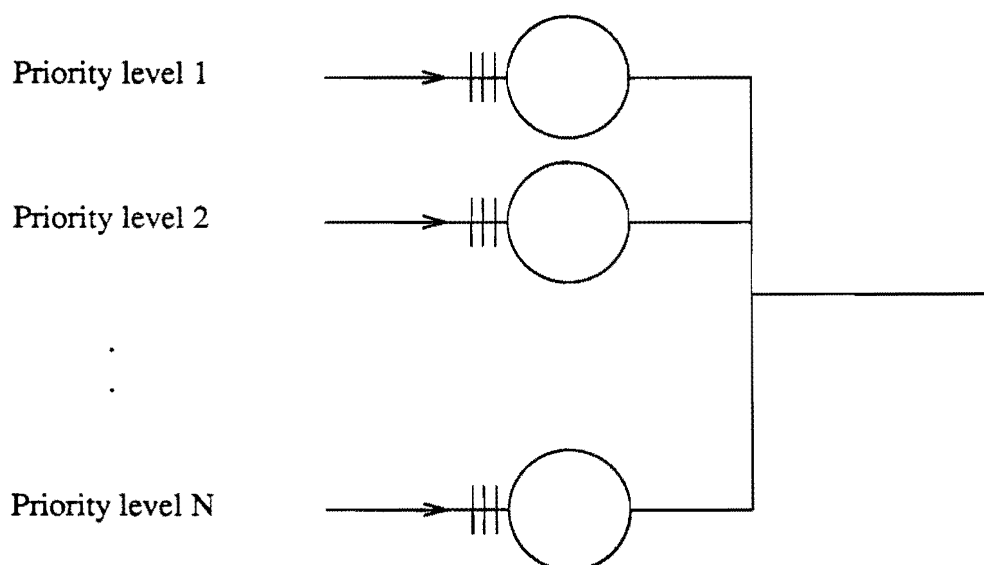

figure A2. The transformed Priority station

In the transformed station clients of priority n do not have to wait for clients of a higher priority, so the sojourn times will be too small, especially for the low priority clients (priority level N, $N-1$,..). To overcome this problem we will slow down the progress of the clients by adjusting the workload. It seems reasonable to do this by dividing the workload by the probability that the station is not utilized by higher priority clients. This leads to an estimation of the *new* average workloads $w^*_{m,r}$

$$w^*_{m,r} = \frac{w_{m,r}}{1 - \sum_{pr(s)<pr(r)} \hat{\rho}_{m,s}} \qquad \text{(A.32)}$$

In the original approach the utilizations $\hat{\rho}_{m,s}$ are unknown, but they can be approximated by the following algorithm:

Original Shadow approximation

(1) Initialize the utilizations $\hat{\rho}_{m,s}$.

(2) Approximate the (new) mean workload of each client type r by

$$w^*_{m,r} = \frac{w_{m,r}}{1 - \sum_{pr(s)<pr(r)} \hat{\rho}_{m,s}}$$

(3) Solve the queuing network model, with the single server preemptive resume priority station replaced by N single server shadow stations as described above.
This will lead to new values for the utilization $\hat{\rho}_{m,s}$.

(4) Usually the utilizations converge. If the desired accuracy is not reached, go to step (2).

Of course solving a whole model (step 3) with maybe bad estimations for the *new* workloads seems not very favorable, so it was proposed to use an iterative or recursive algorithm to solve the model, and adjust the estimations for these *new* workloads in every iteration step.

If the MVA-algorithm is used to solve the model, then it is obvious to choose for the utilization of a client of type r

$$\hat{\rho}_{m,s} = \Lambda_{m,s}[\underline{k}-\underline{e}_r]w_{m,s} \qquad \text{(A.33)}$$

Note that the adjusted workload now becomes population dependent, and therefore has to be evaluated in every iteration step

$$w^*_{m,r}[\underline{k}] = \frac{w_{m,r}}{1 - \sum_{pr(s)<pr(r)} \Lambda_{m,s}[\underline{k}]w_{m,s}} \qquad \text{(A.34)}$$

Now it is possible to determine the mean sojourn time for a closed client of type r

$$S_{m,r}[\underline{k}] = w^*_{m,r}[\underline{k}-\underline{e}_r] + \sum_{pr(s)=pr(r)} N_{m,s}[\underline{k}-\underline{e}_r] w^*_{m,s}[\underline{k}-\underline{e}_r] \tag{A.35}$$

For the open clients we have to solve a system analogous to the ones in the previous sections

$$\begin{cases} S_{m,r}[\underline{k}] = w^*_{m,r}[\underline{k}] + \sum_{pr(s)=pr(r)} N_{m,s}[\underline{k}] w^*_{m,s}[\underline{k}] \\ N_{m,s}[\underline{k}] = \Lambda_{m,s} S_{m,s}[\underline{k}] \\ S_{m,s}[\underline{k}] = S_{m,r}[\underline{k}] - w^*_{m,r}[\underline{k}] + w^*_{m,s}[\underline{k}] \end{cases} \tag{A.36}$$

Where the type s clients of the second equation are open clients, and for the last equation they are open clients with $pr(s) = pr(r)$.

Fortunately, the *new* workloads for the open client types are known if the performance characteristics for the closed clients are available (which they are), because for the open clients the throughput $\Lambda_{m,r}[\underline{k}]$ does not depend on the population $\underline{k}$.

By substitution of the last two equations in the first one, we obtain the following sojourn time

$$S_{m,r}[\underline{k}] = w^*_{m,r}[\underline{k}] + \frac{\sum_{\substack{pr(s)=pr(r) \\ s \text{ closed}}} N_{m,s}[\underline{k}] w^*_{m,s}[\underline{k}] + \sum_{\substack{pr(s)=pr(r) \\ s \text{ open}}} \Lambda_{m,s} w^*_{m,s}[\underline{k}] w^*_{m,s}[\underline{k}]}{1 - \sum_{\substack{pr(s)=pr(r) \\ s \text{ open}}} \Lambda_{m,s} w^*_{m,s}[\underline{k}]} \tag{A.37}$$

2.1.4.2. The Completion Time Approximation (CTA)

A major source of error for the Shadow approximation can be found in the fact that the clients of a higher priority, waiting in the queue at an arrival instant, aren't considered in the model.
Wijbrands [12] suggested to consider also those clients by replacing (A.35), where only client types with the same priority are found in the queue at an arrival instant, by an equation where the clients with a higher priority are also considered

$$S_{m,r}[\underline{k}] = w^*_{m,r}[\underline{k}-\underline{e}_r] + \sum_{pr(s)\leq pr(r)} N_{m,s}[\underline{k}-\underline{e}_r] w^*_{m,s}[\underline{k}-\underline{e}_r] \tag{A.38}$$

We will first explain why the algorithm that uses (A.38), is called the Completion

Time Approximation-algorithm (CTA-algorithm).
The *service completion time* is defined as the time that passes between the moment that a client obtains its first service, and the moment that its service is finished, and the client leaves the station. The completion time of a client of type r consists of two parts: the time that this client is served, and the time he has to wait for clients of a higher priority, that join the station after the service of the type r client started.
The adjusted workload $w^*_{m,r}[\underline{k}]$ can be seen as an approximation for this service completion time for a client of type r in a system with population $\underline{k}$, because it is the solution of

$$w^*_{m,r}[\underline{k}] = w_{m,r} + \sum_{pr(s)<pr(r)} (\Lambda_{m,s}[\underline{k}]w^*_{m,r}[\underline{k}])w_{m,s} \tag{A.39}$$

Note that $\Lambda_{m,s}[\underline{k}]w^*_{m,r}$ is an approximation for the total number of clients of type s that arrive at the station during the completion time of the type r client. The observation that the *new* workloads of the Shadow Approximation and the approximation for the completion time are equal justifies the use of (A.38) and analogous formulas, and therefore (A.38) can be seen as an improvement of the Shadow Approximation.

For the open client types the mean sojourn time can be obtained by solving the following linear equation system

$$\begin{cases} S_{m,r}[\underline{k}] = w^*_{m,r}[\underline{k}] + \sum\limits_{pr(s)\le pr(r)} N_{m,s}[\underline{k}]w^*_{m,s}[\underline{k}] \\ N_{m,s}[\underline{k}] = \Lambda_{m,s}\, S_{m,s}[\underline{k}] \end{cases} \tag{A.40}$$

Unfortunately, the mean sojourn time of an open type s client with $pr(s)\le pr(r)$ can not be expressed in terms of the sojourn time for the type r client (as we did for the other service disciplines). Therefore we have to follow another strategy to obtain the mean sojourn time for an open type r client.
First we define $sum\,1$ and $sum\,2$

$$sum\,1 = \sum_{\substack{pr(s)\le pr(r) \\ s\ closed}} N_{m,s}[\underline{k}]w^*_{m,s}[\underline{k}] \tag{A.41}$$

$$sum\,2 = \sum_{\substack{pr(s)<pr(r) \\ s\ open}} N_{m,s}[\underline{k}]w^*_{m,s}[\underline{k}] \tag{A.42}$$

Note that $sum\,1$ is known, because the performance characteristics for the closed clients are calculated first.
Now we can rewrite the equation system of (A.40)

$$\begin{cases} S_{m,r}[\underline{k}] = w^*_{m,r}[\underline{k}] + sum\,1 + sum\,2 + \sum\limits_{\substack{pr(s)=pr(r) \\ s\ open}} N_{m,s}[\underline{k}] w^*_{m,s}[\underline{k}] \\ N_{m,s}[\underline{k}] = \Lambda_{m,s} S_{m,s}[\underline{k}] \\ S_{m,s}[\underline{k}] = S_{m,r}[\underline{k}] - w^*_{m,r}[\underline{k}] + w^*_{m,s}[\underline{k}] \end{cases} \tag{A.43}$$

As in (A.36) the clients of type s in the last two equations have to be open clients, and for the last equation the clients have to be open with $pr(s)=pr(r)$. By substitution the mean sojourn time can be written as (with $sum\,2$ still unknown)

$$S_{m,r}[\underline{k}] = w^*_{m,r}[\underline{k}] + \frac{sum\,1 + sum\,2 + \sum\limits_{\substack{pr(s)=pr(r) \\ s\ open}} \Lambda_{m,s} w^*_{m,s}[\underline{k}] w^*_{m,s}[\underline{k}]}{1 - \sum\limits_{\substack{pr(s)=pr(r) \\ s\ open}} \Lambda_{m,s} w^*_{m,s}[\underline{k}]} \tag{A.44}$$

We observe that $sum\,2$ is known only if all queue lengths for the open clients of types s with $pr(s)<pr(r)$ are known.
These queue lengths can be calculated in the following way

Calculate for all open client types s with $pr(s)<pr(r)$ *in order of priority* (starting with the open client type of the highest priority)

1 The mean sojourn time $S_{m,s}[\underline{k}]$ by using (A.44).

2 The mean queue length $N_{m,s}[\underline{k}]$ for this client type, by using Little.

Because this is done in order of priority, $sum\,2$ is known if the sojourn time of a client type of the next priority level is calculated.
It is obvious that the sojourn time for the type r clients now is easy to obtain with (A.44), because all queue lengths for the open clients of type s with $pr(s)<pr(r)$ are calculated.

2.1.5. Summary for the computation of the sojourn times

In this section we will repeat the formulas for the computation of the sojourn time. In fact equations A.8 and A.12 of the MVA-algorithm (paragraph 1.3. of this appendix) can be replaced by the equations of this section.

If the service discipline at station m is First Come First Served (FCFS), and the workload does not depend on the client type and has an exponential distribution, then the exact calculation of the sojourn time is as follows

$$S_{m,r}[\underline{k}] = w_m + \sum_{s=1}^{R+L} N_{m,s}[\underline{k}-\underline{e}_r] w_m \qquad r = 1,..,R \tag{A.16}$$

$$S_{m,r}[\underline{k}] = \frac{w_m + \sum_{s=1}^{R} N_{m,s}[\underline{k}] w_m}{1-\rho_m} \qquad r = R+1,..,R+L \tag{A.18}$$

With a FCFS service discipline, and a workload depending on the client type (still with an exponential distribution) the computation of the mean sojourn time is no longer exact.

$$S_{m,r}[\underline{k}] = w_{m,r} + \sum_{s=1}^{R+L} N_{m,s}[\underline{k}-\underline{e}_r] w_{m,s} \qquad r = 1,..,R \tag{A.19}$$

$$S_{m,r}[\underline{k}] = w_{m,r} + \frac{\sum_{s=1}^{R} N_{m,s}[\underline{k}] w_{m,s} + \sum_{s=R+1}^{R+L} \rho_{m,s} w_{m,s}}{1-\rho_m} \qquad r = R+1,..,R+L \tag{A.22}$$

If the service discipline is FCFS, and the workload depends on the client and has not an exponential distribution, the sojourn time can be approximated in the following way.

$$R_{m,r} = \left(\frac{\sigma_{m,r}^2 + w_{m,r}^2}{2\, w_{m,r}} \right) \tag{A.23}$$

$$S_{m,r}[\underline{k}] = w_{m,r} + \sum_{s=1}^{R+L} N_{m,s}[\underline{k}-\underline{e}_r] w_{m,s} + \sum_{s=1}^{R+L} \rho_{m,s}[\underline{k}-\underline{e}_r](R_{m,s} - w_{m,s}) \qquad r = 1,..,R \tag{A.25}$$

$$S_{m,r}[\underline{k}] = w_{m,r} + \frac{\sum_{s=1}^{R} N_{m,s}[\underline{k}] w_{m,s} + \sum_{s=R+1}^{R+L} \rho_{m,s} w_{m,s}}{1 - \rho_m} + \frac{\sum_{s=1}^{R+L} \rho_{m,s}[\underline{k}](R_{m,s} - w_{m,s})}{1 - \rho_m} \qquad r = R+1,..,R+L \tag{A.27}$$

For a Processor Sharing (PS) service discipline at station m, the calculation of the mean sojourn time is exact.

$$S_{m,r}[\underline{k}] = \left(1 + \sum_{s=1}^{R+L} N_{m,s}[\underline{k}-\underline{e}_r]\right) w_{m,r} \qquad r = 1,..,R \tag{A.28}$$

$$S_{m,r}[\underline{k}] = \frac{\sum_{s=1}^{R} N_{m,s}[\underline{k}] w_{m,r} + w_{m,r}}{1 - \rho_m} \qquad r = R+1,..,R+L \tag{A.30}$$

The calculation for the Infinite Server (IS) service discipline at station m is exact and very simple.

$$S_{m,r}[\underline{k}] = w_{m,r} \qquad r = 1,..,R+L \tag{A.31}$$

If station m has a (Preemptive Resume) priority scheduling, then the shadow approximation can be used.

$$w^*_{m,r}[\underline{k}] = \frac{w_{m,r}}{1 - \sum_{pr(s)<pr(r)} \Lambda_{m,s}[\underline{k}] w_{m,s}} \qquad r = 1,..,R+L \tag{A.34}$$

$$S_{m,r}[\underline{k}] = w^*_{m,r}[\underline{k}-\underline{e}_r] + \sum_{pr(s)=pr(r)} N_{m,s}[\underline{k}-\underline{e}_r] w^*_{m,s}[\underline{k}-\underline{e}_r] \quad r = 1,..,R \tag{A.35}$$

$$S_{m,r}[\underline{k}] = w^*_{m,r}[\underline{k}] + \frac{\sum_{\substack{pr(s)=pr(r) \\ s\ closed}} N_{m,s}[\underline{k}] w^*_{m,s}[\underline{k}]}{1 - \sum_{\substack{pr(s)=pr(r) \\ s\ open}} \Lambda_{m,s} w^*_{m,s}[\underline{k}]} + \tag{A.37}$$

$$\frac{\sum_{\substack{pr(s)=pr(r) \\ s\ open}} \Lambda_{m,s} w^*_{m,s}[\underline{k}]\, w^*_{m,s}[\underline{k}]}{1 - \sum_{\substack{pr(s)=pr(r) \\ s\ open}} \Lambda_{m,s} w^*_{m,s}[\underline{k}]} \quad r = R+1,..,R+L$$

Also the Completion Time Approximation can be used for a priority station.

$$S_{m,r}[\underline{k}] = w^*_{m,r}[\underline{k}-\underline{e}_r] + \sum_{pr(s)\leq pr(r)} N_{m,s}[\underline{k}-\underline{e}_r] w^*_{m,s}[\underline{k}-\underline{e}_r] \quad r = 1,..,R \tag{A.38}$$

For the open clients ($r = R+1,..,R+L$) we have to solve the following system of linear equations as described in 1.3.4.2. of this appendix

$$\begin{cases} S_{m,r}[\underline{k}] = w^*_{m,r}[\underline{k}] + \sum_{pr(s)\leq pr(r)} N_{m,s}[\underline{k}] w^*_{m,s}[\underline{k}] \\ N_{m,s}[\underline{k}] = \Lambda_{m,s} S_{m,s}[\underline{k}] \end{cases} \tag{A.40}$$

3. Schweitzer and Schweitzer-FODI

3.1. The Schweitzer approximation algorithm

The Schweitzer approximation algorithm [10] can be seen as an approximate version of the MVA-algorithm (described in the previous paragraph), where the recursion in the population vector is replaced by the iterative solution of only the last recursion step, which is the step for the maximum system population $\underline{K}$.
The Schweitzer algorithm starts with an initial guess for the performance characteristics of the system with population $\underline{K}$ (usually initiating only the mean queue lengths is already sufficient). The choice of these initial values does not have great influence on the iteration process. So it is for instance possible to distribute the clients over all stations, thus initializing the mean queue length at station m for a client of type r by

$$N_{m,r}[\underline{K}] = \frac{K_r}{M} \tag{A.45}$$

Another possibility is to choose

$$N_{m,r}[\underline{K}] = 0 \tag{A.46}$$

After setting the initial values, the iteration starts. For the last iteration step of the MVA-algorithm the performance characteristics for the system with populations $\underline{K}-\underline{e}_r$, $r = 1,..,R$ are needed. In the Schweitzer algorithm these characteristics are approximated by using the characteristics of the system with population $\underline{K}$.

The mean queue length for a client of type s in a system with population $\underline{K}-\underline{e}_r$ is approximated by the mean queue length of that client type in the system with population $\underline{K}$, unless $s = r$. In that case the mean queue length at every station is reduced proportionally with the number of clients. So the approximation for the mean queue lengths is

$$N_{m,s}[\underline{K}-\underline{e}_r] = \begin{cases} N_{m,s}[\underline{K}] & \text{if } r \neq s \\ \left[\dfrac{K_r - 1}{K_r}\right] N_{m,s}[\underline{K}] & \text{if } r = s \end{cases} \tag{A.47}$$

Now that the queue lengths for the system with populations $\underline{K}-\underline{e}_r$ are approximated, the last step of the MVA-algorithm can be performed ($\underline{k} = \underline{K}$), resulting in new approximations for the characteristics of the system with population $\underline{K}$.

If we consider a system with only FCFS stations (for simplicity, extensions are obvious) then the iteration step of the Schweitzer approximation will be

For the closed client types

$$\begin{cases} S_{m,r}[\underline{K}] = w_m + \sum_{\substack{s=1 \\ s \neq r}}^{R+L} N_{m,s}[\underline{K}] w_m + \left[\dfrac{K_r - 1}{K_r} \right] N_{m,r}[\underline{K}] w_m \\ \Lambda_{m,r}[\underline{K}] = \dfrac{f_{m,r} K_r}{\sum_{n=1}^{M} f_{n,r} S_{n,r}[\underline{K}]} \\ N_{m,r}[\underline{K}] = \Lambda_{m,r}[\underline{K}] S_{m,r}[\underline{K}] \end{cases} \qquad \text{(A.48)}$$

For the open client types

$$\begin{cases} S_{m,r}[\underline{K}] = \dfrac{w_m + \sum_{s=1}^{R} N_{m,s}[\underline{K}] w_m}{1 - \rho_m} \\ N_{m,r}[\underline{K}] = \Lambda_{m,r} S_{m,r}[\underline{K}] \end{cases} \qquad \text{(A.49)}$$

The iterative algorithm, described above, usually converges. It can be stopped if the performance characteristics of two subsequent iteration steps are sufficiently close to each other.

In some situations the approximated queue lengths $N_{m,r}[\underline{K}-\underline{e}_r]$ are not the only characteristics, needed for the computation of the mean sojourn time $S_{m,r}[\underline{K}]$ for the closed clients. This is for instance the case at a FCFS-station with a non exponential workload distribution, and at a priority station. In both cases the mean throughput rates $\Lambda_{m,r}[\underline{K}-\underline{e}_r]$ are needed for the computation of the mean sojourn time (refer paragraphs 1.3.1.3. and 1.3.4. of this appendix). A simple approximation that turns out to be rather good is

$$\Lambda_{m,s}[\underline{K}-\underline{e}_r] = \Lambda_{m,s}[\underline{K}] \qquad \text{(A.50)}$$

3.2. The Schweitzer-FODI algorithm

This First Order Depth Improvement of the Schweitzer algorithm can be used to obtain more accurate results. Instead of using the Schweitzer algorithm to calculate the performance characteristics for the system with population $\underline{K}$, Schweitzer is used to calculate the characteristics for the system with populations $\underline{K}-\underline{e}_r$, $r = 1,..,R$. These characteristics then can be used to perform the last step of the MVA-algorithm. The computation time of the Schweitzer-FODI algorithm will be larger as for the Schweitzer algorithm, because there are R Schweitzer approximations instead of 1.

4. Row By Row Analysis

Another way of analyzing a queuing network is by using the RBR analysis. The idea is to transform the network into a single queue. Then the theory on single queues is used to obtain some information about the queuing network. We will first discuss the RBR-algorithm; the algorithm that is based on the ideas of the RBR analysis.

4.1. The Row By Row algorithm

The Row By Row-algorithm (RBR-algorithm) can only be used for a *closed* queuing network, consisting of *two* stations (station 1 and station 2). For each of the two stations the mean service rates are assumed to be known. These service rates however may depend on the population at the station. This makes it possible to use the RBR-algorithm in the following situation.

Consider a queuing network, decomposed in two parts (see e.g. 2.4. of this thesis). Assume that for each of these two parts the throughput rates for all possible populations at that part are calculated (for example by using the MVA-algorithm for that part of the network). Now these throughput rates can be seen as the population dependent service rates for the two (aggregated) stations of the network.
Usually the RBR-algorithm is used in this situation, where a station is an aggregated part of the network, with known throughput rates $\Lambda_r[\underline{k}]$, and therefore the service rate for a client of type r at station m ($m = 1,2$) with population $\underline{k}$ at that station will also be denoted by $\Lambda_{m,r}[\underline{k}]$.

Because there are exactly two stations it is possible to consider the system as a simple birth and death process, where the birth rate depends on the service rate at station 1, and the rate of dying depends on the service rate at station 2.

By using the theory on birth and death processes, it is possible to approximate (by iteration) the marginal probabilities $p_{m,r}[k]$ of k clients of type r at station m ($m = 1,2$).
This can be done by considering the client types individually (row by row).
Other performance characteristics (such as mean queue lengths, mean sojourn times and mean throughputs) can be obtained by using these marginal probabilities.
The algorithm, based on this row by row approach, was presented independently by Brandwajn[2] and by Lazowska & Zahorjan [7].

If we denote the population at station 1 by $\underline{k}_1$, and the population at station 2 by $\underline{k}_2$, then the system population $\underline{K}$ can be written as

$$\underline{K} = \underline{k}_1 + \underline{k}_2 \tag{A.51}$$

And the network, with service rates, can be depicted as in figure A3.

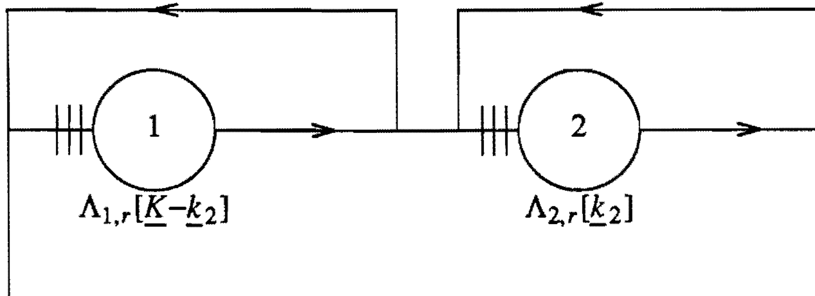

figure A3. Network with population dependent service rates

As shown in this figure it is possible for a client to visit a station more than once before he joins the queue at the other station. Since we are only interested in the transition from a client from one station to another we determine the mean number of visits $f_{m,r}$ for a client of type r to station m before he joins the other station. This number of visits is also called the relative visiting frequency.
(Actually the relative visiting frequency gives us the mean number of visits to a station during a cycle. In this case we've defined a cycle as the visits between two successive transitions from one station to the other. Fortunately, other cycle definitions, and thus other visiting frequencies, do not influence the RBR-algorithm)

If the visiting frequencies are known, it is possible to determine the transition rate from one station to the other. The transition rate from station 1 to station 2 is given by $f_{1,r}^{-1}\Lambda_{1,r}[\underline{K}-\underline{k}_2]$, and the transition rate from station 2 to station 1 is $f_{2,r}^{-1}\Lambda_{2,r}[\underline{k}_2]$.

Now it is possible to transform the network into a single queue, with birth and death rates as shown in figure A4.

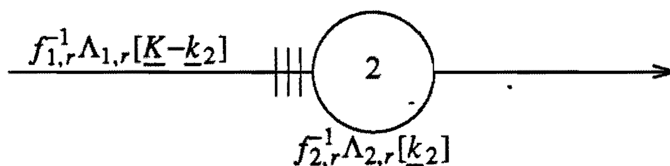

figure A4. A single queue with population dependent birth and death rate

At this stage it would be possible to calculate the (steady state) probabilities of a population $\underline{k}_2$ in station 2 for all populations $\underline{k}_2$ in the range from $\underline{0}$ to $\underline{K}$. This however costs far too much computation time, especially for larger populations and more client types. Therefore the client types are considered one by one (row by row). In

fact for each client type a single queue can be defined

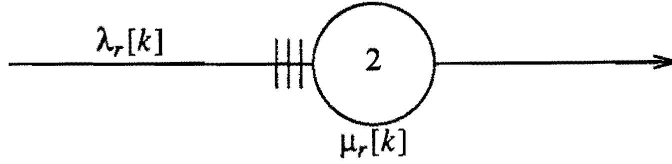

figure A5. A single queue for clients of type r

Here $\lambda_r[k]$ and $\mu_r[k]$ are the birth and death rates for a client of type r, if there are k clients of that type at station 2.

To obtain approximations for $\lambda_r[k]$ and $\mu_r[k]$ we assume that the number of clients of the other types s at station 2 can be represented by their (unknown) averages $N_{2,s}$. Because the population at station 2 has integer components we cut down these averages to integer values $N^*_{2,s}$, so that $N^*_{2,s}$ is the nearest integer smaller than $N_{2,r}$.
The approximation for the birth and death rates then becomes

$$\lambda_r[k] = f_{1,r}^{-1}\, \Lambda_{1,r}[(K_1-N^*_{2,1},..,K_{r-1}-N^*_{2,r-1},K_r-k,K_{r+1}-N^*_{2,r+1},..,k_R-N^*_{2,R})] \quad \text{(A.52)}$$

$$\mu_r[k] = f_{2,r}^{-1}\, \Lambda_{2,r}[(N^*_{2,1},..,N^*_{2,r-1},k,N^*_{2,r+1},..,N^*_{2,R})] \quad \text{(A.53)}$$

The number of clients of type r at station 2, with the corresponding transition rates, are depicted in figure A6.

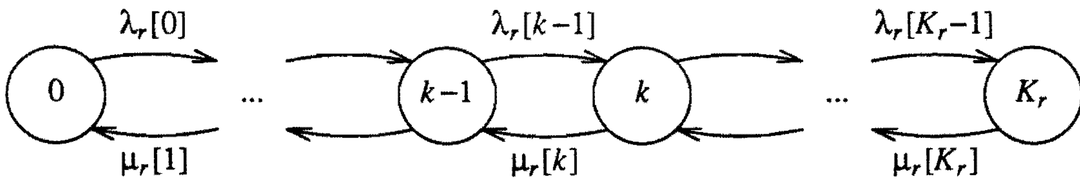

figure A6. Transition rates for a single queue with one client type

By applying the theory on Markov chains we can formulate the following relation:

$$p_{2,r}[k-1]\lambda_r[k-1] = p_{2,r}[k]\mu_r[k] \quad k=1,..,K_r \quad \text{(A.54)}$$

From these relation we obtain the solution for the marginal state distribution:

$$p_{2,r}[k] = p_{2,r}[0] \prod_{j=1}^{k} \frac{\lambda_r[j-1]}{\mu_r[j]} \quad k=1,..,K_r \tag{A.55}$$

$$p_{2,r}[0] = \frac{1}{\sum_{k=0}^{K_r} \prod_{j=1}^{k} \frac{\lambda_r[j-1]}{\mu_r[j]}} \tag{A.56}$$

Now we can compute the mean number of clients of type r at station 2 :

$$N_{1,2} = \sum_{k=0}^{K_r} k\, p_{2,r}[k] \tag{A.57}$$

We will use these adjusted queue lengths in the computation of the marginal probabilities for clients of type $r+1$.

An iterative scheme suggests itself.
At the start of the iteration, we initiate the first approximation of the mean queue lengths as

$$N_{m,r} = \frac{K_r}{2} \quad m = 1,2 \quad r = 1,..,R \tag{A.58}$$

and determine $N^*_{m,r}$. These queue lengths then are improved by iteration, with an iteration step as described by (A.52) - (A.57), and summarized as follows

For all client types r, $r = 1,..,R$ do

(1) Determine the birth and death rates as in (A.52), (A.53).
(2) Calculate the marginal probabilities $p_{2,r}[k]$, $k = 1,..,K_r$.
(3) Use (A.57) to compute the mean queue length $N_{2,r}$.
(4) Determine $N^*_{2,r}$.

The iteration stops if the mean queue lengths $N^*_{2,r}$ do not change during one iteration step.
Generally, the iteration will converge, but we can think of counter examples. Therefore the iteration will be stopped after 50 steps.

At the end of the iteration, the obtained approximations of the mean queue lengths and the marginal probabilities for station 2 can be used to compute some other

performance characteristics:
For each client type r, $r = 1,....,R$, we find :

$$N_{1,r} = K_r - N_{2,r} \tag{A.59}$$

and

$$p_{1,r}[k] = p_{2,r}[K_r - k] \qquad k=0,.....,K_r \tag{A.60}$$

For the mean throughput $\Lambda_{m,r}$ of a type r client in station m, and for the mean system throughput Λ_r we find

$$\Lambda_{1,r} = \sum_{k=0}^{K_r} f_{1,r} \lambda_r[k]\, p_{1,r}[k] \tag{A.61}$$

$$\Lambda_{2,r} = \sum_{k=0}^{K_r} f_{2,r} \mu_{2,r}[k]\, p_{2,r}[k] \tag{A.62}$$

$$\Lambda_r = \frac{\Lambda_{1,r}}{f_{1,r}} = \frac{\Lambda_{2,r}}{f_{2,r}} \tag{A.63}$$

And finally the mean sojourn time can be computed

$$S_{m,r} = \Lambda_{m,r}^{-1} N_{m,r} \tag{A.64}$$

Finally we remark that if there is only one client type, the computations are exact.

4.2. Row By Row with multi programming

The Row By Row-algorithm with multi programming is a special case of the RBR-algorithm as discussed in the previous section. Again only *closed* networks, consisting of *two* stations are allowed. The only difference is that it can handle networks where the number of clients of type r at one of the two stations (say station 2) is limited to a certain *multi programming level* L_r.
If this level is reached, a client arriving at this station has to wait in a buffer. This situation is depicted in figure A7.

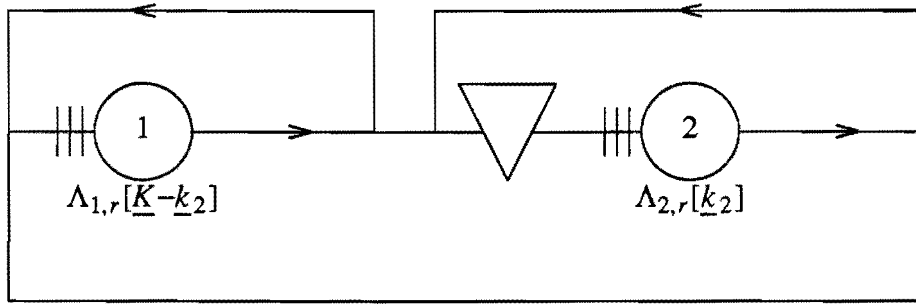

figure A7. Network with a buffer at station 2

As in the previous section the network can be transformed into a single queue. Again we will investigate this queue row by row. Figure A8. gives us such a queue for one row (the row for client type r).

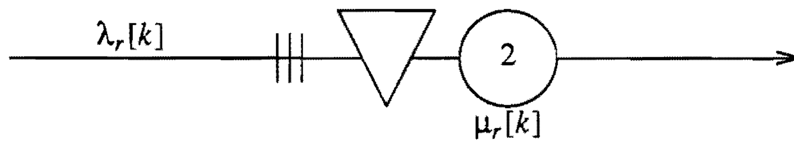

figure A8. A single queue with a buffer for clients of type r

Here $\mu_r[k]$ is the service rate for a client of type r, if there are k clients at station 2, including the clients waiting in the buffer. So the only difference between the ordinary RBR-algorithm and the RBR-algorithm with multi programming is the calculation of this service rate.

$$\mu_r[k] = \begin{cases} f_{2,r}^{-1}\,\Lambda_{2,r}[(N_{2,1}^*,..,N_{2,r-1}^*,k,N_{2,r+1}^*,..,N_{2,R}^*)] &, \ 0 \le k \le L_r \\ f_{2,r}^{-1}\,\Lambda_{2,r}[(N_{2,1}^*,..,N_{2,r-1}^*,L_r,N_{2,r+1}^*,..,N_{2,R}^*)] &, \ L_r \le k \le K_r \end{cases} \tag{A.65}$$

Appendix B: Introductory manual

1. Introduction

The purpose of this document is to get familiar with the PET package. We will do this by solving the following problem:

Figure B1 shows a computer system modeled as a queuing network.

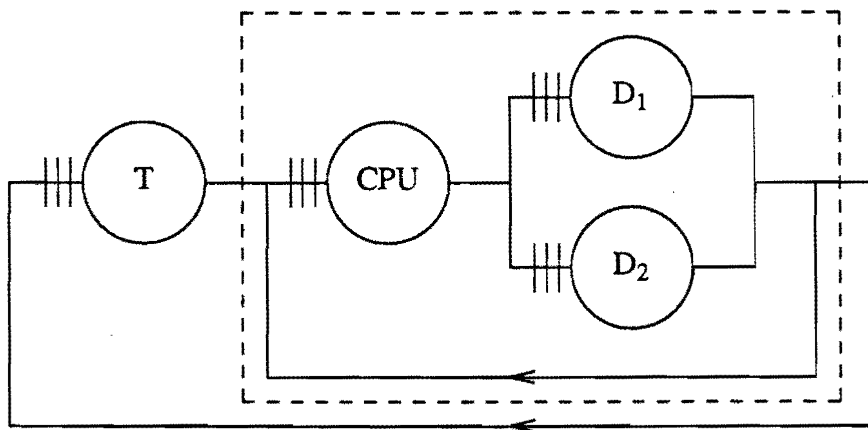

figure B1. A simple computer system modeled as a queuing network

We assume there are two different kinds of jobs. So we have to use two client types in the model. The workloads of these client types are given in the following table.

	CPU	$Disk_1$	$Disk_2$	Terminal
client type 1	5	10	15	5
client type 2	10	10	15	3

Table 1. Workloads per station and client type.

At the CPU there's a priority schedule. If a client of type 1 enters the CPU and a client of type 2 is in service, then the service of this client is interrupted, and resumed only if there are no more clients of type 1 at the CPU.
The disks are modeled as FCFS stations and at the terminal station there is an IS service discipline.
In the closed queuing network there are 4 clients of type 1 and 3 clients of type 2.

After a visit to the CPU, a client of type 1 joins the queue at disk 1 with probability 0.7 and the queue at disk 2 with probability 0.3. For a client of type 2 these probabilities are 0.6 and 0.4.
After leaving a disk a client joins the terminal station with probability 0.2 or he returns to the CPU with probability 0.8.

Suppose we decompose the model into a computer (Comp) part (the dashed box of figure B1) and a terminal (T) part. The tree representation of the decomposed model is shown in figure B2.

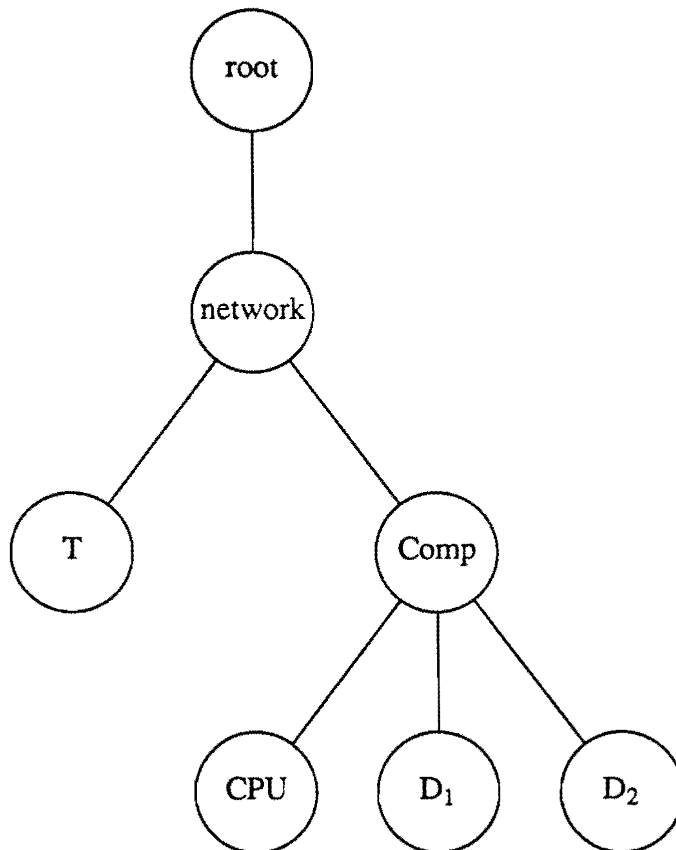

figure B2. tree representation of a queuing network model

We already added the root at the top of the tree, because in the PET package this root is always present. It forms the starting point of the tree.

We will try to solve this problem by using the PET package. We want to use the Row By Row (RBR) algorithm to analyze the computer-terminal model, where the parameters for the computer part are to be obtained with the MVA-algorithm.

2. Getting started

PET is available on the Sun and on the Unix-VAX, both belonging to Eindhoven University of Technology, Department of Mathematics and Computing Science, and both connected to the TUE network and the ETHERNET network.
For the Sun, as well as for the VAX, a user code and a password is required. To be able to work with PET, also some of your global parameters (like your PATH) have to be adjusted. Therefore it seems wise to contact someone who's familiar with the PET

package before you try to run it.

Suppose we are logged in on either the Sun or the VAX. Then PET can be started by typing *monitor* in response to the prompt of the machine (here "$").

```
$ monitor
Type ? or help for more information
monitor>
```

The first line tells us there are several ways to ask for help. This can be done at *any* time the program waits for input. By typing a *?* the program will tell you what kind of input is valid at that place.

```
monitor> ?
    Commands...
    define      exit        set             show
    compute     report      replace         edit
    save        delete      rename          describe

                Abort command with "$", terminate loop with "^"
Type ? or help for more information
monitor>
```

The *help* command gives more detailed information.

```
monitor> help

help

  monitor

   Help can be obtained in various ways:

   By typing a ? in response to a question. The program shows the
   expected type of input.

   By typing the help command. This can be done at any  stage  of
   the conversation, without disrupting the flow of the  program.
   The program will prompt for the name of a subject (topic).
   Type a ? to see which  subjects  are  available. One  or  more
   empty lines cause exit from the help facility.

  Additional information:
  ! $ < ^ compute define  delete  describe  edit  rename  replace
  report save set show

help monitor topic?
```

Now it is possible to return to the program by sending one or more empty lines. Alternatively, we may obtain some additional information by typing one of the topics as listed in the topic list.

```
help monitor topic? define

help

  monitor

    define

     format: define <category>

     Creates an object of the specified category.

    Additional information:
    process

help monitor define topic?
```

3. Defining the process tree

We enter some empty lines (by pressing the return key) to return to the program, and then we start defining the process tree.

```
help monitor define topic?
help monitor topic?
Type ? or help for more information
monitor> define
what: process
parent process name: ?
        Commands...
        root

        Abort command with "$", terminate loop with "^"
```

Fortunately there's already one process present. We will use this process, called *root*, as a starting point of the process tree.

```
parent process name: root
new process name: network
program name: ?
            Commands...
            mva          rbr          rbr-station   rbr-multi-prog
            schweitzer   s-fodi


program name: rbr
monitor>
```

Note that there are several programs that can be connected to the root. We chose the rbr program since we want to solve the computer terminal model with the RBR algorithm.

The first process is defined. Now the building of the rest of the process tree will be a piece of cake. Therefore we will use the defining of some other processes to show that it is possible to enter several commands on the same input line, and that you may use abbreviations.

```
monitor> define
what: process
parent process name: network
new process name: computer
program name: mva
monitor> define process computer cpu mva-prior-cta
monitor> def proc comp disk1 mva-stati
monitor> def proc comp disk2 mva-stati
monitor> def proc net terminal rbr-sta
```

Now that the process tree is defined we may use the *show model* command to see it. The *show* command can also be used to view a single process (*show process*), get some information about the inputs and outputs of a program (*show program*) and to see the amount of computation time each process has used (*show times*).

```
monitor> show
what: model

    parent_name:      process_name:     program_name:

    root              network           rbr
    network           terminal          rbr-station
    network           computer          mva
    computer          cpu               mva-prior-cta
    computer          disk1             mva-station
    computer          disk2             mva-station

process root          (no parameters available)

process network       (no parameters available)

process terminal      (no parameters available)

process computer      (no parameters available)

process cpu           (no parameters available)

process disk1         (no parameters available)

process disk2         (no parameters available)
monitor>
```

Here we observe that a graphical representation of the process tree should be a lot easier to read. Maybe in future this will be possible.

4. Setting the parameters

Since there are no parameters available at all, we have to use the *set* command. Lets start with the root.

```
monitor> set
what: process
process: root
number of closed chains: 2
number of open chains: 0
population for client type 1: 4
population for client type 2: 3
monitor>
```

And now we choose to set the cpu process.

```
monitor> set process cpu
priority class (0 is highest) for client type 1, station cpu: 0
workload for client type 1, station cpu: 5
priority class (0 is highest) for client type 2, station cpu: 1
workload for client type 2, station cpu: 10
priority class (0 is highest) for client type 3, station cpu:
```

The order in which the processes are passed through during the setting of the parameters doesn't matter. Therefore the cpu process doesn't know that the number of client types is already available. To know how the setting can be stopped we type a *?*.

```
priority class (0 is highest) for client type 3, station cpu: ?
      Lower limit...... 0

      Abort command with "$", terminate loop with "^"
priority class (0 is highest) for client type 3, station cpu: $
monitor>
```

Remember the "$" and the "^" to stop what you are doing.

The other processes are set in a similar way. If this is done we take a look at the model.

```
monitor> show model

    parent_name:     process_name:     program_name:
```

```
    root              network           rbr
    network           terminal          rbr-station
    network           computer          mva
    computer          cpu               mva-prior-cta
    computer          disk1             mva-station
    computer          disk2             mva-station

process root

number_of_closed_chains: 2
number_of_open_chains: 0

population_vector:

client_type:   population:

         1:             4
         2:             3

process network

way_to_report_inputs: -no_slave_input

visiting_frequencies_per_client_type:

client_type:    terminal:    computer:

         1:      1.00000      5.00000
         2:      1.00000      5.00000

process terminal

service_discipline: infinite_server
workloads:

  client:                  workload:
      1:                     5.00000
      2:                     3.00000

process computer

way_to_report_results: -maximum_population

visiting_frequencies_per_client_type:
```

```
client_type:        cpu:       disk1:      disk2:

         1:      1.00000      0.70000      0.30000
         2:      1.00000      0.60000      0.40000

process cpu

priority_class_and_workload:

client_type:  priority_class:   workload:

         1:                0     5.00000
         2:                1    10.00000

process disk1

service_discipline: first_come_first_served
workload:

client_type:                          workload:

         1:                           10.00000
         2:                           10.00000

process disk2

service_discipline: first_come_first_served
workload:

client_type:                          workload:

         1:                           15.00000
         2:                           15.00000
```

Note that the network process and the computer process have some *options* (starting with a minus sign) as parameters. Options tell something about the way the output will look like, the number of iterations, the accuracy that has to be reached, etc. They all have default values that can be changed during the setting of the parameters of the specific process. Remember that the *?* will show all valid commands (including the options).
Now that the model is defined it seems wise to save it before we start the computations (in case something goes wrong).

```
monitor> save
what: model
file: whatever
monitor>
```

To read this file back into the PET program we only have to type

```
monitor> <whatever
      end-of-file level 1
monitor>
```

5. Computations

The model is defined, the parameters are set, everything is saved, so now the time seems right to start the computation. The command *compute complexity* isn't working to well so we will stick to the *compute results* command.

```
monitor> compute
what: ?
      Commands...
      complexity     results

      Abort command with "$", terminate loop with "^"
what: results
```

Now we have to wait for the prompt to reappear. Hopefully this doesn't take to long......

```
monitor>
```

At last. Lets see what the results are. This can be done for each process. Of course the network process is usually the most interesting process. But it is also possible to report the results of another process.

```
monitor> report process computer
PROCESS

process name: computer
program used: mean value algorithm (mva)

INPUTS

stations in the network: cpu, disk1, disk2

population for closed client types:
```

client type	population
1	4
2	3

visiting frequencies at each stations per client type:

client type	cpu	disk1	disk2
1	1,00000	0.70000	0.30000
2	1.00000	0.60000	0.40000

RESULTS FOR MAXIMUM POPULATION VECTOR:

POPULATION VECTOR : (4, 3)

response time at each station:

client type	cpu	disk1	disk2
1	7.91185	33.83783	35.28636
2	45.43353	37.70097	35.25702

mean queue lengths at each station:

client type	cpu	disk1	disk2
1	0.75022	2.24600	1.00378
2	1.65903	0.82600	0.51497

mean throughput at each station:

client type	cpu	disk1	disk2
1	0.09482	0.06638	0.02845
2	0.03652	0.02191	0.01461

utilization at each station for the closed chains:

client type	cpu	disk1	disk2
1	0.47411	0.66375	0.42670

```
                2          0.36515          0.21909          0.21909

monitor>
```

6. Making some changes

Now suppose we want to replace the program of the cpu process by the program that uses the shadow approximation instead of the Completion Time Approximation. This can be done by using the *replace* command.

```
monitor> replace
what: process
process: cpu
by program: ?
      Commands...
      mva-station    mva-prior-cta  mva-nonexp     mva-prior-shadow

      Abort command with "$", terminate loop with "^"
by program: mva-prior-shadow
monitor>
```

Note that only those algorithms are shown that will fit at that place of the process tree. This is also the case for the *define* command.
Now that the cpu uses another algorithm, we have to set the parameters of this algorithm again. After that we can compute the results and compare them with the results obtained for the first model.
In this way it is very easy to "play" with the model.

If we want to change some parameters of one of the processes, we have to consider the fact that if we use the *set* command, we have to enter *all* parameters of that process. Therefore we've build in an editor, that can be called by the *edit* command. The standard Unix editor *vi* is used. In fact the *edit* command is essentially the same as the following: save the model on a file, say *pet.tmp*, then edit this file with the *vi pet.tmp* command, and finally read back the pet.tmp file with *<pet.tmp*.
The *edit* command is very convenient for those who are acquainted with the *vi* editor. If you are not, we advise to use only the *set* command.

7. Leaving PET, entering Unix

If we are finished with investigating the model, we can leave the program with the *exit* command.

```
monitor> exit
$
```

Lets see if the file we have saved is present.

```
$ ls
whatever     computer.report
$
```

We can view the files with the *more filename* command. The whatever file is almost the same as the output we obtained with the *show model* command, and the process-name.report files are the same as the output of the *report process-name* command.
Files can be removed by typing *rm filename*, and the *lpr filename* command will print the file.

8. And finally...

We hope that this introductory will be a guiding line for new users of the PET package, but most of all we think and hope that PET will be an easy to use, and time-saving Performance Evaluation Tool.
For those who want to write their own modules we refer to the appendix about this subject.

We will finish this appendix with a table containing the basic commands and a table of the modules that are already available.

command	description
define	define a process of the process tree.
replace	replace a program of a process in the process tree.
delete	deletes one or all (use delete process -all) processes.
rename	rename a process.
set	set the parameters of a process.
edit	set the parameters of a process or the model by editing an input file.
save	save a process or the model on a file.
show	show a process, the model, the computation times or a program.
compute	compute the results.
report	report the results of a computation.
describe	describe a program in more detail.
exit	leave PET.

Table B2. The basic commands in PET.

program name	description
mva	uses the MVA-algorithm to solve a network. The slaves of this process have to use mva-.... The mva program itself can be used as a slave for the rbr program.
schweitzer	uses the Schweitzer approximation algorithm. Its slaves have to use a program starting with mva-.
s-fodi	the Schweitzer-FODI approximation (First Order Depth Improvement) is used. Also this program needs slaves that use mva-....
mva-station	computes the sojourn times for a station with a FCFS, a PS or an IS service discipline.
mva-nonexp	can be used for a FCFS station where the workloads have a non exponential distribution.
mva-prior-cta	uses the Completion Time Approximation to compute the mean sojourn times at a Preemptive Resume Priority station.
mva-prior-shadow	the shadow approximation is used to calculate the sojourn times at a PR-PRIOR (Preemptive Resume Priority) station.
rbr	the Row By Row (RBR) algorithm is used to compute results for a closed network consisting of two stations. The slaves of the rbr program have to deliver the throughputs for all populations. Programs that do so are mva and rbr-station.
rbr-multi-prog	this program is similar to the rbr program, except that it is possible to use a multi programming level (a limit on the number of clients) at one of the stations.
rbr-station	computes the throughputs for all populations at a station with a FCFS, PS or IS service discipline.

Table B3. Modules available in the PET package.

Appendix C: Writing new modules

1. Introduction

This Appendix is intended for those PET users who want to write their own modules. It is assumed that the modules are written in the programming language C, in a Unix environment (for instance on the SUN or the Unix-VAX at the Eindhoven University of Technology).

We will start by referring to the *man* command, that can be used to view the manual pages of a program. For example *man mva* will show the manual page of the mva module.

Before you start writing a new module, contact the PET administrator, because he knows in what directories the PET modules and manual pages can be found. You have to specify these directories in your environment variables PET, PATH and MANPATH.

2. Writing the module

First create your own directory, where you want to write your module. The only thing you have to do is to write three files: the capability file called *name.cap*, the program file written in C, called *name.d* and a file we will discuss later: the *makefile*. The extension *d* stands for *dynamic*; in this file it is allowed to use dynamic arrays.
Normally, the PET administrator can provide you with examples of these files.

If the *name.cap* and the *name.d* file are written, a number of other files has to be generated as shown in figure C1. Fortunately this generation is fully automatic. The only thing you have to do is type

make name

The *make* command only works if there's an especially for the PET modules designed *makefile* available in your working directory. It's easiest to copy for instance the makefile from the mva module and then change all appearances of the name *mva* in that file into *name*.

The *name.c* file is the ordinary source file, written in C, that can be compiled into an executable program, called *name*.

If PET is started it searches the directories specified in the PATH environment variable for a *name.cap* and a *name* file, which should both reside in

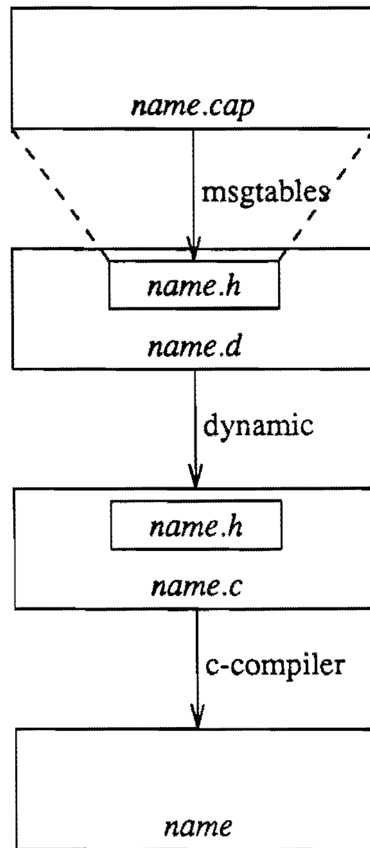

figure C1. name.cap, name.d and other files

the same directory.

3. The name.cap file

The *name.cap* file is the file where all information about the input and output of the module has to be given. An example of such a file can be found in Chapter 6 of this thesis.

This file may contain comment lines, because only the lines starting with one of the special words are taken as input. These words are

fm	for input from the master
tm	for output to the master
ts	for output to the slave
fs	for output from the slave
fu	for input from the user

ns   to specify the number of slaves

Such a special word (except ns) has to be followed by the name of the input or output data message. Only the names as listed in the petdata.h file are excepted. Use *man 5 petdata* to view these data names.

If only one out of two types of data has to be available, their names can be separated by a comma.

It is also possible to specify if the data is available without computations, or if the compute() or complexity() function (or both) has to be used to obtain the data. Simply add fl_compute, fl_complex (or both separated by a "|"), or a zero. So some lines of the *name.cap* file could be:

```
fm nrclosed fl_compute|fl_complex
fm arrivalrateopen fl_compute|fl_complex

This is comment

tm workloadtype 0

ns >= 1
```

To specify the number of slaves (ns) one may use ==, !=, >=, <=, > or <.

4. The name.d file

The *name.d* file is like an ordinary source file written in C, except for the dynamic arrays. Such arrays are needed for data exchange between modules. Information on how to use these arrays can be obtained by typing *man dynamic*.

In chapter 6 all functions are listed that have to be present in the *name.d* file. In one of them, the ask_user() function, you may use some functions that take care of the input from and output to the user. These functions are the cio functions (Conversational Input Output). With the *man cio* command some information about these functions can be obtained.

Only in the compute() part of *name.d* you can use functions to send data to another module, and to ask results from another module. These functions are message() and request().

We will end this appendix with the advise to copy already existing files (as provided by the PET administrator) to your working directory, and then adjust these files so that they become what you want them to be.

References

[1] BASKETT, F., CHANDY, K.M., MUNTZ, R.R., AND PALACIOS, F.G., "Open, Closed and Mixed Networks of Queues with Different Classes of Customers," *J.ACM* **22** (1975), 248-260.

[2] BRANDWAJN, A., "Fast Approximate Solution of Multiprogramming Models," *Performance Eval.Rev.* **11**-4 (1982), 141-149.

[3] GRIENT-DREUX, A. P. DE, "Performance onderzoek naar het TUE VAX-cluster systeem," *Master's Thesis*, Dep. of Math. and Comput.Sci., Eindhoven U. of Techn., Eindhoven (1987).

[4] HOOGENDOORN, J., "Towards a DSS for Performance Evaluation of VAX/VMS-clusters.," *Master's Thesis*, Dep. of Math. and Comput.Sci., Eindhoven U. of Techn., Eindhoven (1988).

[5] HOOGENDOORN, J., MARCELIS, R.C., GRIENT-DREUX, A.P. DE, WAL, J. VAN DER, AND WIJBRANDS, R.J., "The VAX/VMS Analysis and Measurement Packet (VAMP): A Case Study," *Memorandum COSOR 88-09*, Dep. of Math. and Comput.Sci., Eindhoven U. of Techn., Eindhoven (1988).

[6] KOOPMAN, A., "PET, Performance Evaluation Tool," *Master's Thesis*, Dep. of Math. and Comput.Sci., Eindhoven U. of Techn., Eindhoven (1987).

[7] LAZOWSKA, E.D. AND ZAHORJAN, J., "Multiple Class Memory Constrained Queueing Networks," *Performance Eval.Rev.* **11**-4 (1982), 130-140.

[8] LITTLE, J.D.C., "A Proof for the Queuing Formula: $L=\lambda W$," *Oper.Res.* **9** (1961), 383-387.

[9] REISER, M. AND LAVENBERG, S.S., "Mean-Value Analysis of Closed Multichain Queuing Networks," *J.ACM* **27** (1980), 313-322.

[10] SCHWEITZER, P.J., "Approximate Analysis of Multiclass Closed Networks of Queues," Lecture presented at *The International Conference on Stochastic Control and Optimization* (Amsterdam, 1979).

[11] SEVCIK, K.C., "Priority Scheduling Disciplines in Queueing Network Models of Computer Systems," in B. Gilchrist (ed.), *Information Processing 77* (North-Holland, Amsterdam, 1977), pp. 565-570.

[12] WIJBRANDS, R.J., "On an Approximation Method for Priority Queuing in CPU-Disk Models," in *Proceedings NGI-SION Symposium 4: Stimulerende Informatica* (Stichting Informatica Congressen, Amsterdam, 1986).

[13] WIJBRANDS, R.J., "A Note on Enumeration Methods for the Mean Value Analysis Algorithm," *Memorandum COSOR 87-04*, Dep. of Math. and Comput.Sci., Eindhoven U. of Techn., Eindhoven (1987).

[14] WIJBRANDS, R.J., "Queueing Network Models and Performance Analysis of Computer systems," *Ph.D. Thesis*, Dep. of Math. and Comput.Sci., Eindhoven U. of Techn., Eindhoven (1988).

Glossary of notations

M	The number of stations in the network.
m,n	Index denoting a station.
R	The number of closed client types.
L	The number of open client types.
r,s	Index denoting a client type.
$\underline{k}$	The population for the closed clients in the network in vector notation, $\underline{k} = (k_1,..,k_R)$, where k_r denotes the number of closed clients of type r.
$\underline{K}$	The maximum system population.
$\underline{e}_r$	Vector denoting a population of a single client of type r.
$\lambda_1,..,\lambda_L$	The arrival rates for the open client types.
$w_{m,r}$	The average amount of work (called the average *workload*) a client of type r offers at station m. If this mean workload does not depend on the client type, then it can also be written as w_m.
$\sigma^2_{m,r}$	The variance of the workload a client of type r offers at station m.
$R_{m,r}$	The mean residual workload for a client of type r, who is in service at station m, at the moment another client arrives at that station.
$f_{m,r}$	For a closed client of type r, $r = 1,..,R$ this is the mean number of visits to station m during a cycle. This number is also called the *relative visiting frequency*, since the magnitude of the visiting frequencys depends on the choice of the cycle. For the open clients it is the mean number of visits (visiting frequency) of a client of type r, $r = R+1,..,R+L$ to station m during the time this client is in the network.
p^r_m	The probability that an *open* client of type r joins station m when he arrives at the network.
$p^r_{m,n}$	The probability that a client of type r joins station n after leaving station m.
$S_{m,r}[\underline{k}]$	mean time a client of type r spends in the queue during a visit at station m, given population $\underline{k}$. This time is also called the *sojourn time*.
$\Lambda_r[\underline{k}]$	mean throughput rate in the network, measured in cycles per unit of time, for a *closed* client of type r, given population $\underline{k}$.
$\Lambda_{m,r}[\underline{k}]$	mean throughput rate at station m for clients of type r, given population $\underline{k}$. For open clients ($r = R+1,..,R+L$) this throughput rate is independent of the population, and will also be written as $\Lambda_{m,r}$.
$N_{m,r}[\underline{k}]$	mean number of clients at station m of type r, given population $\underline{k}$.
$C_r[\underline{k}]$	mean cycle time for a client of type r, given population $\underline{k}$.
$\rho_{m,r}[\underline{k}]$	mean utilization for a client of type r at station m, given population $\underline{k}$. The mean utilization can be calculated as the product of the mean throughput rate and the average workload. For the open client types ($r = R+1,..,R+L$) this utilization is population independent, and will also be written as $\rho_{m,r}$.

ρ_m	The total utilization of open clients at station m. This is the sum of the utilizations per (open) client type.
$pr(r)$	Priority level for a type r client.
N	Number of priority levels.
$w^*_{m,r}[\underline{k}]$	Adjusted workload for a client of type r at a (transformed) priority station m, given population $\underline{k}$.
$p_{m,r}[k]$	Marginal probability of k clients of type r at station m.
$\underline{k}_m$	Population for the closed client types at station m.
$\lambda_r[k]$	Birth rate for a client of type r if there are k clients of that type in the station.
$\mu_r[k]$	Death rate for a client of type r if there are k clients of that type in the station.
$N^*_{m,r}$	If $N_{m,r}$ denotes the mean number of clients of type r at station m, then $N^*_{m,r}$ is the nearest integer, smaller than $N_{m,r}$.
L_r	Multi programming level for a type r client.