Algorithms for System Performance Analysis

Abhimanyu Chugh
ac1108@ic.ac.uk

Supervisor: Giuliano Casale
Second Marker: Jeremy Bradley

June 19, 2012

Abstract

The prevalence of computer and communication systems nowadays have escalated the need to analyse their performance for purposes of capacity planning. (Closed) Queuing networks are widely used to model these systems, whose performance can be calculated analytically. Until recently, their performance was evaluated using exact algorithms such as Mean Value Analysis (MVA), which prohibit evaluation of large systems servicing several different types of requests with hundreds or thousands of jobs of each type circulating the system, a case commonly encountered in modern applications. To overcome this infeasibility to solve large queuing models, several approximate algorithms have been proposed in the past, which show drastic reduction in computation cost in comparison with MVA and hence, providing a way to evaluate performance of large queuing models.

In this report, we present a library of these approximate algorithms and their implementations in Java. Moreover, we fully integrate them into JMVA, an analytic tool used for performance evaluation and allow users to select different algorithms for evaluation. We also add the ability to compare results of different algorithms (including MVA) graphically, to compare their accuracy.

Furthermore, we discuss their applicability in real application models and conduct an experiment involving comparison of these algorithms and MVA for a range of models measuring parameters, such as runtime and maximum error.

Lastly, in addition to measuring performance indices, we use the algorithms to compute moments for station queue lengths (in a model), in order to determine their mean and variance.

Acknowledgements

I would like to thank my supervisor, Giuliano Casale, for his continuous support and invaluable advice throughout this project, and also my friends and family for their moral support.

Contents

Chapter 1

Introduction

1.1 Motivation

Over the years, computers and communication systems have become progressively more complex, with astounding capabilities, partly due to the presence of a large number of components. Since client-server architectures and distributed systems are so prevalent now-a-days, it has become exceedingly important to analyse the performance of these systems for various reasons, such as to ensure they can cope with varying amounts of workloads the systems will receive, or to check the utilisation of all resources within the system, or even to measure the impact any changes in the architecture would have on the system performance.

Queuing network models have proven extremely helpful in accurately representing real-life computer systems and using the properties of queuing networks, we can easily compute the performance measures for these systems analytically. In this approach, the system is modelled as a network of interconnected queues, where the elements in the queue are the jobs the system needs to service and each queue represents a resource of the system.

In this project, we consider a special subcategory of queuing networks called product-form queuing networks. These networks have a closed-form expression that has enabled the development of efficient algorithms to evaluate their performance [13]. However, this efficiency does not scale well for real-life systems, because of the large number of components involved. This is due to the recursive computation that needs to be performed over a large state space, leading to very high computational costs, even for small networks.

This situation is made worse by the inability to exactly evaluate the performance of systems serving jobs belonging to different workload classes, i.e. jobs that put a different burden on the system depending on their type. We models these systems using multi-class models, where classes represents the different types of jobs. These models are extremely important in effectively modelling and evaluating the performance of modern system architectures. For example, the IT architecture behind modern web-sites comprises of computer servers and multi-tier web applications, which accept different types of request (such as HTTP, FTP) with different costs depending on their type [13]. The evaluation of these systems using exact techniques quickly becomes infeasible as the populations and number of classes increase; both of which are essential for accurately

modelling modern system architectures.

Since the exact evaluation techniques are only feasible for small models and the excessive computation cost for large models makes them infeasible, several approximate evaluation algorithms have been proposed, namely Chow [7], Bard-Schweitzer [15], AQL [17] and Linearizer [6]. These algorithms can compute approximate solutions with a high degree of accuracy, with significant reduction in computation cost, making the evaluation of large models feasible. These algorithms form the basis of this project. There are several software packages available which allow the performance of queuing networks to be evaluated using exact or approximate techniques. One such software is JMVA, which is an analytic tool that uses the exact MVA technique to solve queuing models, however due to infeasibility of exact MVA for large models, JMVA cannot be used to solve large models. Hence, we will be aiming to add the aforementioned approximate algorithms within the tool to extend its functionality, vastly increase the models the tool can be used for and exploring some other uses of the algorithms in this project.

1.2 Contributions

The main contributions of this project are summarised below:

- A Java library of approximate algorithms (namely Chow, Bard-Schweitzer, AQL (Aggregate Queue Length), Linearizer and De Souza-Muntz Linearizer) implemented, and designed in a way to be flexible and ease future additions to the library.
- Complete and seamless integration of the aforementioned Java library in JMVA (part of JMT suite) – JMVA previously only used exact MVA algorithm which is highly inefficient for large models, so the addition of new approximate algorithms allows users to solve even large models within a reasonable amount of time.
- Provision of a clean, usable interface allowing use of these implemented algorithms (and their associated tolerances) through the JMVA interface, and for comparison of these algorithms (graphically and numerically) in what-if analysis mode[1].
- Successful loading and saving state of the model upon opening and saving a model file, complying with the XML structure used by JMVA for model files.
- Computation of different moments, in order to calculate mean and variance of station queue lengths in a queuing model, through a simple command-line interface.

[1]What-If analysis mode in JMVA allows user to solve multiple models changing the value of a control parameter, such as customer numbers, arrival rates, population mix and service demands. The user can specify a range of values for the control parameter, which are then used to compute multiple models. It also allows user to compare the solutions of these models visually on a graph.

- Systematic evaluation of performance of the implemented algorithms with that of the existing MVA for a range of models, covering the aspects of runtime and accuracy.

1.3 Report Structure

The remainder of the report is organised as follows:

- **Chapter 2** provides a brief introduction to queuing networks, lists and defines the model inputs and outputs and finally, covers queuing networks with multiple classes and their strengths and weaknesses.
- **Chapter 3** details the exact and approximate algorithms that are the basis of this project. It also explains how these algorithms could be used to compute different moments for the queuing model. Furthermore, the applications of closed queuing networks are presented, along with a description of JMVA, which is modified as part of this project to incorporate the aforementioned algorithms.
- **Chapter 4** describes the design choices, implementation of the features added to JMVA, data structures used and any points of interest that played an important in realising the project objectives.
- **Chapter 5** presents applicability of the implemented algorithms in real application models, and the results of the experimental evaluation carried out to compare the effectiveness of the algorithms against MVA and conclusions drawn from it. Additionally, it enlists the strengths and weaknesses of the project.
- **Chapter 6** contains concluding remarks about the entire project, as well as future work that could be done to extend the project.

Chapter 2

Introduction to Queuing Networks

Queuing network models are broadly utilised to represent congestion systems (such as communication, computer and production systems) and to evaluate their performance. The model representation of these systems is used to calculate performance measures describing system performance. These include resource utilisation, system throughput and response time. [2]

For this project, we focus primarily on product-form (or separable) queuing network models, which are a subset of general class queuing network models, with constraints over the behaviour of the service centres and customers. They are of special importance because each service centre from the network can be treated individually and evaluated in isolation, and the solution for the whole network is a combination of these individual solutions. They also follow the flow balance assumption, i.e. the number of arrivals is equal to the number of completions. Moreover, product-form queuing network models require significantly less computation than general queuing network models, making them a lot more practical for evaluation purposes [2]. In the following sections, we mention the input and output parameters needed to describe a queuing network, before moving on to the techniques used to evaluate them and their practical applications.

2.1 Model Inputs

A queuing network model mainly comprises of *service centres*, which represent system resources, *customers*, which represent users, jobs or transactions and a *network topology*, which defines the interconnections between the service centres and the path customers follow through the system.

2.1.1 Customer Description

The workload intensity within the system can be described in three ways, depending on the type of workload, as suggested by the following names [11]:

- A *transaction* workload's intensity is defined by a parameter λ, which

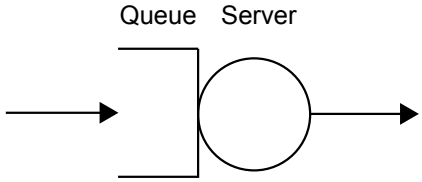


Figure 2.1: A service centre

represents the *arrival rate* of requests or customers. Completed requests leave the model.

- A *batch* workload's intensity is defined by a parameter N, which represents the average number of active jobs/customers and is fixed. Completed jobs leave the model and are immediately replaced from a backlog of waiting jobs.
- A *terminal* workload's intensity is defined by two parameters: N, representing the number of active terminals/customers, and Z, representing the average length of time between customers having completed service and customers' next course of action, i.e. their next interaction with a terminal. Z is commonly referred to as *think time*.

Models with transaction workloads are called *open* models, as they have an infinite stream of arriving customers, whereas models with batch or terminal workloads are called *closed* models, as the customers re-circulate within the system and there is a fixed population. The distinction is essential as methods or algorithms used to evaluate these models differ depending on their type [11]. For the purpose of this project, we will only be focussing on performance analysis algorithms for closed models.

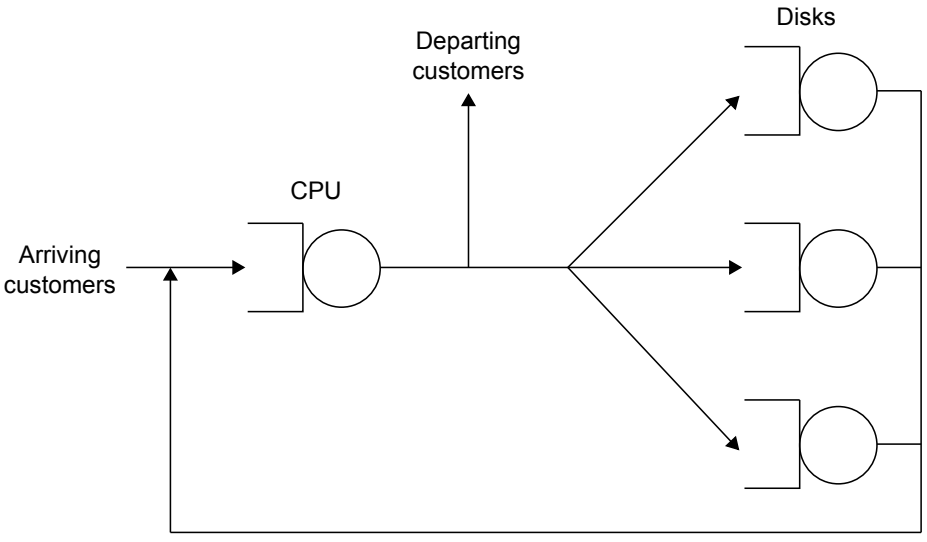


Figure 2.2: An open model

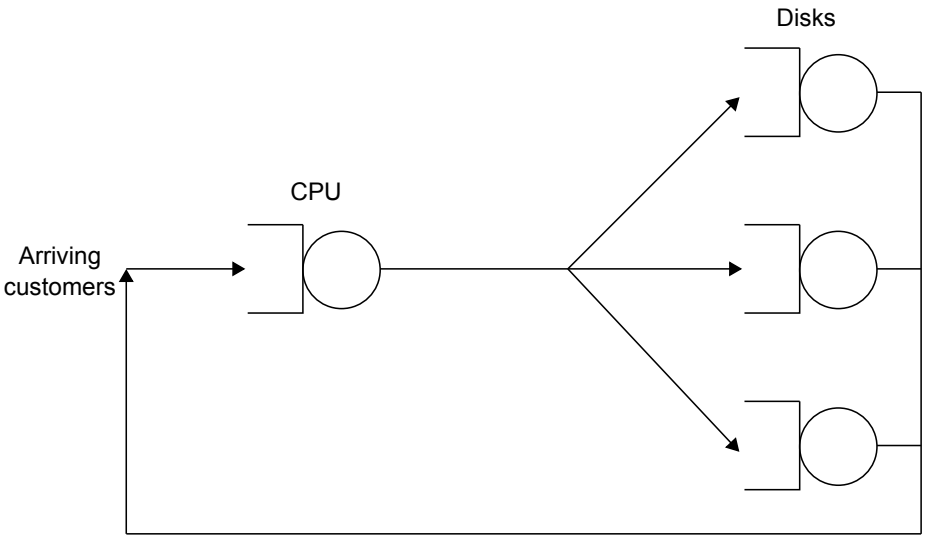


Figure 2.3: A closed model

2.1.2 Service Centre Description

Service centres, also called stations (we use the terms interchangeably), as mentioned before, represent system resources, such as CPU and disk, or different types of servers. They can be divided into two categories: *queuing* and *delay*.

Customers at a queuing service centre contend for the use of server/to be serviced, such as CPU and I/O devices. Hence, time spent by a customer at a queuing centre has two components: time spent waiting and time spent receiving service. On the other hand, there is no competition for service at a delay centre, as each customer (logically) is assigned their own server. Hence the time spent by customers at a delay centre is just their service demand there [11].

2.1.3 Service Demand

The service demand of a customer at centre k, D_k, is the total amount of time it takes for the customer to be serviced at that centre. It can be calculated numerically as B_k/C, where B_k is the total busy time of centre k and C is the number of system completions, or $V_k * S_k$, where V_k is the number of customer visits to the centre k and S_k is the service requirement per visit. The total service demand of a customer at all centres, D, is defined as the sum of individual service demands at each centre, i.e. $D = \sum_{k=1}^{K} D_k$ [11].

2.2 Model Outputs

After the queuing network is evaluated using the above input parameters, the outputs listed below are obtained. Although other outputs can also be calculated, we are mainly looking at the most common/desired outputs for the purpose of this project, i.e. utilisation, residence time, throughput and queue length.

The outputs are obtained for the individual service centres in the network, and also, in some cases, for the system as a whole.

A vital rule used during queuing model analysis is *Little's Law*, which states that the average number of customers in a system, N, is equal to the product of the throughput of that system, X, and the average time a customer spends in that system (i.e. average system response), i.e. $N = XR$. It has many benefits, as listed below, which are essential for algorithms discussed later for evaluating queuing network models [11]:

- It is widely applicable, so it can be used to validate the measurement data.
- It provides a simple, algebraic way to calculate one quantity, provided the other two are known.
- In a computer system model, it can be applied at many different levels: to a single resource, to a subsystem, or even to the system as whole.

Since we also have the think time, Z, for a terminal workload, the Little's Law is revised, as follows, to take this into account.

$$N = X(R + Z)$$

For a batch workload, $Z = 0$, which gives the same equation as earlier, so consistency is maintained. A rearrangement of the above equation expressing R in terms of other quantities is known as the *Response Time Law*, as its application is so ubiquitous:

$$R = \frac{N}{X} - Z$$

2.2.1 Residence Time (R)

The average residence time of a service centre k, R_k, is the mean time spent by a customer at the centre (during all visits to the centre), both waiting and receiving service.

Average system response time, R, is the average interval of time between a customer arrival and departure from the system. It is the sum of the residence times at all service centres: $R = \sum_{k=1}^{K} R_k$

2.2.2 Throughput (X)

Throughput refers to the rate at which customers depart from the system/centre. If a model is parameterised in terms of D_k, then we can get system throughput, X, but we are unable to determine individual service centre throughputs, X_k, due to insufficient information. However, if the model is parameterised in terms of V_k and S_k, then device throughputs are calculated as $X_k = V_k * X$, also known as the forced flow law [11].

2.2.3 Utilisation (U)

The utilisation of a service centre k, U_k, is defined as the proportion of time the centre is busy, or as the average number of customers in service there [11]. This output is only calculated on a per centre basis.

$$U_k = \frac{B_k}{T} = \frac{C_k}{T} \times \frac{B_k}{C_k} = X_k S_k (XV_k S_k = XD_k)$$

The above equation is known as the *Utilisation Law*, a special case of Little's Law. It states that the utilisation of a resource is equal to the product of the throughput of that resource and the service requirement at that resource.

2.2.4 Queue Length

The average queue length at centre k, Q_k, is the average number of customers at that centre, both waiting and receiving service. The number of customers waiting is, simply, $Q_k - U_k$, since U_k is the average number of customers in service at centre k.

The average number of customers in the system is represented by Q, the calculation of which differs depending on the workload [11]:

- Batch workload: $Q = N$
- Transaction workload: $Q = XR$ (from Little's Law)
- Terminal workload: $Q = N - XZ$ (from Little's Law and Response Time Law)

In general, the average population of any subsystem can be determined either by calculating the product of the throughput of that subsystem and the residence time of that subsystem, or by summing the queue lengths at the centres belonging to the subsystem [11].

2.3 Multiple Class Networks/Models

In a basic queuing network model, all customers are identical for all intents and purposes, i.e. they are based on the same probability distribution and routed through the network in the same manner. However, a more realistic representation of a congestion system would likely have various types of customers with distinct workload intensities and resource usage, or where inputs or outputs are specified for the individual types rather than the aggregate system workload. This characteristic can be represented in a queuing network model via a well-known concept of *customer classes*.

In single (customer) class models, all customers are indistinguishable from one another. Although they are the simplest models, they can accurately represent real systems and provide correct performance data, as long as the level of detail provided by them is sufficient for the user.

In multiple (customer) class models, each customer class has its own workload intensity and its own service demand at each centre. The customers within

each class are indistinguishable. The outputs are provided in terms of the individual customer classes, in addition to the outputs from earlier, which are given for single class models. Typical scenarios where multiple class models are beneficial include computer systems with a mixture of CPU and I/O bound jobs, web servers handling HTTP and FTP requests and communication networks supporting TCP and UDP connections. This technique provides a lot more detail and a better understanding of the performance of the system. However, the main strength of multiple class models is also its prime weakness, as the ability to define distinct workloads requires you to spend additional effort on providing input parameter values for each workload [11]:

- The inclusion of multiple customer classes means that multiple sets of input parameters are required, leading to a more tedious data input stage of the process.
- The majority of current measurement tools do not provide enough information to determine the input parameters necessary for each customer class with the same level of accuracy as the single class models.
- The solution techniques available for multiple class models are much more complex, and hence, more difficult to implement and demand more system resources than the single class techniques.

Multiple class models comprising solely of open (transaction) classes are called *open* models, whereas models comprising solely of closed (batch and terminal) classes are called *closed* models. Models that have a mixture of both types of classes are called *mixed* models.

Consider a multiple class model with C customer classes, its workload intensity would be described by a vector with an entry for each class: $\overrightarrow{\lambda} = (\lambda_1, \lambda_2, \ldots, \lambda_C)$ for an open model, $\overrightarrow{N} = (N_1, N_2, \ldots, N_C)$ for a closed model (with the addition of $\overrightarrow{Z} = (Z_1, Z_2, \ldots, Z_C)$ for terminal classes), and $\overrightarrow{I} = (N_1 \textit{ or } \lambda_1, N_2 \textit{ or } \lambda_2, \ldots, N_C \textit{ or } \lambda_C)$ for a mixed model. The service demand of a class c customer at centre k is defined as $D_{c,k}$ and analogous to single-class case, the total service demand of a class c customer at all centres, D_c, is defined as: $D_c = \sum_{k=1}^{K} D_{c,k}$ [11].

All performance measures for multiple class models are obtained on a per-class basis (such as $U_{c,k}$ and X_c) as well as on an aggregate basis (such as U_k and X). For utilisation, queue length and throughput, the aggregate performance measure equals the sum of the per-class performance measures (e.g. $Q_k = \sum_{c=1}^{C} Q_{c,k}$). However, for residence time and system response time, the per-class measures must be weighted by relative throughput [11]:

$$R_k = \sum_{c=1}^{C} \frac{R_{c,k} X_c}{X} \quad \textit{and} \quad R = \sum_{c=1}^{C} \frac{R_c X_c}{X}$$

Chapter 3

Queuing Network Analysis Algorithms

The solution of a queuing network model obtained after evaluation is a set of performance measures that describe the time averaged (or long term) behaviour of the model. For product-form queuing network models, solutions can be determined analytically. The techniques used for evaluating them differ for open and closed models.

For open models (with transaction workloads), the throughput is given as an input (with Forced Flow Law in mind). Therefore, the solution can be easily calculated algebraically using successive applications of the formulae mentioned in [11]. In fact, tools already exist that can solve open models in all cases. On the other hand, closed queuing networks (with batch or terminal workloads) are more difficult to evaluate and rely on iterative algorithms for solutions, requiring significant system resources. Thus, for the purpose of this project, we focus on some of these solution techniques and approximate ones for closed queuing models. Additionally, the specific algorithm used for evaluation depends on the size and complexity of the model. We cover these algorithms in the following sections.

3.1 Mean Value Analysis (MVA)

The mean value analysis algorithm differs slightly for single class and multiple class models, as model inputs and performance measures for individual classes have to be taken into account. We consider each in turn.

3.1.1 Single Class Models

In the case of single class models, MVA algorithm is based on the following three equations [11]:

1. *Little's Law (Response Time Law version) applied to the queuing network as a whole:*

$$X(N) = \frac{N}{Z + \sum_{k=1}^{K} R_k(N)} \tag{3.1}$$

where $X(N)$ is the system throughput, $R(N)$ the average system response time and $R_k(N)$ the residence time at centre k, given a population of N customers in the network. As mentioned earlier, $Z = 0$ for batch workloads.

2. *Little's Law applied to individual service centres:*

$$Q_k(N) = X(N)R_k(N) \tag{3.2}$$

where $Q_k(N)$ is the average queue length at centre k, given a population of N customers in the network.

3. *Service centre residence time equations:*

$$R_k(N) = \begin{cases} D_k & \text{(delay centres)} \\ D_k[1 + A_k(N)] & \text{(queuing centres)} \end{cases} \tag{3.3}$$

where $A_k(N)$ is the average number of customers seen at centre k on arrival of a new customer, or simply, arrival instant queue length at centre k, and D_k the service demand at centre k.

Considering these three equations and model inputs, we can see that the first step to evaluating the model is computation of $A_k(N)$. Two basic techniques exist for this: exact and approximate (in Section 3.2). The two techniques differ in how the arrival instant queue lengths are computed. It is important to note the distinction between these techniques refers to how the solution relates to the model, rather than to the computer system itself. For now, we focus on the exact solution technique.

The exact solution technique incorporates computation of arrival instant queue length $A_k(N)$ exactly, followed by application of equations (3.1)–(3.3) to obtain the required performance measures. For a closed, product-form queuing network, this is defined as simply:

$$A_k(N) = Q_k(N - 1) \tag{3.4}$$

This can be shown with the following example. Consider a closed queuing network with a population of N. When a customer arrives at a centre, it is not already in the centre's queue. This implies only the other $N-1$ customers could interfere with the new arrival and the number of these actually in queue is just the average queue length at the centre when only those $N - 1$ customers are in the network [11].

After applying the above equations iteratively, we can obtain system throughput, average centre queue lengths and centre residence times when there are n customers in the network, given we know the average centre queue lengths with $n - 1$ customers. For the base case, we observe that all queue lengths are zero with zero customers in the network. Using this and equations (3.1)–(3.4), we can compute the solution for the scenario with one customer in the network, and the average queue length obtained from that can be used as the arrival instant queue length for the network with two customers. Similarly, successive applications can provide solutions for customer populations up to N [11].

Algorithm 1 Exact MVA Solution Technique for single class models [11]

for $k = 1 \rightarrow K$ **do**
    $Q_k = 0$
end for
for $n = 1 \rightarrow N$ **do**
    $sumR = 0$
    for $k = 1 \rightarrow K$ **do**
        $R_k = \begin{cases} D_k & \text{(delay centres)} \\ D_k[1 + Q_k] & \text{(queuing centres)} \end{cases}$
        $sumR = sumR + R_k$
    end for
    $X = \frac{N}{Z + sumR}$
    for $k = 1 \rightarrow K$ **do**
        $Q_k = XR_k$
    end for
end for

From Algorithm 1, we can see that it has the time complexity of $N * K$ arithmetic operations, as the equations need to be applied N times and each iteration requires looping over all K service centres. The space complexity, on the other hand, is just K, as the results from previous iterations do not need to be stored.

Once Algorithm 1 terminates, the other performance measures can be calculated using Little's Law, as summarised below:

System throughput:	X
System response time:	$N/X - Z$
Average number in system:	$N - XZ$
Service centre k throughput:	$X * V_k$
Service centre k utilisation:	$X * D_k$
Service centre k queue length:	Q_k
Service centre k residence time:	R_k

3.1.2 Multiple Class Models

Consider a closed, multiple class model with C customer classes and a workload intensity defined by: $\overrightarrow{N} = (N_1, N_2, \ldots, N_C)$ where N_c is class c's population size, and $\overrightarrow{Z} = (Z_1, Z_2, \ldots, Z_C)$ where Z_c is class c's think time. Similar to single class models, multiple class MVA is based on three central equations, except adjusted for per-class performance measures [11]:

1. *For each class, Little's Law applied to the queuing network as a whole:*

$$X_c(\overrightarrow{N}) = \frac{N_c}{Z_c + \sum_{k=1}^{K} R_{c,k}(\overrightarrow{N})} \tag{3.5}$$

2. *For each class, Little's Law applied to individual service centres:*

$$Q_{c,k}(\overrightarrow{N}) = X_c(\overrightarrow{N}) R_{c,k}(\overrightarrow{N}) \tag{3.6}$$

The total queue length at a centre k is also a useful measure:

$$Q_k(\overrightarrow{N}) = \sum_{c=1}^{C} Q_{c,k}(\overrightarrow{N}) \tag{3.7}$$

3. *For each class, the service centre residence time equations:*

$$R_{c,k}(\overrightarrow{N}) = \begin{cases} D_{c,k} & \text{(delay centres)} \\ D_{c,k}[1 + A_{c,k}(\overrightarrow{N})] & \text{(queuing centres)} \end{cases} \tag{3.8}$$

where $X_c(\overrightarrow{N})$ is the throughput for class c, $Q_{c,k}(\overrightarrow{N})$ the average queue length for class c at centre k, $R_{c,k}(\overrightarrow{N})$ the residence time for class c at centre k, $D_{c,k}(\overrightarrow{N})$ the service demand for class c at centre k, and $A_{c,k}(\overrightarrow{N})$ the arrival instant queue length at centre k seen by an arriving customer of class c.

As with the single class models, the performance measures can be computed once $A_{c,k}(\overrightarrow{N})$ are known, and there are two techniques used for evaluation: exact and approximate (discussed in the next section). Similar to single class MVA algorithms, the two techniques differ in the computation of the arrival instant queue lengths. For now, we focus on the exact technique.

To calculate arrival instant queue length, the same analogy as in single class models can be used to obtain this generalisation [11]:

$$A_{c,k} = Q_k(\overrightarrow{N - 1_c}) \tag{3.9}$$

where $(\overrightarrow{N - 1_c})$ is population $\overrightarrow{N}$ with one class c customer removed.

As with the single class models, we start with the base case of population $\overrightarrow{0}$ ($Q_k(\overrightarrow{0}) = 0$ for all centres k) and apply equations (3.5)–(3.9) iteratively to obtain solutions for increasing populations. It should be noted that in general, the solution for each population $\overrightarrow{n}$ needs C input solutions, one for each population $\overrightarrow{n - 1_c}$, c = 1,..., C. Figure 3.1 demonstrates this by showing the network populations for which solutions need to be computed to evaluate a network with three class X customers ($N_X = 3$) and two class Y customers ($N_Y = 2$): the solution from the base case is used to compute solutions for populations with one customer, i.e. (1X, 0Y) and (0X, 1Y), which are used in turn to compute solutions for populations comprising of two customers, and so on, until the solution for population (3X, 2Y) is obtained. Due to these recursive dependencies, the time and space requirements of the multiple class MVA algorithm are considerably larger than those of the single class algorithm. There are roughly proportional to:

time: $CK \prod_{c=1}^{C} N_c + 1$ *arithmetic operations*

space: $K \prod_{c=1}^{C} N_c + 1$ *storage locations*

It is clear that the performance of MVA would deteriorate as number of classes and/or class populations increase. Therefore, it is infeasible to compute the solutions of closed networks with exact technique with more than four

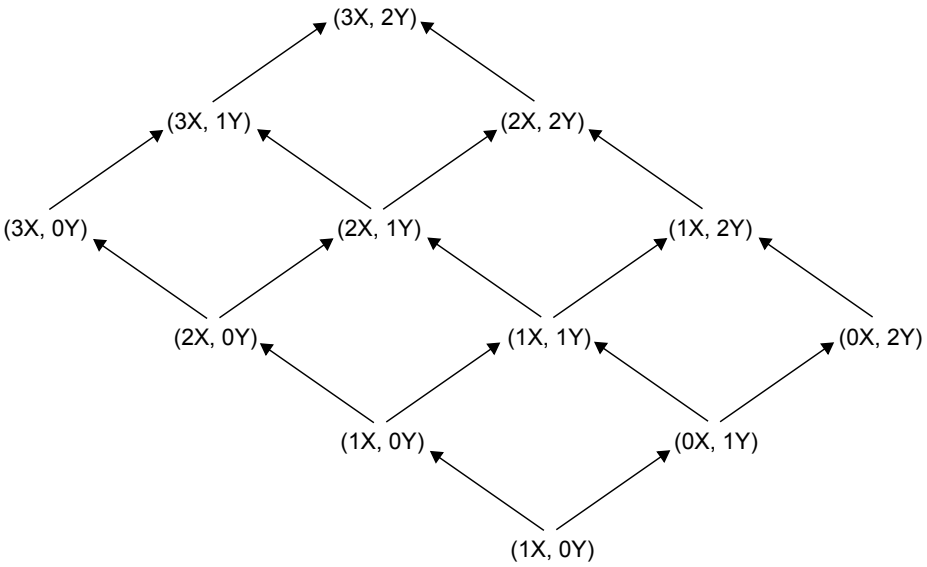


Figure 3.1: Population graph for Exact MVA

classes. This is the motivation behind the development of approximate solution techniques as discussed in the next section. [11]

Algorithm 2 Exact MVA Solution Technique for multiple class models [11]

for $k = 1 \rightarrow K$ **do**
  $Q_k(\overrightarrow{0}) = 0$
end for
for $n = 1 \rightarrow \sum_{c=1}^{C} N_c$ **do**
  for each feasible population $\overrightarrow{n} \equiv (n_1, \ldots, n_C)$ with n total customers **do**
    for $c = 1 \rightarrow C$ **do**
      for $k = 1 \rightarrow K$ **do**
        $$R_{c,k} = \begin{cases} D_{c,k} & \text{(delay centres)} \\ D_{c,k}[1 + Q_k(\overrightarrow{n - 1_c})] & \text{(queuing centres)} \end{cases}$$
      end for
      $X_c = \frac{n_c}{Z_c + \sum_{k=1}^{K} R_{c,k}}$
    end for
    for $k = 1 \rightarrow K$ **do**
      $Q_k(\overrightarrow{n}) = X_c R_{c,k}$
    end for
  end for
end for

As with single class MVA algorithm, once Algorithm 2 terminates, the other performance measures can be calculated using Little's Law, as summarised below:

Class c system throughput:	X_c
Class c system response time:	$N_c/X_c - Z_c$
Average number of class c in system:	$N_c - X_c Z_c$
Class c throughput at centre k:	$X_c * V_{c,k}$
Class c utilisation at centre k:	$X_c * D_{c,k}$
Class c queue length at centre k:	$X_c * R_{c,k}$
Class c residence time at centre k:	$R_{c,k}$

3.2 Approximate Mean Value Analysis (AMVA)

Due to the large time and space requirements of the exact MVA algorithm for large numbers of classes, the approximate mean value analysis is the only feasible method for evaluation. AMVA algorithms provide significant time and space improvements by substituting non-recursive approximations for arrival instant queue lengths, $A_{c,k}(\overrightarrow{N})$.

We still use the equations from exact MVA technique (equations (3.5)–(3.8)); however, the arrival instant queue lengths are estimated iteratively.

We obtain the estimates for $A_{c,k}(\overrightarrow{N})$ based on the average queue lengths at the service centres with the full customer population, avoiding the need to solve models for populations from zero up to full customer population. We start with an initial guess for average queue lengths, which is then input into an approximating function and the resulting value for arrival instant queue length is substituted into equation (3.8). Applying equations (3.5)–(3.7) to the new residence values provides new estimates for average queue lengths, which are used in the next iteration. We keep iterating until successive estimates of these queue lengths are within a given tolerance (specified by the user) [11]. Next, we describe some of these AMVA algorithms.

3.2.1 Chow Algorithm

The Chow algorithm was proposed by We-Min Chow [7]. The algorithm is based on the assumption that the mean queue lengths of service centres are identical in the model with full customer population and in the model with one less customer, i.e. $Q_k(\overrightarrow{N}) \approx Q_k(\overrightarrow{N - 1_c})$. This is reasonable for models with large customer populations. In addition, its computational complexity is several orders of magnitude less than that of exact MVA [7]. Substituting the assumption into equations (3.4) and (3.9), the system can be solved iteratively using some tolerance between values at the start and end of each iteration, and/or maximum number of iterations.

3.2.2 Bard-Schweitzer Algorithm

The Bard-Schweitzer algorithm was proposed by Y. Bard and P. J. Schweitzer [15] in 1979. Bard proposed the same assumption as the one in Chow algorithm, however Schweitzer immediately suggested an improvement, such that removing a customer from a class affects the queue lengths of only that class and no other. Hence, when calculating the queue length of a service length with one less customer of class c, we scale the queue length with class c to reflect the

omitted customer, while leaving the values of the other classes unchanged. A generalised formalised is presented in equation (3.10).

$$Q_k(\overrightarrow{N-1_c}) \approx \frac{N_c - 1}{N_c} Q_{c,k}(\overrightarrow{N}) + \sum_{\substack{j=1 \\ j \neq c}}^{C} Q_{j,k}(\overrightarrow{N}) \tag{3.10}$$

This algorithm provides more accurate estimates than Chow algorithm. Its computational complexity is $\mathcal{O}(CK)$, which is drastically better than exact MVA for the multi-class case.

3.2.3 Linearizer Algorithm

The Linearizer algorithm, proposed by Chandy and Neuse [6], is a very well-known approximation technique for closed, product-form queuing network models, which provides far more accurate results than Bard-Schweitzer. While maintaining a high level of accuracy, it avoids the pitfall of exact MVA algorithm, i.e. the need for recursive computation of solutions/performance measures for all populations from $\overrightarrow{0}$ up to $\overrightarrow{N}$ in order to get a complete solution. Instead, Linearizer approximates, heuristically, performance measures for population $\overrightarrow{N-1_j}$ using information about population $\overrightarrow{N}$ and then using the MVA equations to work out new performance measures for population $\overrightarrow{N}$. The method followed is described below [6]:

First, define $F_{c,k}(\overrightarrow{N})$ as the fraction of class c customers at centre k, when the population is $\overrightarrow{N}$, for all classes c and centres k:

$$F_{c,k}(\overrightarrow{N}) = \frac{Q_{c,k}(\overrightarrow{N})}{N_c} \tag{3.11}$$

Also, define $S_{c,k,j}(\overrightarrow{N})$ as the difference in the fraction of class c customers at centre k, when population is $\overrightarrow{N}$ and the same fraction when population is $\overrightarrow{N-1_j}$ (i.e. with one less class j customer), for all classes c and j, and centres k:

$$S_{c,k,j}(\overrightarrow{N}) = F_{c,k}(\overrightarrow{N-1_j}) - F_{c,k}(\overrightarrow{N}) \tag{3.12}$$

Upon substituting population $\overrightarrow{N-1_j}$ into equation (3.11) and using the result along with (3.12), we can derive the expression for average queue length of class c customers at centre k, when population is $\overrightarrow{N-1_j}$:

$$Q_{c,k}(\overrightarrow{N-1_j}) = (\overrightarrow{N-1_j})_c (F_{c,k}(\overrightarrow{N}) - S_{c,k,j}(\overrightarrow{N})) \tag{3.13}$$

where $(\overrightarrow{N-1_j})_c$ is the population of class c with one class j customer removed from full population $\overrightarrow{N}$.

We cannot calculate $S_{c,k,j}(\overrightarrow{N})$ using equation (3.12) because of the unknown $Q_{c,k}(\overrightarrow{N-1_j})$ values, which are necessary. We, instead, estimate the $S_{c,k,j}(\overrightarrow{N})$ values and then work out approximations for $Q_{c,k}(\overrightarrow{N-1_j})$ using equation (3.13). The Linearizer algorithm estimates the values of $S_{c,k,j}(\overrightarrow{N})$ by successive calls to the Core Algorithm, although the Core algorithm requires $S_{c,k,j}(\overrightarrow{N})$ values to be passed as inputs. The Core algorithm is as follows: [6]

Algorithm 3 Linearizer Core Algorithm

Step 1:
Initialisation: Obtain estimate values for $S_{c,k,j}(\overrightarrow{N})$ and $Q_{c,k}(\overrightarrow{N})$ for all classes c and j and centres k.

Step 2:
Compute new approximations for $Q_{c,k}(\overrightarrow{N-1_j})$ from equations (3.11) and (3.13) for all c, k, j.

Step 3:
From the above approximations for $Q_{c,k}(\overrightarrow{N-1_j})$, compute new estimates for $Q_{c,k}(\overrightarrow{N})$ and $R_{c,k}(\overrightarrow{N})$ using Little's law and the MVA equations.

Step 4:
If the maximum difference between new and old estimates of $Q_{c,k}(\overrightarrow{N})$ is below a given tolerance, then terminate and return the new estimates. Otherwise, go to Step 2.

As mentioned earlier, the Linearizer algorithm calls Core algorithm and computes its own, more accurate estimates of $S_{c,k,j}(\overrightarrow{N})$, from the estimates returned by the Core algorithm. The Linearizer algorithm also makes the following assumption:

$$S_{c,k,j}(\overrightarrow{N-1_j}) = S_{c,k,j}(\overrightarrow{N}) \tag{3.14}$$

The code of the actual Linearizer algorithm is shown in Algorithm 4.

Algorithm 4 Linearizer Algorithm

Step 1:
Initialisation: Assume uniform distribution of customers of each class for population $\overrightarrow{N}$, i.e. $Q_{c,k}(\overrightarrow{N}) = N_c/K$.
Assume, $Q_{c,k}(\overrightarrow{N-1_j}) = (\overrightarrow{N-1_j})_c/K$ for all c, k, j.
Assume $S_{c,k,j}(\overrightarrow{N}) = 0$ for all classes j.
Because of equation (3.14), we can assume $S_{c,k,j}(\overrightarrow{N-1_j}) = 0$.
Set $I = 1$.

Step 2:
Invoke the Core algorithm at population $\overrightarrow{N}$, passing the latest values of $S_{c,k,j}(\overrightarrow{N})$ and $Q_{c,k}(\overrightarrow{N})$ as inputs. Isolate the execution of Core algorithm so that it does not interfere with values used in Linearizer.

Step 3:
If I=3, then terminate, otherwise continue.

Step 4:
Invoke the Core algorithm at all populations $\overrightarrow{N-1_j}$, for all classes j, passing the latest values of $S_{c,k,j}(\overrightarrow{N-1_j})$ and $Q_{c,k}(\overrightarrow{N-1_j})$ as inputs. $Q_{c,k}(\overrightarrow{N})$ can be used in place of $Q_{c,k}(\overrightarrow{N-1_j})$, because of equation (3.14).

Step 5:
Compute estimates of the fractions $F_{c,k}(\overrightarrow{N})$ and $F_{c,k}(\overrightarrow{N-1_j})$ for all c, k, j, using equation (3.11) and subsequently, estimates of $S_{c,k,j}(\overrightarrow{N})$ for all c, k, j, using equation (3.12). With equation (3.14) in mind, assume $S_{c,k,j}(\overrightarrow{N-1_i}) = S_{c,k,j}(\overrightarrow{N})$ for all c, k, j, i.

Step 6:
Set I=I+1, and then go to Step 2.

Upon termination of the algorithm, use the final values of $Q_{c,k}(\overrightarrow{N})$ and $R_{c,k}(\overrightarrow{N})$ to compute other performance measures using the MVA equations. Although the termination condition for Linearizer could be set to a maximum change being below a given tolerance, in practice, continuing for more than four iterations provides minimal improvement, and hence, is probably not worth the extra computation [6].

The time complexity of Core algorithm is $\mathcal{O}(KC^2)$, since it iterates over all classes c and j, and centres k in Step 2, and since the Linearizer algorithm, calls Core algorithm $K+1$ times every iteration, its own time complexity comes to $\mathcal{O}(KC^3)$ and its space complexity is $\mathcal{O}(KC^2)$. Hence, there is a trade-off between accuracy and time taken to compute the solution.

3.2.4 De Souza-Muntz Linearizer Algorithm

While being the best approximation algorithm, Linearizer is very computationally heavy. Therefore to overcome this issue, E. De Souza E Silva and Richard R.

Muntz suggested an improvement, which reduces the complexity from $\mathcal{O}(KC^3)$ to $\mathcal{O}(KC^2)$, without compromising on accuracy [8].

The changes proposed are made to the Core algorithm. It exploits the fact that in Step 2 of Core algorithm even though we compute $Q_{c,k}(\overrightarrow{N-1_j})$, we only use $Q_k(\overrightarrow{N-1_j})$ in Step 3 to compute the residence times, i.e. $R_{c,k}(\overrightarrow{N})$. Therefore, we calculate just $Q_k(\overrightarrow{M-1_j})$ using the equations (3.15) and (3.16) [8].

For the case where $(\overrightarrow{M} = \overrightarrow{N})$, we have

$$Q_k(\overrightarrow{M-1_j}) = Q_k(\overrightarrow{M}) - \frac{Q_{j,k}(\overrightarrow{M})}{N_j} + S'_{j,k}(\overrightarrow{M}) - S_{j,k,j}(\overrightarrow{M}) \tag{3.15}$$

where $S'_{j,k}(\overrightarrow{N}) = \sum_{c=1}^{C} N_c S_{c,k,j}(\overrightarrow{N})$.

However, for the case where $(\overrightarrow{M} = \overrightarrow{N-1_i})$, we have

$$Q_k(\overrightarrow{M-1_j}) = Q_k(\overrightarrow{M}) - \frac{Q_{j,k}(\overrightarrow{M})}{(\overrightarrow{M})_j} + S'_{j,k}(\overrightarrow{M}) - S_{j,k,j}(\overrightarrow{M}) - S_{c,k,j}(\overrightarrow{M}) \tag{3.16}$$

such that $(\overrightarrow{M})_j > 0$.

In summary, the following modifications are made to the original Linearizer algorithm [8]:

1. In Steps 1 and 5 of the Linearizer algorithm, compute $S'_{j,k}(\overrightarrow{M})\forall j,k$ before any other computations and save the values for use in the calls to the Core algorithm during Steps 2 and 3.

2. Step 2 of the Core algorithm is replaced by a computation of $Q_k(\overrightarrow{M-1_j})\forall j,k$ using equation (3.15) if $(\overrightarrow{M} = \overrightarrow{N})$ or equation (3.16) if $(\overrightarrow{M} = \overrightarrow{N-1_c})$ with the precomputed values for $S'_{j,k}(\overrightarrow{N})(= S'_{j,k}(\overrightarrow{N-1_l}))$ and $S_{c,k,j}(\overrightarrow{N})(= S_{c,k,j}(\overrightarrow{N-1_i}))$.

$S'_{j,k}(\overrightarrow{N})\forall j,k$ can be computed at a cost of $\mathcal{O}(KC^2)$ and these values are used for each of the $(C+1)$ calls to the Core algorithm, whose complexity has been reduced to $\mathcal{O}(KC)$. Thus, the cost of this optimised Linearizer has been reduced to $\mathcal{O}(KC^2)$.

3.2.5 Aggregate Queue Length (AQL) Algorithm

The Aggregate Queue Length (AQL) algorithm, proposed by Zahorjan, Eager and Sweillam [17], improves on original Linearizer in terms of costs of both time and space, while maintaining almost the same level of accuracy in solutions.

It achieves this by working with aggregate per-server queue lengths instead of per-class queue lengths. This clearly yields an improvement by a factor of C in time and space costs, resulting in time and space complexities being roughly proportional to KC^2 and KC, respectively, as opposed to KC^3 and KC^2, respectively, for the original Linearizer. Although, the time complexity

of AQL is the same as that of Improved Linearizer, it improves on space cost, by a factor of C, due to the use of aggregate queue lengths.

In contrast to Linearizer which uses $S_{c,k,j}$ terms that specify the change in the queue length of class c at centre k (as a fraction of the total class c population) obtained after removing one class j customer, for all classes c and j, and centres k, AQL uses $\gamma_{c,k}$ terms that specify the change in aggregate queue length at centre k (as a fraction of the total network population) obtained after removing one class j customer from the network, for all classes c and centres k. [17]

In this technique, we compute arrival instant queue lengths ($A_{c,k}(\overrightarrow{N})$) using the following equation [17]:

$$A_{c,k}(\overrightarrow{N}) = (N-1)(\frac{Q_k(\overrightarrow{N})}{N} + \gamma_{c,k}(\overrightarrow{N})) \tag{3.17}$$

where

$$\gamma_{c,k}(\overrightarrow{N}) \equiv \frac{Q_k(\overrightarrow{N-1_c})}{N-1} - \frac{Q_k(\overrightarrow{N})}{N} \tag{3.18}$$

We iterate on the $\gamma_{c,k}$ terms to achieve better approximations successively. After assigning initial values to $\gamma_{c,k}$ terms (typically zero), we use them to compute performance measures, using equations (3.5)–(3.8) and (3.17), for the network population $\overrightarrow{N}$ and for the C populations $\overrightarrow{N-1_j}$ acquired by the removal of one customer from each class j in turn. The following approximation is used for getting performance measures for reduced populations:

$$\gamma_{c,k}(\overrightarrow{N-1_j}) \approx \gamma_{c,k}(\overrightarrow{N}) \tag{3.19}$$

Next, equation (3.18) can be used to compute new $\gamma_{c,k}$ values from the current average queue length estimates, and the updated $\gamma_{c,k}$ terms utilised in the following iteration. In the pseudo code provided in Algorithm 5, the termination condition is satisfied when the change in successive Q_k terms is below some given tolerance. [17]

Algorithm 5 Approximate Queue Length algorithm [17]

1: *// Initialisation*
2: $N = 0$
3: **for** $c = 1 \rightarrow C$ **do**
4: $\quad N = N + N_c$
5: **end for**
6: **for** $k = 1 \rightarrow K$ **do**
7: $\quad Q_k = N/K$
8: $\quad$ **for** $c = 1 \rightarrow C$ **do**
9: $\quad\quad \gamma_{c,k} = 0$
10: $\quad$ **end for**
11: **end for**
12: **repeat**
13: $\quad$ *// Saving old queue lengths used later to check termination condition*
14: $\quad$ **for** $k = 1 \rightarrow K$ **do**
15: $\quad\quad oldQ_k = Q_k$
16: $\quad$ **end for**
17:
18: $\quad$ *// Solve reduced populations*
19: $\quad$ **for** $j = 1 \rightarrow C$ **do**
20: $\quad\quad$ **repeat**
21: $\quad\quad\quad$ **for** $k = 1 \rightarrow K$ **do**
22: $\quad\quad\quad\quad tempQ_k = Q_k$
23: $\quad\quad\quad$ **end for**
24: $\quad\quad\quad$ **for** $c = 1 \rightarrow C$ **do**
25: $\quad\quad\quad\quad R_c = Z_c$
26: $\quad\quad\quad\quad$ **for** $k = 1 \rightarrow K$ **do**
27: $\quad\quad\quad\quad\quad R_{c,k} = D_{c,k}(1 + (N-2)(\frac{Q_k}{N-1} + \gamma_{c,k}))$
28: $\quad\quad\quad\quad\quad R_c = R_c + R_{c,k}$
29: $\quad\quad\quad\quad$ **end for**
30: $\quad\quad\quad\quad$ **for** $k = 1 \rightarrow K$ **do**
31: $\quad\quad\quad\quad\quad Q_k = (\sum_{c=1}^{C} N_c \frac{R_{c,k}}{R_c}) - \frac{R_{j,k}}{R_j}$
32: $\quad\quad\quad\quad$ **end for**
33: $\quad\quad\quad$ **end for**
34: $\quad\quad$ **until** $max_k \left| \frac{tempQ_k - Q_k}{Q_k} \right| \geq tolerance$
35: $\quad\quad$ **for** $k = 1 \rightarrow K$ **do**
36: $\quad\quad\quad \gamma_{j,k} = \frac{Q_k}{N-1} - \frac{oldQ_k}{N}$
37: $\quad\quad$ **end for**
38: $\quad$ **end for**
39: $\quad$ *// Solve full population*
40: $\quad$ **repeat**
41: $\quad\quad$ **for** $k = 1 \rightarrow K$ **do**
42: $\quad\quad\quad tempQ_k = Q_k$
43: $\quad\quad$ **end for**

Algorithm 5 Approximate Queue Length algorithm [17] (continued)

for $c = 1 \rightarrow C$ **do**
  $R_c = Z_c$
  for $k = 1 \rightarrow K$ **do**
    $R_{c,k} = D_{c,k}(1 + (N-1)(\frac{Q_k}{N} + \gamma_{c,k}))$
    $R_c = R_c + R_{c,k}$
  end for
  for $k = 1 \rightarrow K$ **do**
    $Q_k = \sum_{c=1}^{C} N_c \frac{R_{c,k}}{R_c}$
  end for
end for
until $max_k \left| \frac{tempQ_k - Q_k}{Q_k} \right| \geq tolerance$
until $max_k \left| \frac{oldQ_k - Q_k}{Q_k} \right| \geq tolerance$

Due to all approximations being iterative, their accuracy (and thus, AQL's) is dependent upon the selection of the termination condition in use (normally, when the difference between successive queue length values is below some tolerance). Zahorjan, Eager and Sweillam observed that, in practice, as the number of classes in a model increases, the accuracy of AMVA algorithms decreases [17].

3.3 Moment Analysis

In this section, we discuss how we could use the above algorithms, along with some techniques involving moments, to compute the mean and variance of queue lengths of stations.

A moment is a quantitative measure about a distribution, such as mean, variance and skewness. In our case, the distribution we consider is for mean queue lengths of stations. There are many different kinds of moments. The ones we are interested in are power, binomial and central moments.

The n-th power moment of a discrete random variable N, which takes non-negative integer values, is defined as [9]

$$E[N^n] = \sum_{k=0}^{\infty} k^n P(N = k)$$

where $P(N = k)$ is the probability that N takes the value k and E is the expectation operator.

From the above equation, it is clear that the first power moment $(E(N))$ is the population mean, as mean is the expected value of a distribution.

Similarly, the n-th binomial moment of a variable N with binomial distribution is defined as [9]

$$E\left[\binom{N}{n}\right] = \frac{1}{n!} E[N(N+1)...(N+n-1)] = \sum_{k=n}^{\infty} \binom{k}{n} P(N = k)$$

Using the first power-moment, we can obtain the n-th central moment as

following [9]

$$E[(N - E(N))^n] = \sum_{k=0}^{\infty}(k - E(N))^n P(N = k)$$

Since the first power moment is the mean and by definition variance is the average value of the quantity $(distance from mean)^2$), we can deduce that the second central moment is the variance of the distribution.

Since our distribution is defined on station queue lengths, which are discrete and can only take a finite number of values (as they cannot exceed the total population in the models), unlike the variable N (which takes non-negative integers values), we define the n-th power moment of a station k for a given total population N as [4]

$$E[n_k|N] = \sum_{n_k=0}^{N} n_k p_k(n_k|N) \tag{3.20}$$

where n_k is the number of jobs at station k ($n_k \geq 0$) and $p_k(n_k|N)$ is the marginal probability that there are n_k jobs at station k. Here, the first moment represents the mean number of jobs at station k, i.e. its mean queue length.

In order to compute higher-order binomial moments of queue lengths for a queuing model (a notion used in an algorithm called Method of Moments (MoM)), we add a certain numbers of station replicas to the queuing network.

Let $\Delta\overrightarrow{m} = (\Delta m_1, \Delta m_2, ..., \Delta m_M)$ be a vector of non-negative integers (i.e. $\Delta m_k \geq 0$), which represent the number of replicas of station k, for all stations $1 \leq k \leq M$, to add to the original queuing network, i.e. Δm_2 is the number of replicas we add of station 2. Further assuming that all queues are distinct, i.e. $m_k = 1$, the (joint) binomial moment of queue-length can be defined as [5]

$$E[\Delta\vec{m}, \vec{N}] = \sum_{\vec{S}\in\mathcal{S}(\vec{N})} \prod_{k=1}^{M} \binom{n_k + \Delta m_k}{n_k} Pr(\vec{S}) \tag{3.21}$$

where $\vec{S}$ is a state of the model of the form

$$(n_{0,1}, n_{0,2}, ..., n_{0,M}, ..., n_{c,1}, n_{c,2}, ..., n_{c,k}, ..., n_{c,M}, ...n_{C,M})$$

such that $n_{0,1}$ is the number of jobs of class 0 at station 1, $\mathcal{S}(\vec{N})$ is the state space of the model and $Pr(\vec{S})$ is the probability of the state $\vec{S}$.

Since we want the individual station moments, we simplify equation (3.21) for the case where we only add replicas of station i. The result is the following:

$$E[\Delta\vec{m}, \vec{N}] = \sum_{\vec{S}\in\mathcal{S}(\vec{N})} \binom{n_i + \Delta m_i}{n_i} \prod_{\substack{k=1 \\ k\neq i}}^{M} \binom{n_k}{n_k} Pr(\vec{S})$$

Since $\binom{n_k}{n_k} = 1$, we get

$$E[\Delta\vec{m}, \vec{N}] = \sum_{\vec{S}\in\mathcal{S}(\vec{N})} \binom{n_i + \Delta m_i}{n_i} Pr(\vec{S})$$

We can now use marginal probabilities for the station to compute the r-th binomial moment of queue length of station k, with a given total population of N in the network, using

$$E\left[\binom{n_k+r}{n_k}\right] = \sum_{n_k=0}^{N} \binom{n_k+r}{n_k} p_k(n_k|N) \tag{3.22}$$

where r is the number of replicas of station k we add to the network and p_k are the marginal probabilities for station k.

If we add one replica of station k to the network, we get the following expression for the first binomial moment using equation (3.22).

$$E\left[\binom{n_k+1}{n_k}\right] = \sum_{n_k=0}^{N} (n_k+1) p_k(n_k|N)$$

Expanding this gives

$$E\left[\binom{n_k+1}{n_k}\right] = \sum_{n_k=0}^{N} n_k p_k(n_k|N) + \sum_{n_k=0}^{N} p_k(n_k|N)$$

Since p_k are marginal probabilities, they sum to 1, meaning the latter summation evaluates to 1. We also notice that the former summation is the first power moment. Hence, we obtain the following expression, connecting the first binomial and power moments.

$$E\left[\binom{n_k+1}{n_k}\right] = E(n_k|N) + 1 \tag{3.23}$$

Now, we add two replicas of station k to the network, this would give us the following expression for second binomial moment can be written as

$$E\left[\binom{n_k+2}{n_k}\right] = \sum_{n_k=0}^{N} \frac{(n_k+2)(n_k+1)}{2} p_k(n_k|N)$$

Expanding this gives

$$E\left[\binom{n_k+2}{n_k}\right] = \frac{1}{2} \sum_{n_k=0}^{N} (n_k^2 + 3n_k + 2) p_k(n_k|N)$$

Further expanding and using the fact that marginal probabilities sum to 1, we obtain the following expression

$$E\left[\binom{n_k+2}{n_k}\right] = \frac{1}{2}\left(\sum_{n_k=0}^{N} n_k^2 p_k(n_k|N) + \sum_{n_k=0}^{N} 3n_k p_k(n_k|N) + 2\right) \tag{3.24}$$

We observe that the first summation in (3.24) is the second power moment and the second summation is the first power moment. This relation is summarised below:

$$2E\left[\binom{n_k+2}{n_k}\right] = E(n_k^2|N) + 3E(n_k|N) + 2 \tag{3.25}$$

Next, we discuss how we can use this link between binomial and power moments to compute mean and variance of queue length of stations.

As the first power moment, i.e. mean, is just the mean queue length of the station in the original network, it can be computed by simply solving the original network.

The variance, on the other hand, can be computed using both first and second power moments using [4]

$$Var(n_k|N) = E[n_k^2|N] - (E[n_k|N])^2 \tag{3.26}$$

However, we do not have the second power moment. We can use equation (3.25), but we do not have first and second binomial moments either. So, first, we calculate these.

The first binomial moment can be derived from the first power moment (which we can get by solving the model) using equation (3.23). The second binomial moment of a station can be computed by multiplying the first binomial moment of the station in the original network by the first binomial moment of the station in the same network but with an additional replica of the station in the network. The process is explained below:

1. Find first binomial moment of the station k in the original network, as mentioned before.

2. Solve the model again, but with an additional replica of the station and find the mean queue length of the station k, Q'_k.

3. Using equation (3.23), we get the first binomial moment of station k in this new model, by adding 1 to Q'_k, and multiply it by the first binomial moment from the original network (calculated in Step 1) to compute the second binomial moment of station k in the original network.

Now that we have the second binomial moment and the first power moment, we can calculate second power moment using equation (3.25) and the variance thereafter using equation (3.26).

Since we have to solve the queuing model, in order to compute the mean and variance of station queue lengths. This allows us to utilise the analysis algorithms discussed in the previous section. As MVA would not scale well for models with more than four classes, we can use the approximate techniques for those models. In addition, this work could be further explored to compute the marginal probabilities themselves by using the moments obtained from solving the model and solving the system of linear equations connecting the moments and marginal probabilities with Simplex algorithm.

3.4 Applications of Closed Networks

In this section, we consider the practical applications of closed queuing networks to justify the need for algorithm techniques for evaluating these networks.

The growth in complexity of computer systems and networks in use everyday has largely contributed to the popularity and applicability of closed queuing networks in several areas. Some of these practical applications are discussed below:

1. *Capacity Planning*: Closed queueing networks provide the ability to answer performance questions regarding the capacity of a system architecture. We can model the different services as service centres and compare the performance under different scenarios to determine how much capacity would need to be allocated to a service in order to support peak workload, or the maximum amount of workload the current (or a future) architecture can handle under the given capacity. The performance indices provide good measures for determining system behaviour under these circumstances, for example, utilisation can be used to check for bottlenecks, throughput for saturation. Hence, closed queueing models are routinely used for capacity planning at data centres [12].

2. *Software Applications*: Another application of closed queuing networks is being able to model software applications and the services they communicates with, in order to measure their performance and to see whether any services are having a significant impact. We can use the model to narrow down which service needs immediate attention, if any, to function properly.

3. *Multi-tier web applications*: Closed queuing models are widely used to model multi-tier web applications (the most common IT architecture behind modern web sites), comprising of web servers, database servers and application servers, distributed across a network and communicating centrally. We can model the user interaction with the web servers, as well as the interaction between the servers themselves that goes on behind the scenes, to check the performance of the entire architecture, for example checking for high throughput and low response times, as servicing user web requests is time-critical.

4. *Scalability*: Closed queuing models can also be used to measure scalability of existing system architectures, in terms of work loads as well as the growth of the system itself. This could help in determining the scale at which improvements become noticeable.

5. *Quality of Service (QoS)*: Testing the quality of an IT infrastructure or ensuring a certain QoS is a maintained in order to deliver consistent or better performance to customers are valid applications of closed queuing models. This is very true especially in the case of web services, as the servers have to deliver quick, consistent experience to users, because drop in performance could be a matter of losing or gaining customers. Closed queuing models could be used to ensure no negative impact in encountered on the quality of service provided to the users, as circumstances change, such as high amount of workload in the system, one of the services stop responding. We can also model the current system composition to determine whether a service needs replacement, if it is affecting the QoS. It should be checked when and whether the system reaches saturation, so that steps can be taken to avoid this, as it would impact the time it takes for new requests to be serviced.

6. *Bottleneck identification and tuning*: Closed queuing networks can also be used identify bottlenecks within a system, so that the system could be tuned accordingly to avoid occurrence of bottlenecks. The utilisation of

services could be compared to ensure none of them are being overloaded with requests, while the others work at a fraction of their capacity.

7. *Peer-to-peer networks*: We can model peer-to-peer networks with closed queuing networks to investigate impact that changing system capacity and work load would have

3.5 Software Tools for Evaluating Queuing Networks

3.5.1 JMT

Java Modelling Tools (JMT) is a suite of free open-source applications developed by Politecnico di Milano, intended to be used for performance evaluation, workload characterisation, capacity planning and modelling of computer systems and communication networks. The primary focus of JMT is queuing systems and queuing network models [1]. Its ease of use is accentuated by its use of wizards to describe models and its rich graphical user interface.

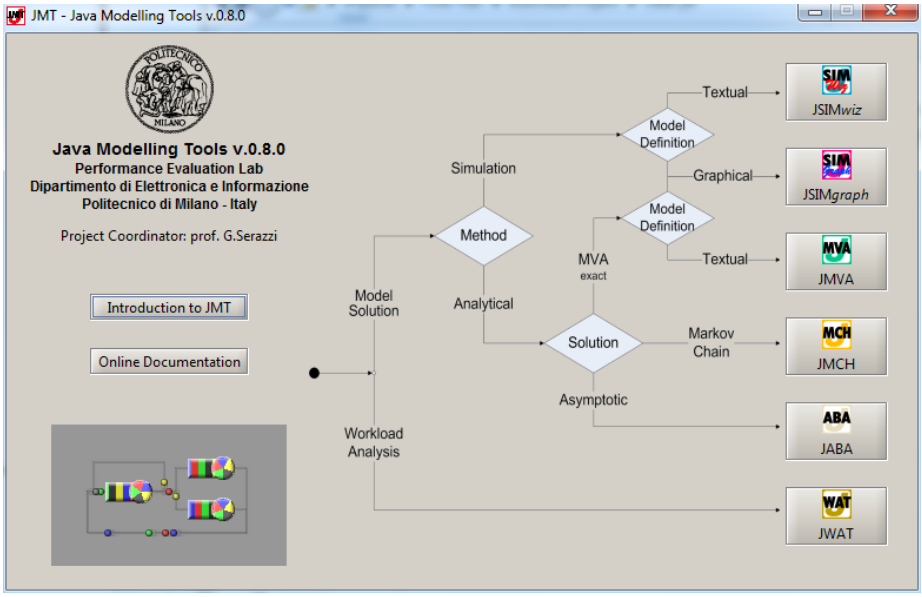


Figure 3.2: Main GUI screen of JMT

It comprises of six distinct applications, each serving a different purpose. One of these applications is JMVA, which is built for analytical evaluation of product-form queuing networks. A user can define a model by stating values for different parameters, such as service demands, arrival rates and/or population, or import a model created graphically using JSIMengine (the engine used in JMT for simulating queuing models). The application can then be used to compute performance measures (such as mean queue length, mean throughput, utilisation and mean response time) for the system being modelled as well as individual service centres in the model, using the exact MVA algorithm for closed

models or similar algorithms for open and mixed models. In the implementation phase of this project, we will be extending the functionality of JMVA to include approximate algorithms for solving closed models.

In contrast to JSIMengine, JMVA has much lower execution times on models, when there are less than three or four customer classes. However, it can have larger memory consumption than JSIMengine for large populations (in hundreds, or more) [3]. This is primarily due to the inefficiency and huge computational cost of exact MVA algorithm. Therefore, to overcome this, we will augment the JMVA application by including implementations of approximate mean value analysis (AMVA) algorithms described earlier, for quicker and fairly accurate evaluations of models in comparison with MVA algorithm.

An extremely useful feature of JMVA is that it provides the ability to perform what-if analysis, i.e. solve multiple models changing the value of a control parameter, such as customer numbers, arrival rates, population mix and service demands [1]. It allows a user to specify a range of values for the control parameter, which are then used to compute multiple models. It also allows user to compare the solutions of these models visually on a graph. In the implementation phase of this project, we will aim to add a new parameter to the what-if analysis window – algorithm selector, which will allow a user to compute the above models (obtained from range of values for the control parameter) under different algorithms and compare the values between the algorithms.

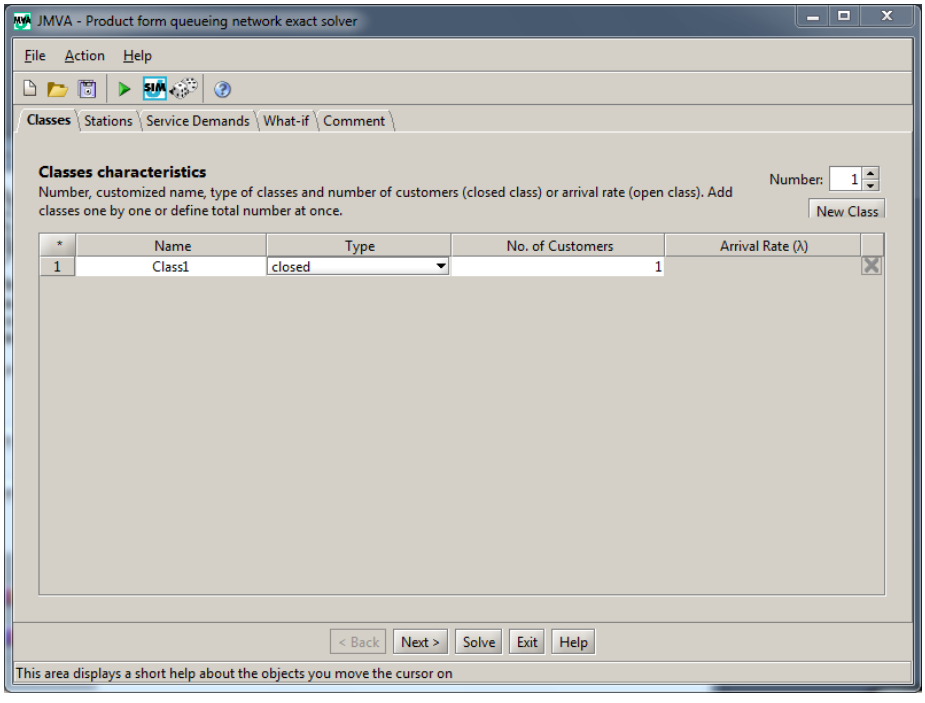


Figure 3.3: Initial screen of JMVA

Chapter 4

Implementation

This chapter covers the architecture, design and implementation aspects of the project. Since, we are extending the functionality of the existing software JMVA, which is managed by several other developers, significant effort was put to highlight the changes made and produce clean and modularised code for easy understanding, extensibility and manageability of the code.

4.1 Key Features

The key additions made to JMVA include:

- Ability to process queuing network model files adhering to a common format (XML).
- Analytical evaluation of closed queuing network models, using approximate algorithms: Chow, Bard-Schweitzer, AQL, Linearizer and De Sousa-Muntz Linearizer, and obtaining performance indices (such as mean throughput, queue length, utilisation for class r or service centre k) as a result of this analysis.
- Ability to graphically compare performance indices for particular stations, classes or aggregates, for different values of input parameters, such as class populations, service demands, and also to graphically compare these values for different algorithms
- Display of algorithm attributes (name, tolerance, iterations) on the Synopsis page of solution panel
- Complete integration with JMVA, from GUI to the underlying analytical engine
- Computation of moments for queuing network models to find mean queue lengths and variance of queue lengths of individual service centres in a given queuing model

4.2 Architecture of JMVA

JMVA has been designed to be flexible. This is achieved by separating the GUI from the underlying analytical solver engine, via an XML layer. This means all the communication between different parts of JMVA and with other tools in JMT is done with the means of the model being expressed in XML. This modularises the analytical engine and allows its reusability in other projects by simply providing a valid XML file.

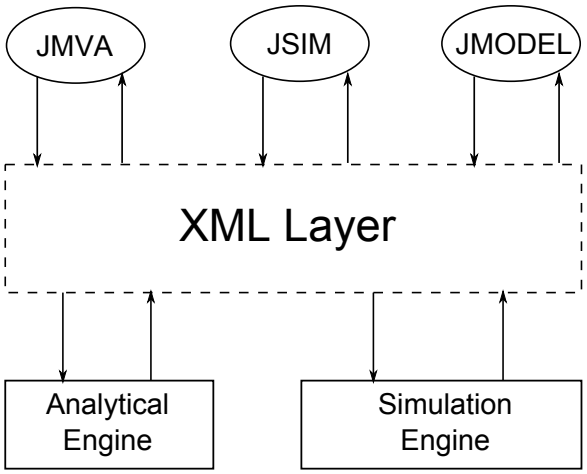


Figure 4.1: JMT Architecture

The core systems of the modified JMVA and their interaction can be represented through Figure 4.2.

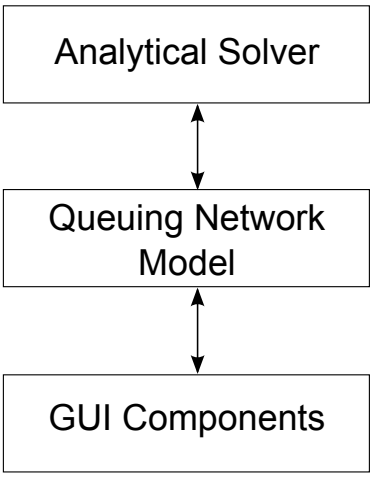


Figure 4.2: Core systems of the modified JMVA

The *GUI Components* here are responsible for providing a user interface to the user, capturing user inputs, which are used for evaluating the model and displaying the solutions to the user, either in numerical or graphical form. The

user inputs are saved in a model and solutions obtained from the same model after evaluation.

The *Queuing Network Model* provides an abstract representation of the queuing model the user wants to solve. It also acts as a model parser, to parse a model into the abstract representation (stored in memory) from an input XML file containing its description. It acts as a bridge between the GUI components and the analytic solver and provides simple APIs to read and modify its state (because of user input from GUI or solutions from solver). In addition, it stores the algorithms that should be used for evaluation of the model, which are set from the options selected by the user in the GUI.

Finally, the *Analytical Solver* is responsible for evaluating the model and computing performance indices. The solver can use many different algorithms for evaluation. The algorithms to use are determined from the model. The result of the evaluation (i.e. performance indices) is sent back to the model, which are in turn read by the GUI components to visualise them on the user interface.

4.3 Design and Implementation

Since JMVA is a big ongoing project with several developers contributing, good software engineering practices (such as use of appropriate patterns) were strictly followed to achieve modularity/flexibility, and ease maintenance and extensibility in the future. However, this also made it difficult to understand the existing codebase, due to the different coding styles practised by the contributors. Only the changes that were absolutely necessary were made to the existing codebase (with the addition of classes relevant to the project), to avoid breaking existing logic and functionality of the program.

Within the JMT project, JMVA is mainly organised into 4 packages, and these are the only packages that were modified during the implementation stage of the project. The packages are as follows:

1. *jmt.analytical* package: This package incorporates the main underlying logic used for solving the queuing models. It contains classes which implement the algorithms discussed in Chapter 3. These classes are only used when the user selects to solve the queuing model. The algorithm to use for evaluation is chosen within this package by the `SolverDispatcher` class, depending on the options specified by user on the GUI. The classes were structured in a way that maximises code reuse. In addition, it includes the class used for calculating mean and variances of queue lengths of service centres in queuing models, as described in Chapter 3. This package corresponds to the *Analytical Solver* component in Figure 4.2.

2. *jmt.gui.exact* package: This package contains the main GUI class for JMVA, while the sub-components of the interface are defined within its sub-packages. It interprets the user's input and calls sub-systems accordingly. It stores a reference to the model, which is used by its sub-components to update the model upon user's request. Moreover, it also contains the `ExactModel` class, which provides an abstract representation of the queuing model in question. Hence, this class corresponds to the

Queuing Network Model component in Figure 4.2, while other classes in this packages and its sub-packages correspond to the *GUI Components*.

3. *jmt.gui.exact.link* package: This package provides a link between the *Queuing Network Model* and *Analytical Solver* components. It connects the input model with the solver dispatcher, which is then called to evaluate the model.

4. *jmt.gui.exact.panels* package: This package contains the sub-components (or panels) of JMVA GUI, which are used to update the model and to specify the evaluation method to use, according to user's specifications. It also includes components used for displaying the result to the user, in numerical or graphical form (where graphical representation is limited to what-if analysis which is used for comparisons).

Although some of the GUI elements were taken from a previous attempt at this project by Georgios Poullaides [14], majority of them were modified and their functionality changed due to them exhibiting incorrect and erratic behaviour. Hence, the final source code as a result of this project looked very different from Poullaides' code.

In the following section, specific implementation details about the packages and classes mentioned in the previous section will be provided.

4.3.1 `jmt.analytical` package

This package primarily contains the implementations of algorithms used for evaluation of performance indices as well as mean and variance of the queuing model, and a class to connect these algorithms with the abstract queuing model. First, we discuss the algorithms, which are divided into single-class and multi-class categories. We implemented a general multi-class solution, which can be easily adapted to single-class, hence we explore multi-class case first.

Multi-class algorithms

The structure of the algorithm classes can be depicted with the class diagram in Figure 4.3.

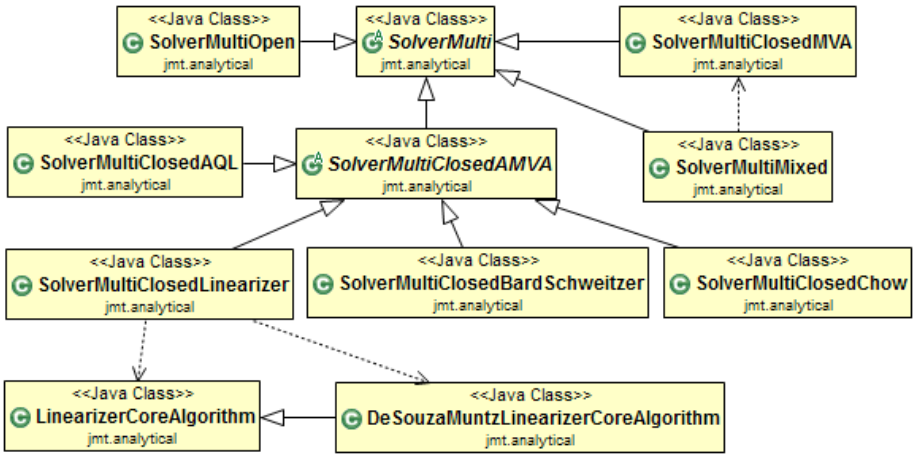


Figure 4.3: Class structure of multi-class algorithms

`SolverMulti` is an abstract superclass, which must be used as the basis for every multi-class queuing model solver. It has an `input(...)` method used for specifying model parameters, such as stations, classes, service demands, class populations, which will be used by its subclasses during evaluation. It also has variables for storing values of performance indices (i.e. throughput, utilisation, queue length and residence times), which will be updated by its subclasses upon evaluation. Finally, it has an abstract `solve(...)` method meant to be implemented by the subclasses, which will provide the main logic for solving the model and computing performance indices.

Since this is an abstract superclass, it serves as an interface for multi-class algorithm solvers and separates the underlying implementation when used for solving a model, reducing dependencies to the concrete implementations.

`SolverMultiClosedMVA`, `SolverMultiOpen` and `SolverMultiMixed` classes were already a part of this program prior to this project, so we just focus on the approximate algorithms for closed models.

`SolverMultiClosedAMVA` is another abstract class, which is a subclass of `SolverMulti`. It serves as a superclass for all approximate MVA algorithms, as it predefines variables, such as iterations, and tolerance and maximum number of iterations, which will be used by every approximate algorithm, to keep track of the number of algorithm iterations and termination criteria, respectively. Since these are recursive algorithms which improve upon each iteration, termination criteria is needed. The termination criteria is checked at the end of each iteration and is set so that the algorithm stops when maximum number of iterations are reached or the maximum difference between old and new queue lengths is below a certain tolerance, specified by the user (this can be seen clearly in Listing 4.1). It also provides methods for validating a tolerance value used by the GUI components and for calculating maximum difference between individual new and old queue lengths, and service centre new and old queue lengths used for checking termination criteria, in order to avoid redundant code in subclasses. Moreover, it implements a basic approximate MVA algorithm (in `solve(...)` method), using equations (3.5)–(3.9) from Chapter 3, with recursion based on the queue lengths, i.e. the termination criteria is that the maximum difference

between old and new queue lengths after an iteration must be less than or equal to the given tolerance. However, it leaves the calculation of queue length with one less customer, i.e. $Q_k(\overrightarrow{N-1_c})$, used in equation (3.9), for subclasses to implement. For algorithms that use this approximate methodology (for example Chow and Bard-Schweitzer), they only have to implement this abstract method, providing the benefit of code reuse, while other algorithms (such as Linearizer and AQL) can just override the `solve(...)` method for other methodologies.

```
// Check convergence criteria
if (iterations >= MAX_ITERATIONS || maxDiff(oldQueueLen, queueLen)
    < tolerance) {
   break;
}
```

Listing 4.1: Code snippet that checks for termination criteria in approximate algorithms

Similar to `SolverMulti`, `SolverMultiClosedAMVA` being abstract provides a common way to interact with approximate multi-class algorithm solvers without worrying about the specific implementations.

`SolverMultiClosedChow` and `SolverMultiClosedBardSchweitzer` classes extend `SolverMultiClosedAMVA` and provide implementations of the Chow algorithm and Bard-Schweitzer algorithm, respectively. Since, they only differ in how they calculate the queue length of a service centre with one less customer and the rest of the algorithm is already implemented in `SolverMultiClosedAMVA`, they only add method bodies for the abstract method `getQueueLensWithOneLessCustomer` defined in `SolverMultiClosedAMVA`.

Since Chow algorithm assumes that the queue length with one less customer is the same as the queue length with that customer present (i.e. $Q_{r,k}(\overrightarrow{N-1_c}) \approx Q_{r,k}(\overrightarrow{N})$), this can be seen in the code snippet from `SolverMultiClosedChow` in Listing 4.2, where k is a service centre and c the class of the customer being removed. Similarly, we can notice in the code snippet of `SolverMultiClosedBardSchweitzer` in Listing 4.3 that in the case where $c = r$, each individual queue length is normalised by $(N_c - 1)/N_c$, before summing up for the service centre k, as Bard-Schweitzer dictates.

```
protected double[][] getQueueLensWithOneLessCustomer() {
   double[][] scQueueLens = new double[stations][classes];
   for (int k = 0; k < stations; k++) {
      for (int c = 0; c < classes; c++) {
         double currQueueLen = 0;
         for (int r = 0; r < classes; r++) {
            currQueueLen += queueLen[k][r];
         }
         scQueueLens[k][c] = currQueueLen;
      }
   }
   return scQueueLens;
}
```

Listing 4.2: Code snippet from `SolverMultiClosedChow.java`

```
protected double[][] getQueueLensWithOneLessCustomer() {
```

```
2     double[][] scQueueLens = new double[stations][classes];
3     for (int k = 0; k < stations; k++) {
4        for (int c = 0; c < classes; c++) {
5           double currQueueLen = 0;
6           for (int r = 0; r < classes; r++) {
7              if (c == r) {
8                 if (clsPopulation[r] != 0) {
9                    currQueueLen += queueLen[k][r]*(clsPopulation[r]
                        - 1)/(double)clsPopulation[r];
10                }
11             }
12             else {
13                currQueueLen += queueLen[k][r];
14             }
15          }
16          scQueueLens[k][c] = currQueueLen;
17       }
18    }
19    return scQueueLens;
20 }
```

Listing 4.3: Code snippet from `SolverMultiClosedBardSchweitzer.java`

The `SolverMultiClosedAQL` class extends `SolverMultiClosedAMVA` and contains the implementation of the AQL (Aggregated Queue Length) algorithm mentioned in Chapter 3. This class overrides the `solve(...)` method in its superclass, as it follows a different technique from the basic one implemented in its superclass. Upon termination, the values for performance indices are saved in the variables of its superclass, which are accessed later to be displayed on GUI.

The `SolverMultiClosedLinearizer` class extends `SolverMultiClosedAMVA` and contains the implementation of the Linearizer algorithm mentioned in Chapter 3. Since Linearizer has a sub-algorithm within it, called Core algorithm, which takes different set of inputs and works different in general, its implementation has been moved to its own (inner) class (namely `LinearizerCoreAlgorithm`), inside `SolverMultiClosedLinearizer`, modularising it and keeping it independent of the Linearizer implementation.

Moreover, since the Linearizer optimisation proposed by E. De Souza and Richard Muntz only requires a change in the Core algorithm, we only had to extend the `LinearizerCoreAlgorithm` to add their proposed implementation of Core algorithm, namely `DeSouzaMuntzLinearizerCoreAlgorithm`. The decision about which Core algorithm to use is made by the boolean variable `useDeSouzaMuntz` in Linearizer class, which is initialised in the constructor according to user selection. Since the modification proposed only changes calculation of queue lengths in Step 2 of Core algorithm (recall Section 3.2.4), this step has been moved to its own method `getSCQueueLengths(...)`, which means the modified Core algorithm only has to override this method preventing the need to replicate the rest of the Core algorithm. In the modified Core algorithm, this method requires values for $S'_{j,k}(\overrightarrow{N})\forall j,k$ passed to the algorithm on every call from Linearizer, a variable has been created for it. Linearizer computes these values at the start of every iteration and sets the variable in the instance of the modified Core algorithm thereafter (as shown in Listing 4.4), which are then used in the overridden version of `getSCQueueLengths(...)` to process queue lengths for Step 2 of Core algorithm.

```
if (useDeSouzaMuntz) {
   for (int k = 0; k < stations; k++) {
      for (int j = 0; j < classes; j++) {
         scCustFracDiffs[k][j] = 0;
         for (int c = 0; c < classes; c++) {
            scCustFracDiffs[k][j] += clsPopulation[c]*custFracDiffs
                [k][c][j];
         }
      }
   }
   for (int c = 0; c < classes+1; c++) {
      if (coreResults[c] instanceof
          DeSouzaMuntzLinearizerCoreAlgorithm) {
         ((DeSouzaMuntzLinearizerCoreAlgorithm) coreResults[c]).
             setScCustFracDiffs(scCustFracDiffs);
      }
   }
}
```

Listing 4.4: Code snippet showing computation of S' (service centre fraction differences) in Linearizer class at the start of every iteration and saving these values in the deSouza-Muntz Core algorithm

Single-class algorithms

The implementations written for the multi-class case were designed as a general solution, hence we can use the same implementations with a wrapper, which makes them suitable for single-class case. The structure of the algorithm classes can be depicted with the class diagram in Figure 4.4.

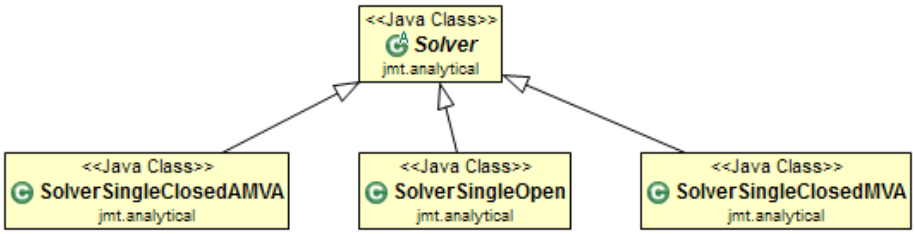


Figure 4.4: Class structure of single-class algorithms

Similar to `SolverMulti`, `Solver` is an abstract superclass, which must be used as the basis for every single-class queuing model solver. It also has an `input(...)` method used for specifying model parameters, such as stations, classes, service demands, class population, and an abstract `solve(...)` method, intended to be implemented by its subclasses. The array variables used for storing values of performance indices (i.e. throughput, utilisation, queue length and residence times) do not have the class dimension, since there is only one class.

Since this an abstract superclass, it serves as an interface for single-class algorithm solvers and separates the underlying implementation when used for solving a model, reducing dependencies to the concrete implementations.

`SolverSingleClosedMVA` and `SolverSingleOpen` classes were already a part of this program prior to this project, so we just focus on the approximate algorithms for closed models.

The `SolverSingleClosedAMVA` class uses the wrapper design pattern to solve single-class queuing models. The wrapper design was used to reuse code from multi-class and to avoid breaking existing design logic, as that might have caused confusions with the other contributors later. In its constructor, it requires the algorithm to use to be passed as a parameter, which initialises its `solver` variable to the relevant multi-class algorithm (this can be seen in Listing 4.5). It overrides the `input(...)` method in its superclass, as it needs to adjust the parameters passed to multi-class case, in order to forward them to the relevant solver. Finally, its `solve(...)` method calls the `solve(...)` method on the previously initiated `solver` and then copies its result into the variables for performance indices defined in its superclass `Solver` (this can noticed in Listing 4.6).

```
1  private void initialiseSolver() {
2      int[] classPop = new int[1];
3      classPop[0] = customers;
4
5      if (SolverAlgorithm.CHOW.equals(algorithm)) {
6          solver = new SolverMultiClosedChow(1, stations, classPop);
7      } else if (SolverAlgorithm.BARD_SCHWEITZER.equals(algorithm)) {
8          solver = new SolverMultiClosedBardSchweitzer(1, stations,
               classPop);
9      } else if (SolverAlgorithm.AQL.equals(algorithm)) {
10         solver = new SolverMultiClosedAQL(1, stations, classPop);
11     } else {
12         solver = new SolverMultiClosedLinearizer(1, stations,
               classPop, SolverAlgorithm.DESOUZA_MUNTZ_LINEARIZER.equals
               (algorithm));
13     }
14     solver.setTolerance(tolerance);
15 }
```

Listing 4.5: Code snippet used for initialising solver `SolverSingleClosedAMVA.java`

```
1  public void solve() {
2      solver.solve();
3
4      totUser = customers;
5      totRespTime = solver.sysResponseTime;
6      totThroughput = solver.sysThroughput;
7
8      queueLen = ArrayUtils.extract1(solver.queueLen, 0);
9      throughput = ArrayUtils.extract1(solver.throughput, 0);
10     residenceTime = ArrayUtils.extract1(solver.residenceTime, 0);
11     utilization = ArrayUtils.extract1(solver.utilization, 0);
12 }
```

Listing 4.6: Code snippet used for solving queuing model in `SolverSingleClosedAMVA.java`

Next, we discuss how these algorithm classes are called and connected with the abstract queuing model.

First, we introduce an enumerated type `SolverAlgorithm`, which we can use to distinguish the algorithm(s) selected by user, as enums are generally easier and quicker to compare than using strings and also considered better coding practice. The enum values for different algorithms are defined in Listing 4.7.

```
public enum SolverAlgorithm {
    EXACT("MVA"),
    CHOW("Chow"),
    BARD_SCHWEITZER("Bard-Schweitzer"),
    AQL("AQL"),
    LINEARIZER("Linearizer"),
    DESOUZA_MUNTZ_LINEARIZER("De Souza-Muntz Linearizer"),
    OPEN("Open"),
    MIXED("Mixed");
}
```

Listing 4.7: `SolverAlgorithm` enum

The text defined in brackets is the string representation of the enum, which will be used by the GUI as algorithm names. The enum stores a static array of values called `CLOSED_VALUES`, which only has the algorithms that can be used for solving closed models (i.e. all except `OPEN` and `MIXED`), which again will be utilised by the GUI to display a choice of algorithms for closed models. Furthermore, the enum has separate methods for checking whether an algorithm is exact, approximate or for closed models, defined in Listing 4.8. These will be used by the GUI to show/hide the tolerance box and to show/hide selection of algorithms for closed models.

```
public static boolean isClosed(SolverAlgorithm alg) {
    return alg == SolverAlgorithm.EXACT ||
           alg == SolverAlgorithm.CHOW ||
           alg == SolverAlgorithm.BARD_SCHWEITZER ||
           alg == SolverAlgorithm.AQL ||
           alg == SolverAlgorithm.LINEARIZER ||
           alg == SolverAlgorithm.DESOUZA_MUNTZ_LINEARIZER;
}

public static boolean isExact(SolverAlgorithm alg) {
    return alg == SolverAlgorithm.EXACT;
}

public static boolean isApproximate(SolverAlgorithm alg) {
    return alg == SolverAlgorithm.CHOW ||
           alg == SolverAlgorithm.BARD_SCHWEITZER ||
           alg == SolverAlgorithm.AQL ||
           alg == SolverAlgorithm.LINEARIZER ||
           alg == SolverAlgorithm.DESOUZA_MUNTZ_LINEARIZER;
}
```

Listing 4.8: Checks whether an algorithm is exact, approximate or closed

Finally, we look at `SolverDispatcher` class which given a model, instantiates the appropriate solver according to number of classes in the model (this decides whether to use single-class or multi-class solvers) and algorithm selected by user (value of which is saved in the model) and then calls the `solve(...)` method in the instantiated solver. Upon solving the model, it passes the solutions to the model where they are saved for later use. If user chose to perform

what-if analysis[1], `SolverDispatcher` makes sure that model is solved and results saved for each value of control parameter, otherwise it only solves the model once with the parameters stored in the model. This process is repeated if more than one algorithm was selected for comparison in what-if analysis (again these algorithms are accessed from the model).

`SolverDispatcher` is used within the `SolverClient` class from `jmt.gui.exact.link` package, which provides the model to dispatcher. Moreover, `SolverDispatcher` adds an inner interface called `SolverListener`, which is used to notify `SolverClient` when computation of an iteration[2] terminates (for what-if analysis, the number of iterations is the number of different values of control parameter). This is useful, as it helps determines how much progress has been made, which is then displayed in a progress window on the GUI to keep user informed. Since this is the only class that communicates with the algorithm solvers and the only class other packages interact with (except the `SolverAlgorithm` enum) within the analytical package, the dependencies between solvers and other packages are minimised, ensuring loose coupling.

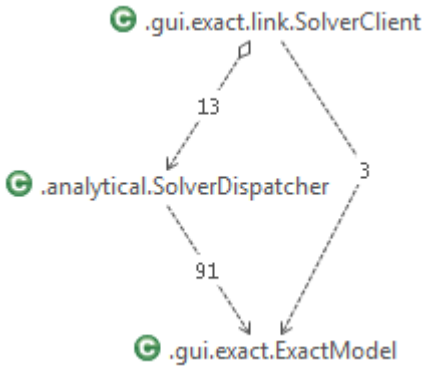


Figure 4.5: Dependencies between these classes

4.3.2 `jmt.gui.exact` package

This package primarily contains two classes essential to JMVA. We explore their roles and usefulness in order:

`ExactModel` class provides a complete representation of the queuing model, as well as all methods required for parsing an input file (containing the model), creating an XML representation of the model which can then be saved to a file and providing access to model parameters to the other classes. The last is

[1]What-If analysis allows user to solve multiple models changing the value of a control parameter, such as customer numbers, arrival rates, population mix and service demands. The user can specify a range of values for the control parameter, which are then used to compute multiple models. It also allows user to compare the solutions of these models visually on a graph.

[2]An iteration here refers to the completion of model evaluation either during normal analysis or for a value of control parameter during what-if analysis, while iteration count within approximate algorithms is the number of times we looped until reaching the termination criteria

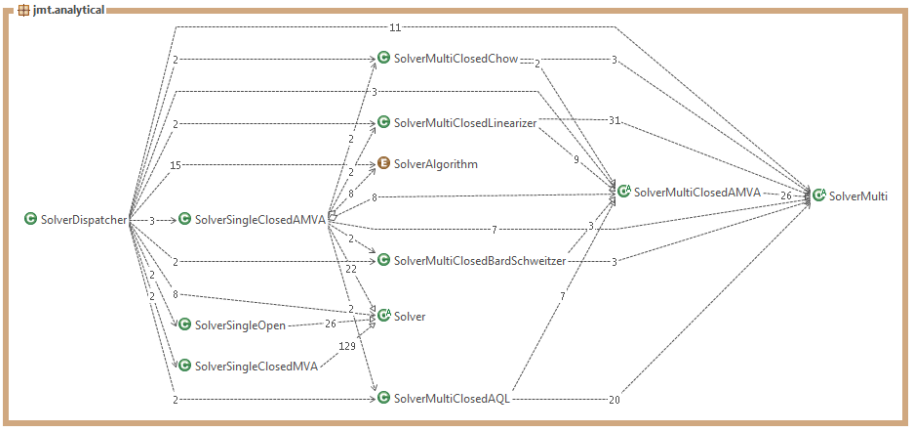


Figure 4.6: Structural diagram of `jmt.analytical` package and dependencies between its classes. As can be seen, there are no cycles, which could have caused issues with maintenance and code reuse.

achieved with the help of getter methods, which provide values such as class populations, number of queues, service demands, algorithms selected by user and their respective tolerances if applicable, as well as performance indices (i.e. throughput, queue length, utilisation and residence times – these are used when displaying results to the user). Additionally, it includes several setter methods used for updating list of selected algorithms and their tolerances as user makes changes and for saving results in the model passed by `SolverDispatcher` after evaluation.

Before this project, JMVA only had one algorithm for solving models, hence to incorporate the addition of other algorithms, variables were created to monitor which algorithms have been selected and their tolerance values in case of approximate algorithms. Furthermore, since we added comparison of performance indices obtained from different algorithms for the same model in what-if analysis (instead of knowing there will only ever using one algorithm), the results from each algorithm had to be stored separately. This was accomplished with the use of a `Map` for each performance index and another variable for storing algorithm iterations. The key of the `Map` is the algorithm and its value the results computed using that algorithm (as shown in Listing 4.9). The built-in implementation of `HashMap` in Java is used for instantiation of these maps, for constant time access.

In light of these changes, methods used for saving a model and parsing an input (model) file had to be updated. We will discuss saving of model first to highlight the XML convention used for representation the model, which will be useful when parsing an input (model) file.

The `createDocument()` method in `ExactModel` provides a DOM (Document Object Model) representation of the model. This is created using the API provided by Java for XML processing. A `Document` is created with a root element, to which we add elements for model parameters (such as stations, classes, service demands, algorithm selections) and their respective values. Moreover, if the model has solutions (from its evaluation) saved within the performance indices'

```
1  /**
2  * number of iterations algorithm performed for each (what-if)
       iteration/execution
3  * dim: algIterations<Algorithm, [iterations]>
4  */
5  private Map<SolverAlgorithm, int[]> algIterations;
6
7  /**
8  * queue lengths
9  * dim: queueLen<Algorithm, [stations][classes][iterations]>
10 */
11 private Map<SolverAlgorithm, double[][][]> queueLen;
12
13 /**
14 * throughput
15 * dim: throughput<Algorithm, [stations][classes][iterations]>
16 */
17 private Map<SolverAlgorithm, double[][][]> throughput;
18
19 /**
20 * residence times
21 * dim: resTime<Algorithm, [stations][classes][iterations]>
22 */
23 private Map<SolverAlgorithm, double[][][]> resTimes;
24
25 /**
26 * utilization
27 * dim: util<Algorithm, [stations][classes][iterations]>
28 */
29 private Map<SolverAlgorithm, double[][][]> util;
```

Listing 4.9: Variable definitions of performance indices. The iteration dimension in the array refers to the results during that iteration of What-If analysis. This will just have 1 value for normal analysis, or n values for what-if analysis where n is the number of control parameter values.

maps, they are also added to the representation, so that the user does not have to re-evaluate a saved model, if they just want the solutions for performance indices again. A sample model file with the added changes is added in Appendix A for reference.

The `loadDocument()` method, on the other hand, is used to read in the DOM representation of a model and copy the parameter values into the current model. The DOM representation is created from the input file using the Java XML API. Starting at root, the elements are read one by one and values copied into the relevant model variables. Furthermore, if the file has solutions saved in it, they are also copied into the relevant performance indices' maps within the model, and after successful loading of the model, a solution window will appear with these results, just like it did when the model was solved before it was saved.

The other important class in `jmt.gui.exact` package is `ExactWizard`, which is the main GUI class for JMVA (referred to as the wizard in subsequent sections). It has several sub-components defined as panels on the GUI. These are organised in the panels sub-package and discussed in Section 4.3.4 of this chapter. This is the only class that stores a reference to the model, so its sub-

components have to access the model through `ExactWizard`, hence a reference to `ExactWizard` is passed when the subcomponents are initialised. The main change made to this class in particular, was the way it decides on the solution window displayed to the user. If, say, the user has selected a few algorithms for comparison in what-if analysis, we have solutions for all those algorithms. Before, JMVA would only have displayed solutions for one of these algorithms. This behaviour was changed by adding a tab for each algorithm in the solution window. Also, since we changed the way we store solutions (by using maps), relevant changes were made to calls to access solutions, in order to pass them to the solution window. The new look of JMVA (or `ExactWizard`) can be seen in Figure 4.7.

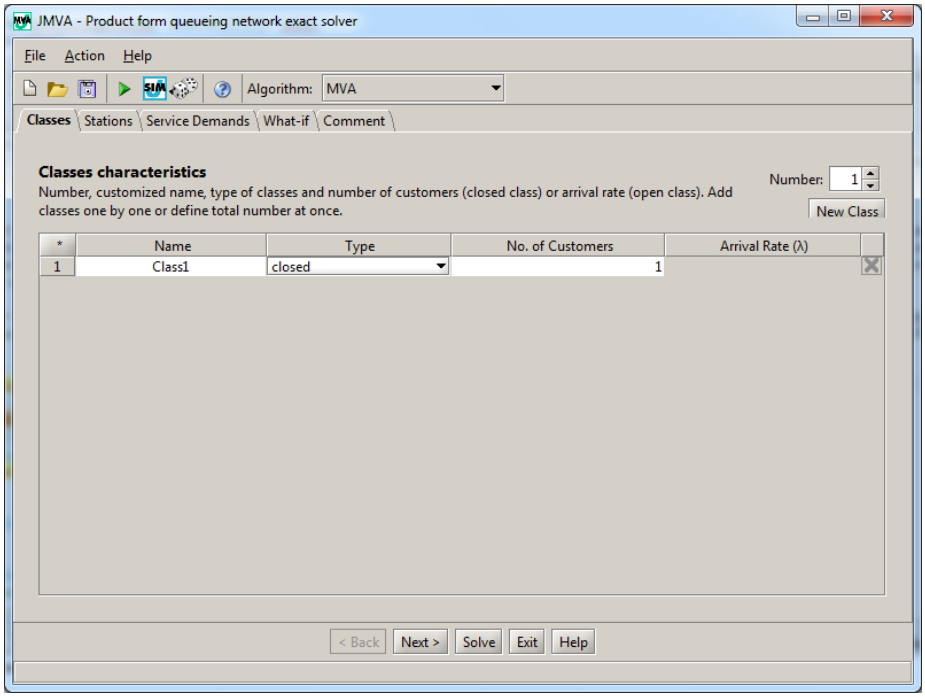


Figure 4.7: New look of JMVA (or `ExactWizard`)

4.3.3 `jmt.gui.exact.link` package

This package only has one class, namely `SolverClient`, which serves as a bridge between the wizard (JMVA GUI) and the analytical solver package. `SolverClient` receives a request from `ExactWizard` to solve the model passed to it as a parameter. The client, in turn, creates a new thread to be used for solving the model. The dispatcher is added to the thread and model handed over to it. Once the evaluation terminates for all iterations (there could be more than one, if what-if analysis is performed), the client returns the model with solutions to `ExactWizard`, which displays the results to the user in a solution window.

4.3.4 `jmt.gui.exact.panels` package

Since the wizard has so many subcomponents, not all of which were modified as part of this project, we mainly focus on the ones that were and the additions made within them, for the betterment of the program.

The first addition made to the GUI was providing users a way to choose between different algorithms. The simplest way to do this was to add a drop-down box (the addition can be seen in Figure 4.8). This was put in its own class called `AMVAPanel`, to distinguish it from other subcomponents. This class extends the abstract class `WizardPanel`, which acts as the superclass of all panels and contains their common elements. In `AMVAPanel`'s constructor, a reference to the wizard (JMVA GUI) is passed, which can be used to access the model or other GUI subcomponents.

The algorithms in the drop-down box are separated by exact and approximate to inform user of their accuracy. Additionally, when the user selects an approximate algorithm (i.e. Chow, Bard-Schweitzer, AQL, Linearizer or De Souza-Muntz Linearizer), user is given the chance to input a tolerance value, as shown in Figure 4.8, the value of which is validated and an error message returned if invalid. Since the tolerance can usually be quite small and is stored as a double in the system, its string value can be in exponential form, which could be confusing for users to read. Hence, a `DecimalFormat` is used to find the decimal representation (up to 15 decimal places).

When the user selects an algorithm in the drop-down box, the `Action-Listener` attached to it accesses the model via `ExactWizard` and updates the algorithm variable. The tolerance text-box works similarly and updates the tolerance variable, however, it uses a `FocusListener` so that the user does not have to press enter to save the tolerance, which is not very intuitive and can be easily forgotten, hence we save the tolerance whenever focus is lost from the box, which is a much better solution.

One of the difficulties that presented during the GUI design was how to add the separators between the exact and approximate algorithms, as a `ComboBox` (equivalent of drop-down box) in Java does not allow you to add an unselectable element in the box. Therefore, a workaround was added to overcome this issue. This involved adding two elements in the combo box, one before start of exact algorithm and one before start of approximate algorithms, which will act as the separators. A custom `ListCellRenderer`, used to render each element in the combo-box, was attached to it, and it checked if an element in the combo-box is not a valid algorithm (validated through `SolverAlgorithm` enum), then disable the element, make it unfocusable and change its background to grey to give it the disabled appearance. Additionally, in the `ActionListener` of the combo-box, whenever one of these non-algorithm elements is selected, do not change the algorithm value in the model or the value in the combo-box, i.e. change the value of combo-box to the value prior to the click, and since the default value is the exact algorithm, reaching a scenario where a greyed element is selected is impossible, avoiding deadlock. This behaviour can be seen in the code snippets in Listings 4.10 and 4.11.

Since the algorithms listed in the drop-down does not support evaluation of open and mixed models, but only closed models, the `ClassesPanel` class within this package was modified. This is the panel that takes user input concerning

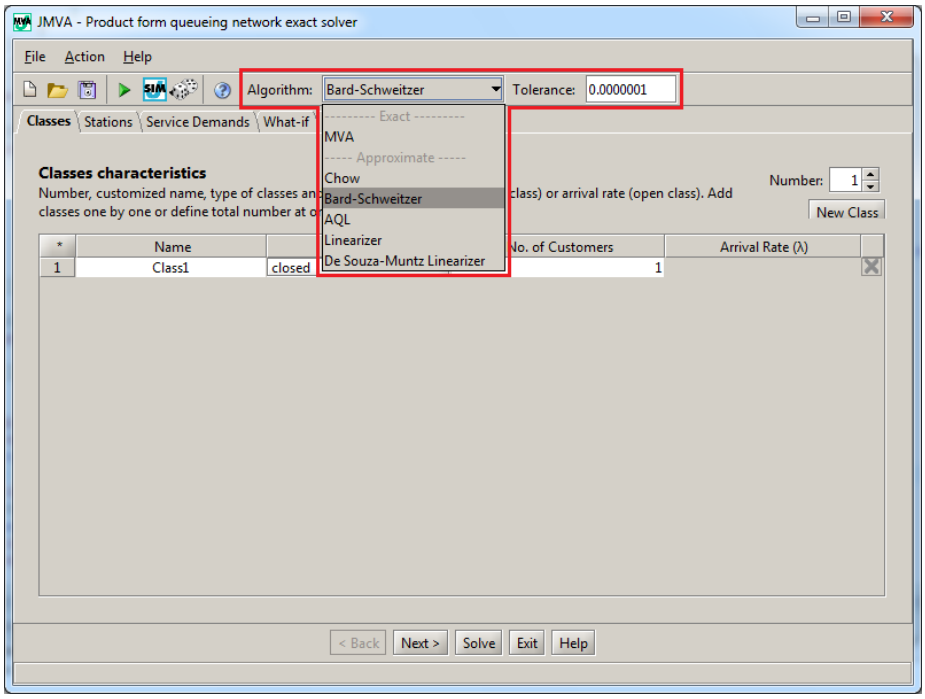


Figure 4.8: The drop-down box added to JMVA to choose algorithm used for evaluation

class parameters, such as class type, population, arrival rate. The change made was such that whenever an open class is added to the model, the drop-down and tolerance boxes are disabled, and open and mixed algorithm solvers used for evaluation instead.

Now that the drop-down box has been added, the next option to add was to allow users to visually compare the results of algorithms against each other. This could help the user in deciding the appropriate algorithm for the evaluation of a bigger model, by comparing the precision in accuracy between the algorithms. This feature was added to the existing what-if analysis in JMVA.

As mentioned earlier, JMVA provides a way to automate evaluations of models, which only different on the value of a control parameter, through what-if analysis. The control parameter can be customer numbers, arrival rates, population mix or service demands. The user can specify a range of values for the control parameter, which are then used to compute multiple models. It also allows users to compare the solutions of these models visually on a graph. This can be seen in Figure 4.9.

```
algorithmList.setRenderer(new DefaultListCellRenderer() {
    @Override
    public Component getListCellRendererComponent(JList list, Object
        value, int index, boolean isSelected, boolean cellHasFocus)
        {
        Component comp = super.getListCellRendererComponent(list,
            value, index, isSelected, cellHasFocus);
        String str = (value == null) ? "" : value.toString();
        if (SolverAlgorithm.find(str) == null) {
            comp.setEnabled(false);
            comp.setFocusable(false);
            setBackground(list.getBackground());
            setForeground(list.getForeground());
        } else {
            comp.setEnabled(true);
            comp.setFocusable(true);
        }
        return comp;
    }
});
```

Listing 4.10: Definition of `ListCellRenderer` used for the algorithm combo-box in `AMVAPanel`

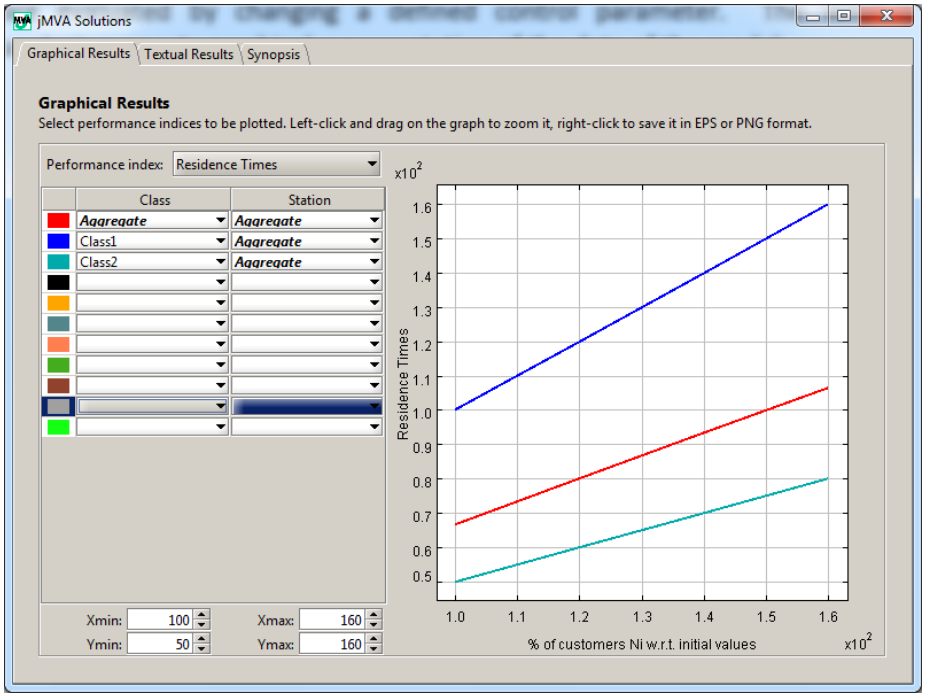


Figure 4.9: Graphical representation of performance indices for different values of control parameter customer classes. Comparison between both aggregate and individual class/station values of performance indices is possible.

Before we add the option to compare different algorithms graphically in the solution window, we need to provide a way for users to select algorithms

```
1  private ActionListener ACTION_CHANGE_ALGORITHM = new ActionListener
       () {
2     // initial value
3     int currentItem = 1;
4
5     public void actionPerformed(ActionEvent e) {
6        JComboBox algorithmList = (JComboBox)e.getSource();
7        algorithm = (String)algorithmList.getSelectedItem();
8
9        // check if algorithm or not
10       if (SolverAlgorithm.find(algorithm) == null) {
11          algorithmList.setSelectedIndex(currentItem);
12       } else {
13          currentItem = algorithmList.getSelectedIndex();
14          ew.getData().setAlgorithmType(algorithm);
15          SolverAlgorithm alg = SolverAlgorithm.find(algorithm);
16          boolean exact = alg != null && SolverAlgorithm.isExact(alg
                );
17          showToleranceField(!exact);
18       }
19    }
20 };
```

Listing 4.11: `ActionListener` defined for the algorithm combo- box in `AMVA-Panel`

they would like to use for comparison. To incorporate this functionality, check boxes were added in `WhatIfPanel` (panel used for inputting values for control parameter), as shown in Figure 4.10. As the figure demonstrates, we have an outer check box (i.e. "Compare Algorithms"), which is used to specify whether the user would like to compare algorithms graphically at all, and the inner check boxes used for selecting the algorithms to be used for comparison. The approximate algorithms have tolerance boxes next to them. Another thing to notice in the figure is that if the "Compare Algorithms" box is selected, the drop-down box for algorithm selection and its corresponding tolerance box are disabled, as their values would be ignored. However, if it is not selected, the algorithm and tolerance from the drop-down box are used for evaluation, to provide a simple way of performing what-if analysis if only one algorithm is to be used.

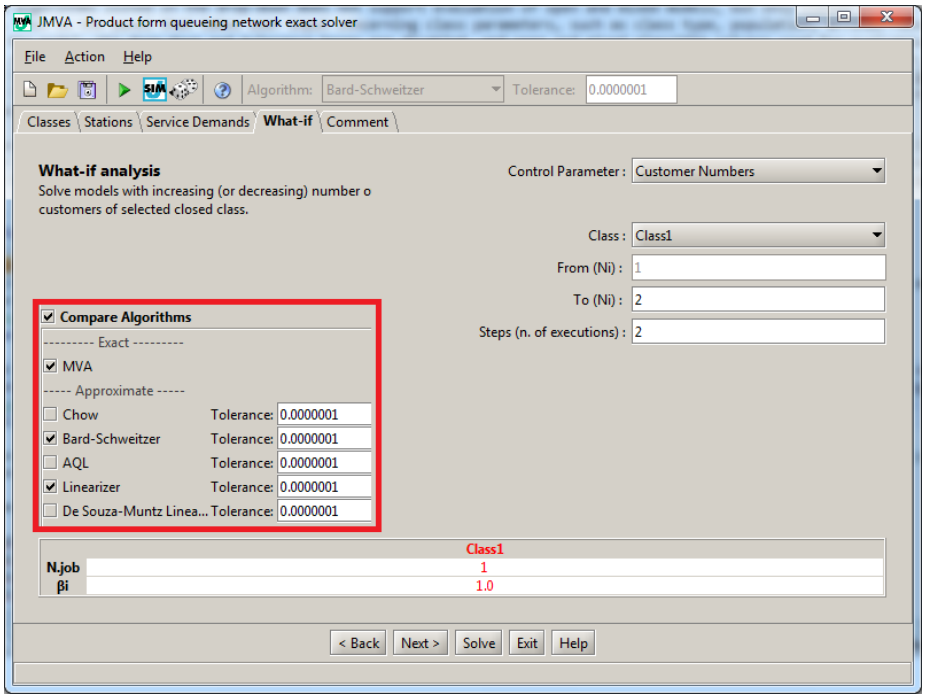


Figure 4.10: Sub-panel of checkboxes, added to `WhatIfPanel`, used for selecting algorithms to use during what-if analysis

An array called `compareAlgs` was added to `ExactModel` for storing the values of the checkboxes, and likewise, `algTols` array for storing the tolerances from text-boxes. They are saved so that these settings can be restored when the user saves the model and opens it again. A `DecimalFormat` is again used for getting a decimal representation of the tolerance. Similar to the drop-down box, the `ActionListener` attached to the checkboxes access the model via `ExactWizard` (a reference to which is again passed to `WhatIfPanel`'s constructor) and updates the relevant value in the `compareAlgs` array. The tolerance text boxes, likewise, update the `algTols` array, however, they use a `FocusListener`, which as mentioned earlier is a more intuitive option.

Next, we discuss how we compare algorithms graphically in the solution window displayed after termination of what-if analysis.

A new column was added to the `GraphPanel` (class responsible for the graphical representation of the performance indices from what-if analysis), to allow the user to select between different algorithms and compare their solutions, as shown in Figure 4.11. For open and mixed models, the algorithm column does not appear. The framework used for plotting the performance index values on the graph is called Ptplot framework (written in Java).

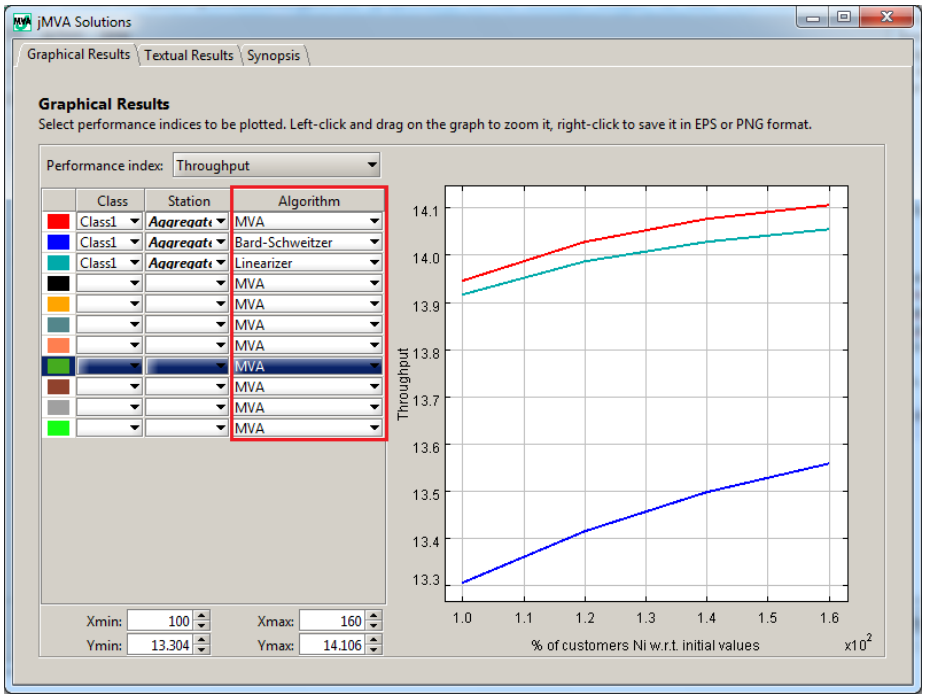


Figure 4.11: The new algorithm column added for comparing solutions from different algorithms

`GraphPanel` uses a custom implementation of `JTable` written as an inner class called `LinesTable`. The combo-boxes used for selecting a class, station and algorithm are stored within a `LinesTable`. It uses the built-in combo-box cell editor for each cell. Each combo-box in the algorithm column is initialised to list of algorithms obtained from the keyset of one of the performance index's map (recall solutions to performance indices were stored in `Map`s with algorithm as the key). `LinesTable` uses a custom table model, namely `LinesTableModel`, which is used to manage the cells in the table. It defines methods for returning the column and row count for the table, whether a cell is editable[3] and the value of a cell[4] and for defining what to do when a cell value (i.e. combo-box value) is changed. The last one is the most interesting one, defined in the method `setValueAt(Object aValue, int rowIndex, int columnIndex)`, where the first parameter represents the selected index in the cell, i.e. combo-box and the other two specify the location of the cell in the table. For each column, it saves the value in an array, to keep track of the class/station/algorithm selected in each row of the table and then updates the performance index values for that row in the graph (Listing 4.12 shows the case for algorithm column and the method called for updating the index values). To update the graph, the saved algorithm value for the modified row is used, in order to get the name of

[3] A cell is editable if it is in the class or station column, or if it is an algorithm column and there is more than one algorithm to choose from. If a cell is not editable, a label is used at its place instead of a combo-box

[4] The value of a cell depends on in which column it appears. A cell in class/station column will be either 'Aggregate' or a class/station name, while a cell in algorithm column will be one of the algorithms selected for evaluation.

the algorithm from the model (using `getAlgorithmName(int index)` method), which subsequently is used to obtain values of the selected performance index for that algorithm from the model, as shown in Listing 4.13 for the case of throughput.

```
1  public void setValueAt(Object aValue, int rowIndex, int columnIndex
       ) {
2      if (columnIndex == 2) {
3          ...
4      } else if (columnIndex == 3) {
5          algorithms[rowIndex] = ((Integer) aValue).intValue() + 2;
6      } else if(columnIndex == 1){
7          ...
8      }
9      // Paints new index
10     paintIndexAtRow(rowIndex);
11 }
```

Listing 4.12: Code snippet from `LinesTableModel` class within `GraphPanel` that shows the steps taken after a value is changed in the table. The '+2' is added because the combo-box editor defined in the table returns the value as $(index - 2)$.

```
1  private void paintIndexAtRow(int rowNum) {
2      ...
3      SolverAlgorithm alg = SolverAlgorithm.find(getAlgorithmName(
           rowNum));
4      // Throughput
5      if (currentIndex.equals(ExactConstants.INDICES_TYPES[0])) {
6          if (classNum >= 0 && statNum >= 0) {
7              graph.draw(rowNum, model.getThroughput(alg)[statNum][
                   classNum]);
8          } else if (classNum < 0 && statNum >= 0) {
9              graph.draw(rowNum, model.getPerStationX(alg)[statNum]);
10         } else if (classNum >= 0 && statNum < 0) {
11             graph.draw(rowNum, model.getPerClassX(alg)[classNum]);
12         } else {
13             graph.draw(rowNum, model.getGlobalX(alg));
14         }
15     }
16     ...
17 }
```

Listing 4.13: Code snippet from `GraphPanel` class for updating the graph

While this covers the graphical representation of the results, JMVA also provides the numerical results to the user. For a normal analysis, these are the only results shown to the user, but for what-if analysis, both graphical and numerical results are shown under separate tabs ('Graphical Results' and 'Numerical Results', respectively). Figure 4.12 shows the solution window after a normal analysis, while Figure 4.13 shows the numerical results from a what-if analysis. As the figures demonstrate, the solutions are distinguished first by the algorithm tabs (in the case of what-if analysis), and then by the performance index tabs. Each tab was implemented as a panel, with the class hierarchy

as shown in Figure 4.14. The performance index panels get their values from the model through the wizard, a reference to which is passed to each panel's constructor.

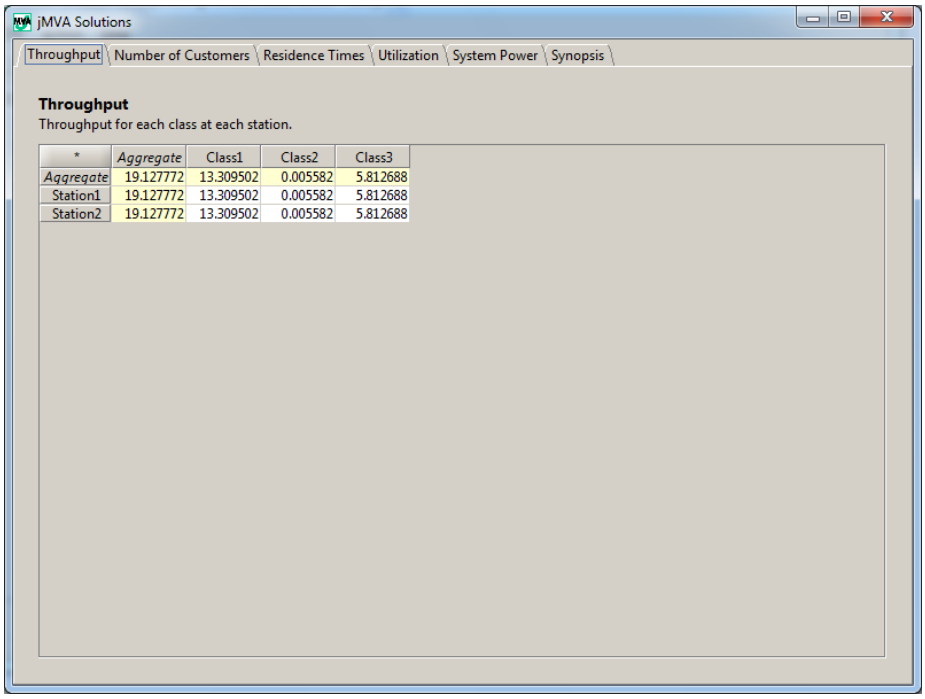


Figure 4.12: Solution window after a normal analysis

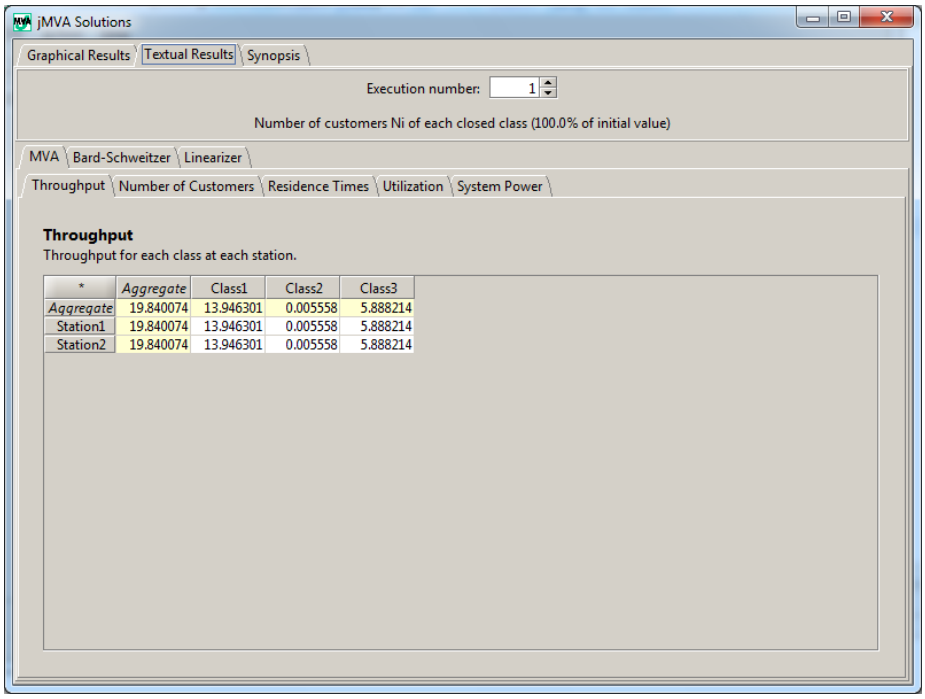


Figure 4.13: Numerical results from what-if analysis shown under 'Textual Results' tab, while graphical representation appears under 'Graphical Results' tab. The 'Execution number' parameter is the iteration number for the different values of control parameter.

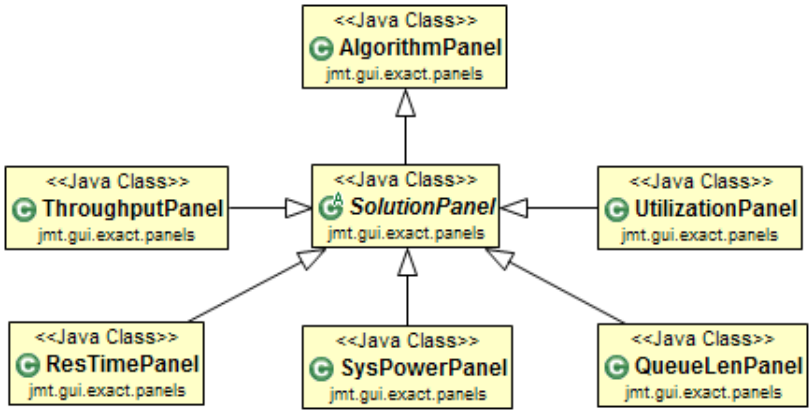


Figure 4.14: The class hierarchy of solution panels

As you may have noticed in some of the screenshots of the solution window, there is one more panel we have not discussed yet, that is the 'Synopsis' panel. This panel displays a summary of the model that was evaluated, i.e. the model parameters provided by the user. Prior to this project, the Synopsis panel displayed the class, station and service demands information. Since users

can now specify algorithms to use in addition to those parameters, this needs to be reflected in the Synopsis panel. The class responsible for this panel is `SynopsisPanel`, which essentially uses an XSLT (EXtensible Stylesheet Language Transformations) template to extract relevant data from the XML representation of the model and copy that into an XML file, conforming to the design specification in the template. The output XML file is then displayed within the Synopsis panel. So, we added an extra table to the XSLT template to show algorithm information that includes algorithm names, their respective tolerances and their iteration count (in the case of what-if analysis, the algorithm iteration count for each execution are displayed in the order the executions were performed and separated by commas). This is only done for closed models, as the algorithms are only applications to them. The result can be observed in Figure 4.15. A code snippet from the XSLT template file for including the algorithm information table is shown in Appendix B.

4.3.5 Moment Analysis

Another feature that has been added to JMVA is the calculation of mean and variance of queues. In addition to providing values of performance indices, the algorithms can also be used to compute mean and variance of queue lengths of individual service centres in a queuing model. This section uses material from Section 3.3.

For this feature, a new class was created, called `Moment`, in the analytical package. It takes a model file as input and returns the mean and variance of each queue in that model. For each service centre, it calculates the first power moment, by solving the original model. The first binomial moment is computed from the first moment and this is used to compute the second binomial moment, along with the queue length obtained from solving the model with an additional replica of the service centre in question (recall Section 3.3). From these binomial moments, second power moments are calculated using the equations in Section 3.3. Finally, from the power moments, the mean and variance are easily computed and returned. It uses MVA algorithm if there is a single class in the model, or if the number of classes and number of stations are less than four (since MVA struggles with models having classes or stations greater than four, as well as taking into consideration that we also have to solve the same model with an additional replica station, so we have to ensure both models are feasible for MVA), otherwise it uses De Souza-Muntz Linearizer, as that is the most accurate approximate algorithm and would provide closest estimates to actual mean and variance. We use De Souza-Muntz Linearizer instead of the original Linearizer, because it provides the same degree of accuracy with lower complexity.

This analysis of moments has not been integrated into JMVA GUI, however, in the future, users could be given the choice to compute mean and variance, in addition to solving the model, and the measures could be displayed in their own panel or with in the Synopsis panel along with other service centre/station characteristics. However, for the time being, they have to be computed through command-line in the following manner:

```
java -cp JMT.jar jmt.analytical.Moment sample_model_file.jmva
```

The testing for class was done by comparing the interim binomial moments

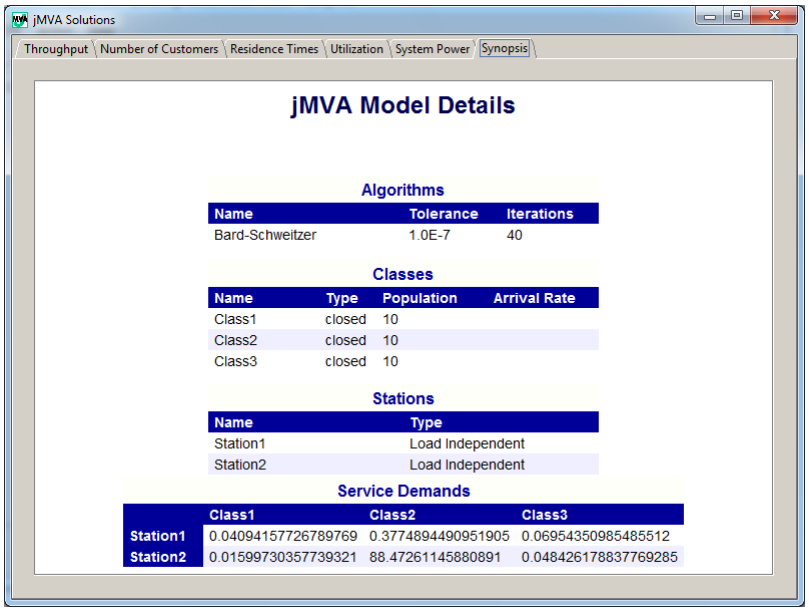
and actual results with the values provided by the supervisor for some given models.

In addition to the above use, the binomial moment generation method in `Moment` class, that is `getBinomialMethod(int[] replicas)` can be used to calculate any binomial moment, with the given replicas array, which contains a value for each service centre in the model, where the value indicates the number of replicas of service centres to be added to the model. Also, there are established methods for easily computing the first, second and third power moments, given we know the values for the first, second and third binomial moments.

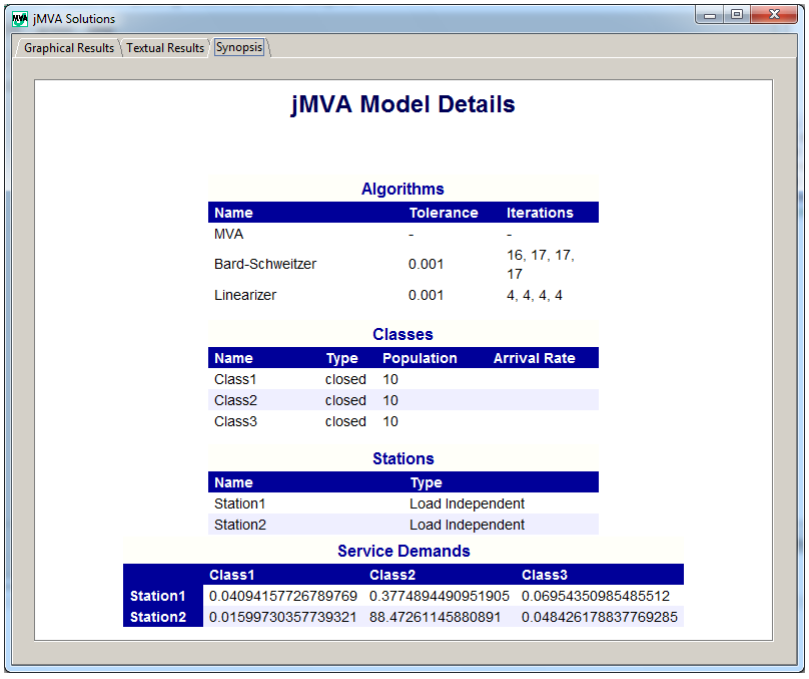
4.4 Testing and Verification

Testing and verification of program's output were very important aspects of the implementation process, as they helped in ensuring the robustness and correctness of the program. Since the algorithms added are approximate and hence only provide estimates, we could not write unit tests to compare their results with the expected values. Therefore, in order to verify that all the algorithms and their associated solvers were working correctly, system-level testing was performed which involved comparing solutions of models taken from research papers and provided by the supervisor, as well as comparison with other existing implementations of some of these algorithms written in MATLAB. In the case of calculation of mean and variance of service centres, examples were provided by the supervisor, which were then compared against the values obtained from the program. Moreover, user acceptance testing was performed thoroughly for the GUI by myself, my supervisor and other contributors who work on JMVA, to pinpoint any erratic or incorrect behaviour.

This concludes the implementation part of the report. In the next chapter, we evaluate the usefulness of the implemented algorithm solvers, and of the work done in this project.



(a) For normal analysis



(b) For what-if analysis

Figure 4.15: Addition of algorithm information in Synopsis panel

Chapter 5

Evaluation

Now that the algorithms have been implemented and fully integrated into JMVA, the next thing to consider is whether they offer any benefits over the exact techniques in practice. In order to quantify the impact of the proposed modifications and to investigate the achieved improvements in real-life scenarios, we used well-known examples given from research papers, as well as created our own a set of queuing network models.

This chapter is structured in the following manner:

- First, we consider evaluation of real application models, to show their applicability and usefulness in different real-world scenarios.
- Secondly, we show results and draw conclusions from an experimental campaign carried out to benchmark each algorithm and compare their performance against the exact algorithm.
- Lastly, we summarise the effectiveness of the approximate algorithms and the strengths and weaknesses of the work accomplished as part of this project.

5.1 Real Application Models

5.1.1 Capacity Planning of an Intranet with Multi-class Workload

Consider an IT infrastructure with a web server, an application server and three database servers[1]. We suppose that the number of customers submitting requests remains constant. This allows us to use a closed model to represent the system, for which we can use the approximate algorithms implemented earlier. Customers submit HTTP requests to the web server and wait a few seconds for the web page to load in their browser. Hence, we suppose the delay between HTTP requests is 1 second. If the submitted request is for a static page, the HTTP response from web server is immediately returned to the customer, otherwise the web server communicates with the application server, which subsequently talks to the database to perform some queries and returns the processed

[1] This example has been adapted from an example in [16]

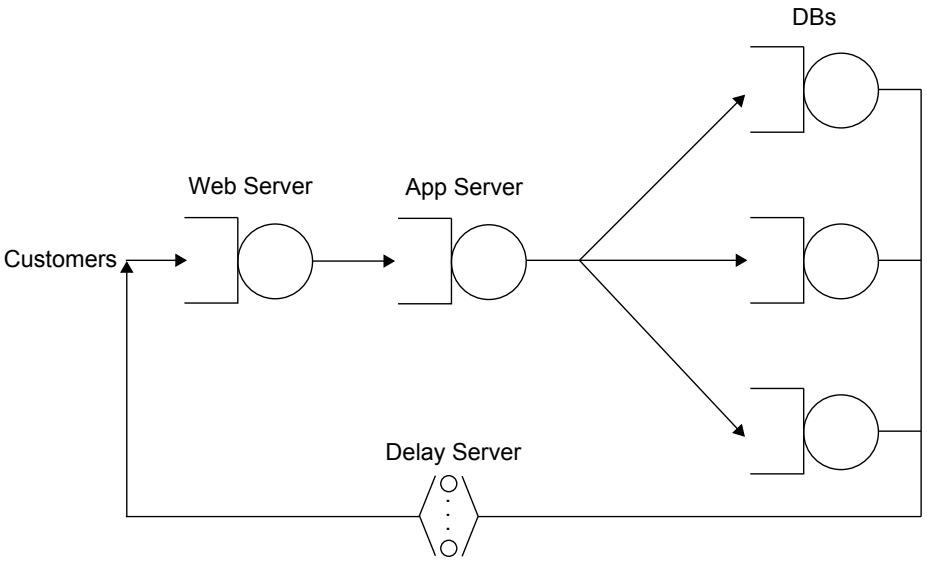


Figure 5.1: System model used for capacity planning case study

	LightLoad	*HeavyLoad*
Web Server	1.40	1.10
App Server	2.10	1.50
Database Server 1	1.10	2.90
Database Server 2	1.20	2.70
Database Server 3	1.10	2.80

Table 5.1: Service demands in milliseconds for each station and class in the system.

data is returned back to the web server. The system model is presented in Figure 5.1. Since we are examining an Intranet, we assume there are no communications delays imposed by the local area network (LAN), due the extremely high speeds offered on a LAN now-a-days.

The database servers work in parallel, and a job is assigned to them by a load-balancer in a random manner. The distribution of requests among the servers is assumed to be uniform. We also assume the load-balancer does not introduce any delay in the network.

The requests are divided into two types: request for a search query on the database and request for update query on the database. These types are modelled as classes in our queuing model, with the servers assuming the role of service centres/stations. Since a search query puts significantly less load on the database than an update query, we refer to the classes as *LightLoad* and *HeavyLoad*, respectively. The service demands for each station and each class are provided in Table 5.1.

Now, we analyse system behaviour as customer population increases. We range the number of customers from 10 to 1000 and assume the ratio between the number of *LightLoad* requests and *HeavyLoad* requests is 3/7. We used what-if analysis in JMVA to perform this analysis. In Figure 5.2, we plotted the values

of the system throughput for the range of customer populations. In addition, we solved the model using exact MVA as well as all approximate algorithms (with tolerance = 0.0000001), so that we could compare their accuracy against exact MVA. As the figure demonstrates, the system reaches saturation around 600 customers. Furthermore, Figure 5.3 shows that the system response time increases linearly after an initial phase. We can infer from these findings that a bottleneck[2] exists in the system. This is confirmed in Figure 5.4, which plots the utilisation of each station/server. It is clear that the application server reaches complete saturation around 600 customers (as its utilisation becomes 1), and constitutes the bottleneck in the system. This signifies that when the application server reaches saturation, the system throughput remains constant (as shown in Figure 5.2).

As this example demonstrates, the algorithms can be used to check for bottlenecks within a system, by modelling it with a queuing network. Also, as Figures 5.2 and 5.3 show the difference between values obtained from approximate algorithms and exact MVA is almost negligible and results are still accurate enough for an estimate, even more so for AQL, Linearizer and De Souza-Muntz Linearizer (see Figure 5.5). Since this was relatively small model, it allowed us to use exact MVA. If, however, we were to add other types of queries to the model as well, such as delete and insert queries, solving with MVA would not be feasible, at least not within any reasonable amount of time. In that case, approximate algorithms would be very effective, as they can solve the model realistically with a reasonable degree of accuracy.

5.1.2 A J2EE Application

We continue the evaluation considering a real application model which cannot be solved with Exact MVA. The results demonstrate how useful the implemented approximate algorithms can be when evaluating large (closed) models. We consider the large-scale J2EE service application modelled by Samuel Kounev and Alejandro Buchmann in [10]. We provide an overview of the model discussed in the paper. For a detailed description, please look at the paper.

The application runs on a two-tier architecture with a cluster of k_{AS} replicated application servers (AS) and a back end which consists of a dual-processor database server (DB). The workload is represented in terms of $C = 5$ classes, labelled C1–C5, which represent a new order placement or an status query. We look at the most complex architecture among the ones evaluated in [10], which has $k_{AS} = 9$ and overall $K = 12$ stations, which also include a station that represents communication overheads (CO). We assume the delay to be $Z_1 = ... = Z_4 = 2s$ for the first four classes and $Z_5 = 3s$ for the fifth class. The service demands for the stations and classes are given in Table 5.2. We evaluate the following three workload profiles:

- Low load: $\overrightarrow{N}$ = (30, 10, 50, 40, 50)
- Medium load: $\overrightarrow{N}$ = (50, 40, 100, 70, 100)
- Heavy load: $\overrightarrow{N}$ = (100, 50, 150, 50, 200)

[2]Presence of a bottleneck indicates that the a station is working at full capacity (this is the bottleneck), while the load of other stations remains limited as the number of customers increase.

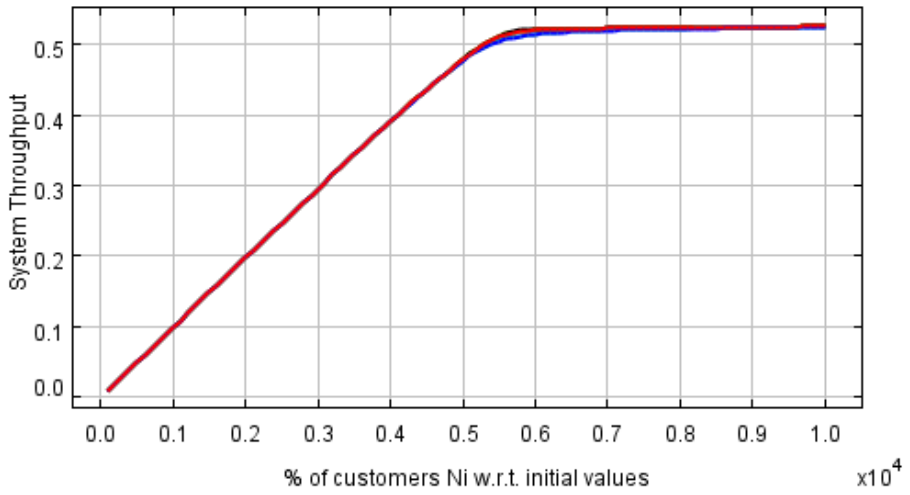


Figure 5.2: Global system throughput for the chosen range of number of customers, computed using different algorithm techniques. Red line represents results of Exact MVA, blue line represents Chow, turquoise represents Bard-Schweitzer, black represents AQL, orange represents Linearizer and green represents De Souza-Muntz Linearizer.

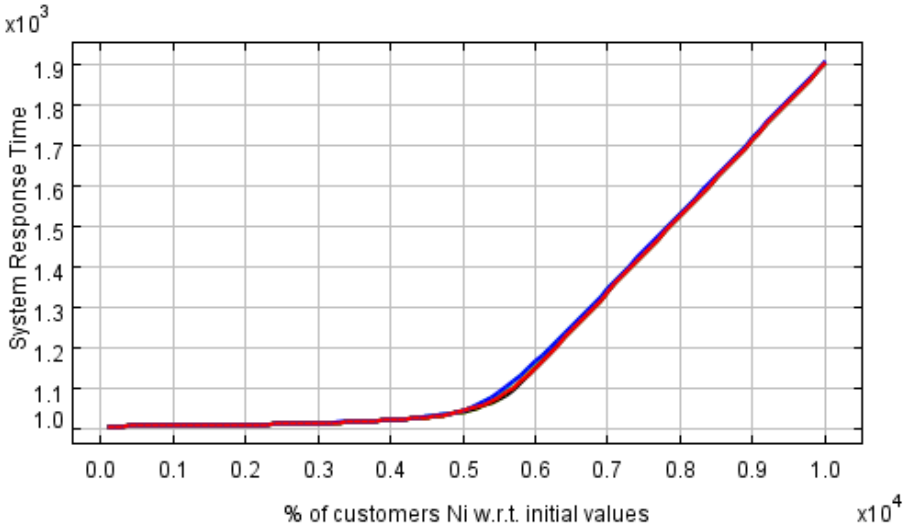


Figure 5.3: Global system response time for the chosen range of number of customers, computed using different algorithm techniques. Red line represents results of Exact MVA, blue line represents Chow, turquoise represents Bard-Schweitzer, black represents AQL, orange represents Linearizer and green represents De Souza-Muntz Linearizer.

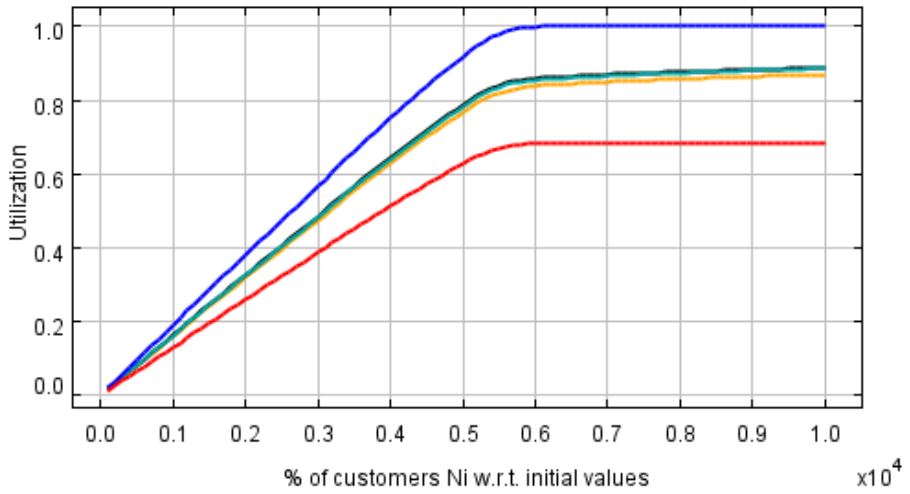


Figure 5.4: Utilisation of each station/server for the chosen range of number of customers, computed using Exact MVA. The top line represents the application server, the bottom line represents the web server and the middle three represent the database servers.

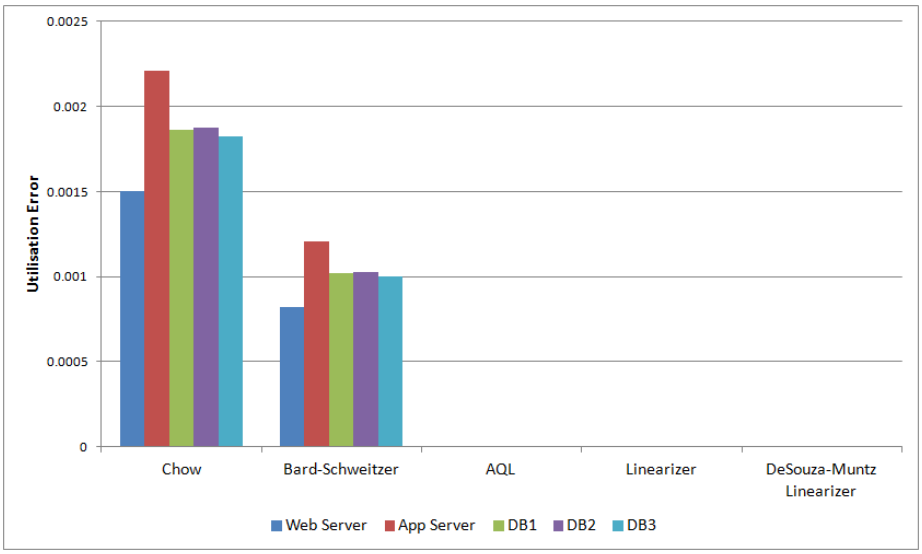


Figure 5.5: Utilisation error on each station for each approximate algorithm. Although the error in AQL, Linearizer and De Souza-Muntz Linearizer is negligible, the errors in the other two are still fairly small.

Station name	Stations	C1	C2	C3	C4	C5
AS	$k_{AS} = 9$	12.98	13.64	2.64	2.54	24.22
DB	$k_{DB} = 2$	5.32	5.18	1.24	1.04	17.07
CO	$k_{CO} = 1$	1.12	1.27	0.58	0.03	1.68

Table 5.2: Service demands $D_{c,k}$ in milliseconds of the J2EE application model.

The evaluations are performed on `batch1.doc.ic.ac.uk` machine, which has 16 cores of Intel Xeon-2.53 Ghz and 23 GB of memory. We plotted the runtimes for these models when solved with all approximate algorithms (using tolerance = 0.0000001) in Figure 5.6. The memory requirements for each individual execution was almost 2 MB, meaning as the memory consumption for these algorithms does not increase very much as population increases. Although we could not run exact MVA on these models, we obtained runtime and memory requirements for Method of Moments (MoM) algorithm[3] from [5]. Using MoM, the low load model was solved in 0.9s using 1.6 MB of memory, the medium load in 4.7s using 2.3 MB of memory and the heavy load in 14.1s using 3.4 MB of memory. Comparing these values with Figure 5.6, it is evident that the approximate algorithms are considerably quicker, and since MoM's time complexity is $\mathcal{O}(N^2)$, it would get slower as population increases, while the approximate algorithms would still be very quick, as their complexity depends on the number of stations and classes. These results further validate the applicability of approximate algorithms on models of practical interest.

5.1.3 Stress Case

The final case study we look at is one where we test the algorithms under extreme stress (i.e. very large customer population). The example is taken from [5] to simplify comparing the results, as the paper provides solutions, while also comparing time and memory requirements.

We consider a model with five distinct stations, named S1–S5, and $C = 7$ workload classes, named C1–C7, and 16,660 jobs present. The service demands are provided in Table 5.3. The populations used in the model are defined by the vector $\overrightarrow{N} = (10000, 5000, 1000, 500, 100, 50, 10)$.

On the same machine as before, we ran the approximate algorithms with tolerance = 0.0000001 (10^{-7}). The runtime, queue length of class 1 at station 1 $Q_{1,1}$, throughout of class 2 X_2 and memory requirements of all approximate algorithms and MoM are provided in Table 5.4. It is evident that the approximate algorithms are extremely quick even with vast populations. While MoM took minutes to solve the model, the approximate algorithms solved it within milliseconds with minimal memory usage and a very high degree of accuracy. The memory requirement of MVA for this model would be approximately 10^{10} GB. This case study further emphasises the key computational and memory advantage of approximate algorithms over the exact algorithms for large models and ever larger populations.

[3]Method of Moments (MoM) algorithm is another exact algorithm, which while being more complex, is not as computationally intensive as exact MVA.

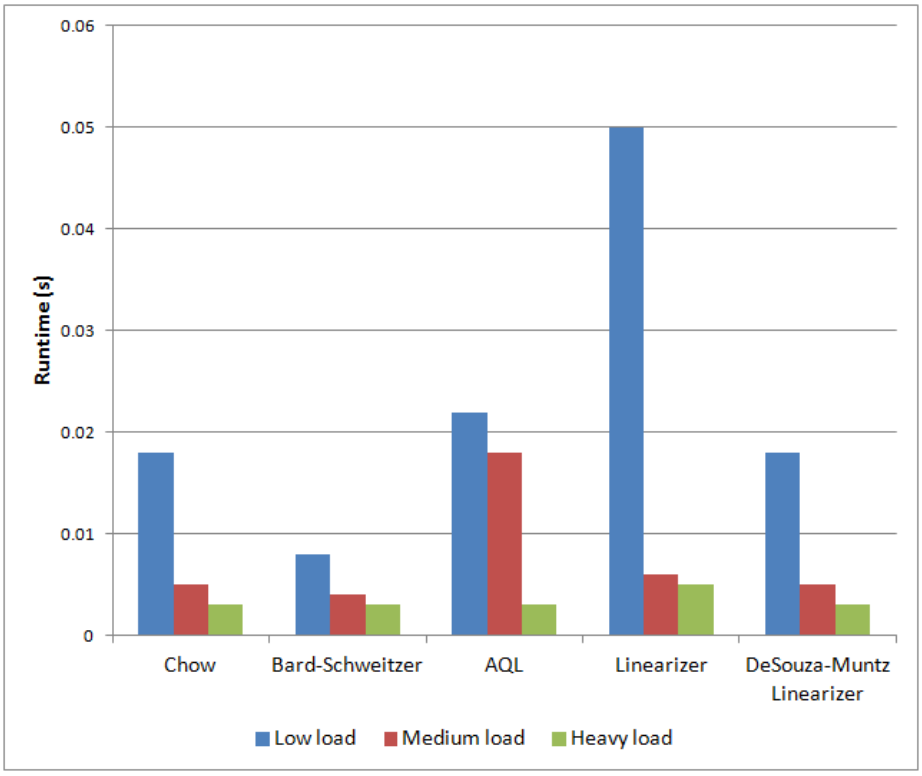


Figure 5.6: Utilisation error on each station for each approximate algorithm. Although the error in AQL, Linearizer and De Souza-Muntz Linearizer is negligible, the errors in the other two are still fairly small.

Station	C1	C2	C3	C4	C5	C6	C7
S1	64	77	93	27	19	60	8
S2	68	70	14	22	32	2	64
S3	28	2	68	34	22	47	81
S4	35	97	69	15	82	83	10
S5	10	85	88	71	62	79	98

Table 5.3: Service demands $D_{c,k}$ in milliseconds for stress case with $N = 16,660$ jobs.

	Runtime (s)	$Q_{1,1}$	X_2	*Memory (MB)*
Chow	0.003	9982.207529	0.00389701	1.89
Bard-Schweitzer	0.005	9982.208475	0.003897244	1.89
AQL	0.004	9982.23135	0.003897244	1.89
Linearizer	0.024	9982.231584	0.003897244	3.78
De Souza-Muntz Linearizer	0.01	9982.231584	0.003897244	3.78
MoM	409	9982.23	0.00389724	407

Table 5.4: Service demands $D_{c,k}$ in milliseconds for stress case with $N = 16,660$ jobs.

5.2 Experimental Evaluation

To benchmark the performance of the approximate algorithms, we carried out an experimental campaign on the same machine as before. The comparison is done against the MVA algorithm because for large populations, it is typically the best performing among existing methods, although inefficient.

We consider in this section simple models with no more than $C = 4$ classes, for which existing methods can be feasible under large populations[4]. The number of queues/stations (K) used is 3 and it is kept constant, while the number of classes ranges from 2 to 4. Several workload profiles are defined, corresponding to super-low, low, medium and heavy load for each model. The load represents the number of jobs circulating the system. The populations assigned to each profile is 100, 300, 500 and 700, respectively. These populations are split equally among the classes (i.e. $N_c = N/C$), with rounding to the nearest integer. A constant tolerance of 0.0000001 (10^{-7}) is used for all approximate algorithms. The measures used for comparing performance of approximate algorithms include runtime and queue length tolerance error, which is defined as

$$\max_{c,k} \frac{|Q_{c,k} - Q*_{c,k}|}{N_c}$$

where $Q_{c,k}$ is the approximate class c queue length at centre k, $Q*_{c,k}$ is the exact value, and N_c is the class c population. Tolerance error is used instead of relative error because the latter measure is very sensitive to small errors in small values, although these errors have negligible impact on our notion of overall accuracy [17].

5.2.1 Queue Length Tolerance Error

Now, we compare the queue length tolerance error percentage of the approximate algorithms obtained from the experimental campaign. From Figures 5.7–5.10, we can deduce that the approximate algorithms scale very well to large populations as the error percentage goes down as populations increase. The figures also confirm our hypothesis that Chow is the least accurate algorithm and Linearizer the most accurate algorithm, among the ones implemented, as well as

[4]Beyond four classes, exact MVA is too computationally expensive to be solved within any reasonable amount of time.

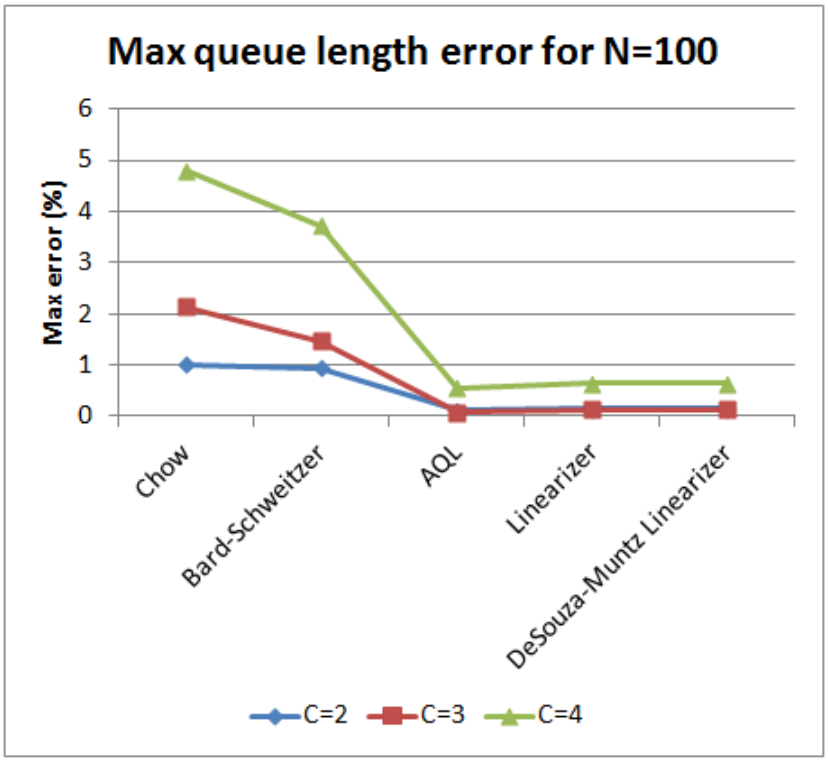


Figure 5.7: Maximum queue length tolerance error percentage when evaluating model, using different algorithms, with total population (N) = 100 and $K = 3$ queues for varying number of classes, C.

the fact that Linearizer and De Souza-Muntz Linearizer should effectively have the same results. In fact, for AQL, Linearizer and De Souza-Muntz Linearizer, the error remains below 0.5% for all populations and even becomes negligible for the model with population $N = 700$. These algorithms demonstrate great potential for providing very accurate estimations, while Chow and Bard-Schweitzer mainly good for a quick analysis (due to their lower complexity), where shorter length of execution is desired over accuracy. Although, as you will see in the next section, the runtime between the approximate algorithms does not differ by much, meaning AQL and Linearizer are not actually that expensive and in most circumstances, would be worth sacrificing a second for better precision.

5.2.2 Runtime

Now, we compare the runtime of different algorithms obtained from the experimental campaign. As Figures 5.11–5.14 demonstrate, there is a significant jump in runtime from $C = 3$ to $C = 4$ with MVA algorithm, which is made even more apparent with larger populations. This implies that the MVA algorithm does not scale well with increase in number of classes and populations. However, all the approximate algorithms barely notice any difference in runtime as number of classes and populations increase. Hence, they will be more suitable for large models, especially those where number of classes is greater than 4. Figure 5.15

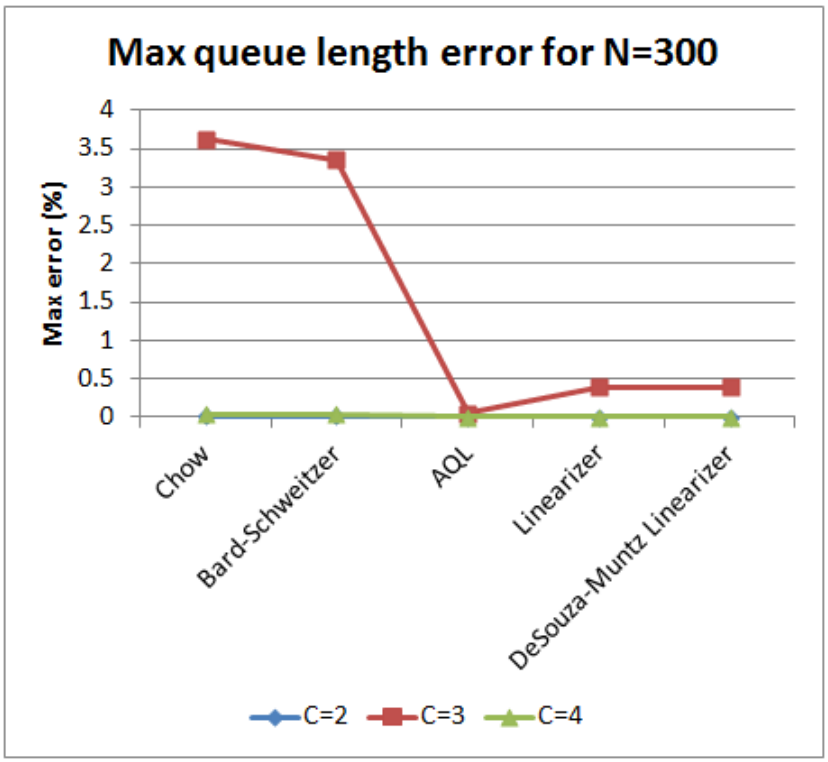


Figure 5.8: Maximum queue length tolerance error percentage when evaluating model, using different algorithms, with total population (N) = 300 and $K = 3$ queues for varying number of classes, C.

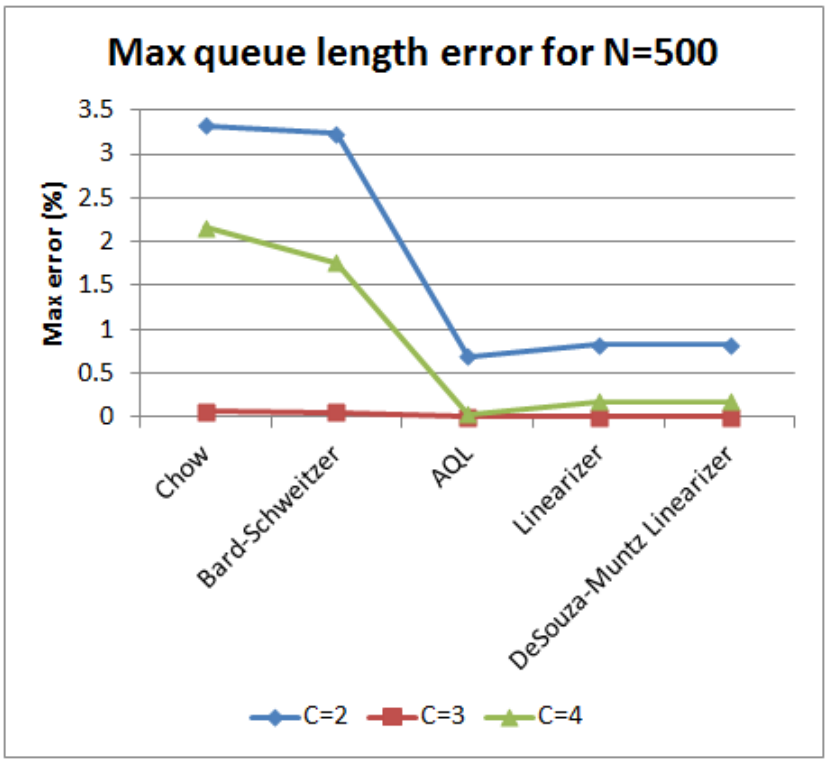


Figure 5.9: Maximum queue length tolerance error percentage when evaluating model, using different algorithms, with total population (N) = 500 and $K = 3$ queues for varying number of classes, C.

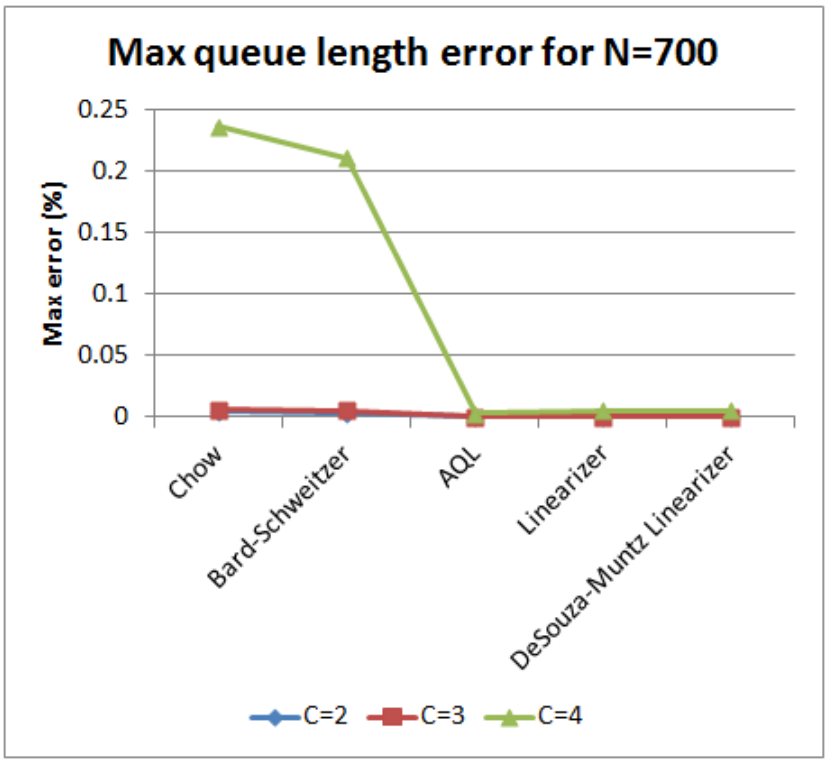


Figure 5.10: Maximum queue length tolerance error percentage when evaluating model, using different algorithms, with total population (N) = 700 and $K = 3$ queues for varying number of classes, C.

shows the runtime for different algorithms for the workloads with five stations and five classes. Even though MVA cannot solve this model, the approximate algorithms can and we also notice that the runtime of algorithm decreases as total population increases, confirming that these algorithms scale well to increase in population.

With the significantly low runtimes and low error percentages even for large populations, we can conclude that these approximate algorithms are far more scalable than MVA algorithm and since real-life models are usually quite big and complex, the approximate algorithms are a much better and effective solution than exact MVA algorithm. It means that practically anyone can run approximate algorithms on their systems, without the need for very powerful systems, due to them being computationally light in comparison to the exact techniques.

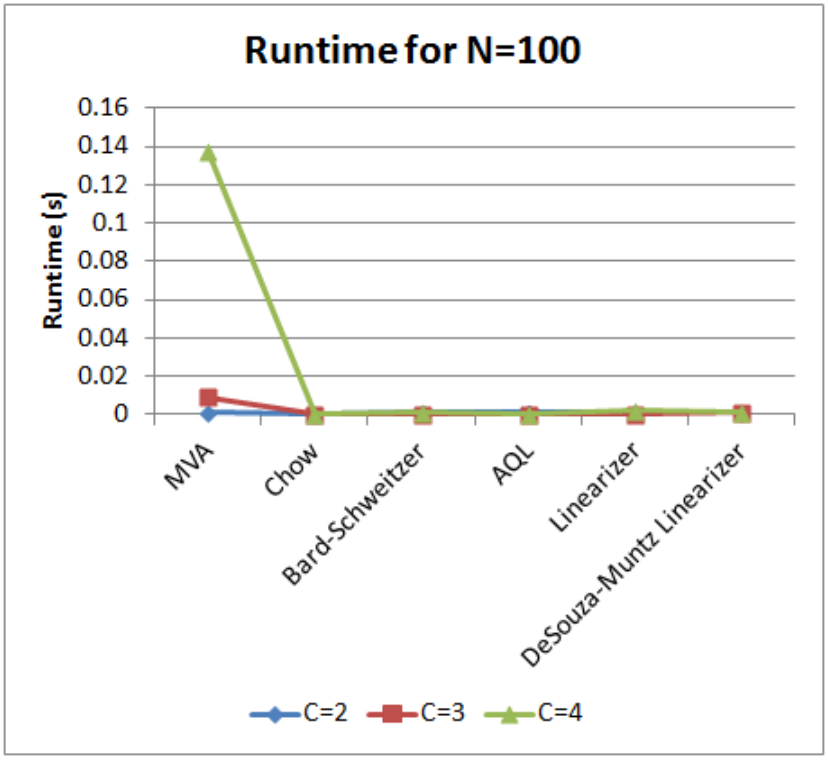


Figure 5.11: Total time needed to evaluate model, using different algorithms, with total population (N) = 100 and $K = 3$ queues for varying number of classes, C.

5.3 GUI Evaluation

Having talked about the quantitative aspects of evaluation, we briefly discuss the qualitative aspects. These aspects were covered as part of the user acceptance testing. It mainly involved thoroughly testing the GUI of the final product by myself and others. To test the usability aspect of the final product, feedback

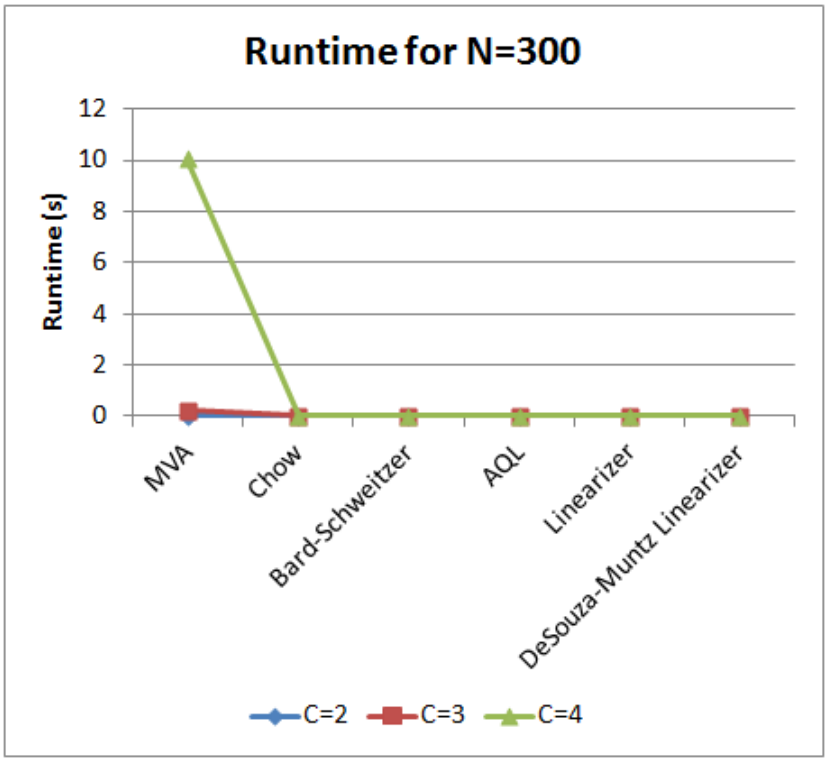


Figure 5.12: Total time needed to evaluate model, using different algorithms, with total population (N) = 300 and $K = 3$ queues for varying number of classes, C.

was be obtained from a group of people familiar with the subject (for example, the supervisor and other JMVA contributors) regarding the ease of use of the new features and how easy, self-contained and self-explanatory they are. In addition, the program was tested with several different inputs (valid and invalid) and workloads to observe its behaviour. Moreover, the behaviour of algorithms was observed, such as in terms of accuracy of algorithms, the order we get is Chow, Bard-Schweitzer, AQL, Linearizer and De Souza-Muntz Linearizer (from least to most accurate), and Linearizer and De Souza-Muntz Linearizer provide similar results. The feedback proved very helpful in attaining a correct and robust program.

5.4 Strengths and Weaknesses

The strengths of the work accomplished as part of this project include:

- Complete and seamless integration of the approximate algorithms within JMVA.
- Simple, intuitive interface for choosing between different algorithms.
- Ability to compare results of all algorithms (including MVA) against each other graphically, as well as numerically.

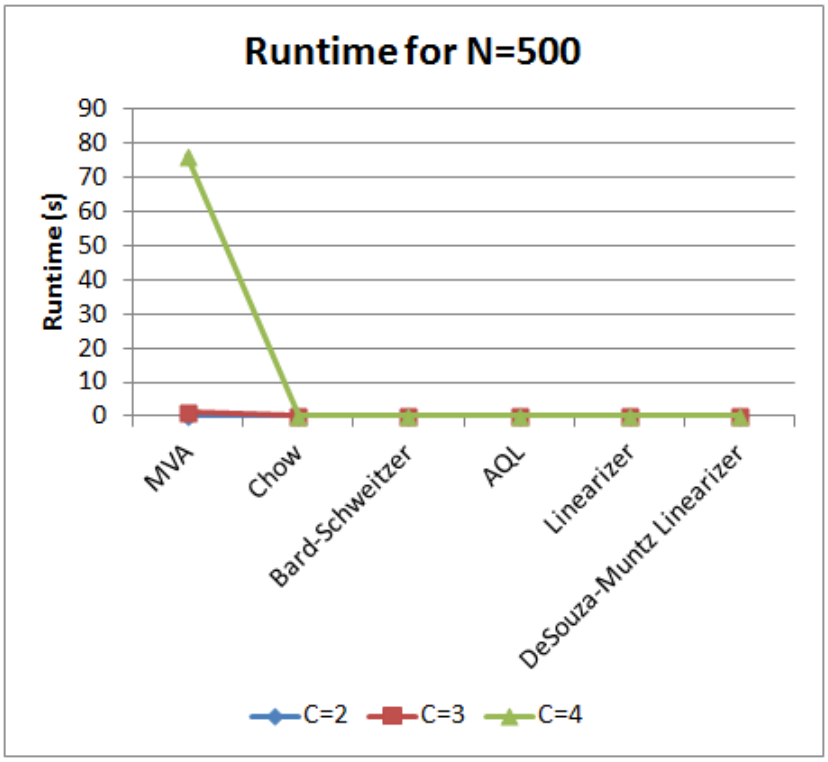


Figure 5.13: Total time needed to evaluate model, using different algorithms, with total population (N) = 500 and $K = 3$ queues for varying number of classes, C.

- Saving of algorithm preferences and their results when a model is saved, and reloading of them upon opening a model.
- Simple, command-line execution of moment analysis for stations in a model to compute their mean and variance.
- Simplicity of adding new algorithms to the system, due to the designed algorithm API.

Although majority of the project was a success, there are some limitations associated with it, as listed below:

- We initially planned to incorporate another exact algorithm called Method of Moments (MoM) (which is not as computationally intensive as MVA) into JMVA, using its implementation by Michail Makaronidis [13]. However, due to issues with his implementation, it had to be removed. Since it can solve even large models, unlike MVA, it would have been a useful addition. Nevertheless, adding new algorithms to the system has been made extremely simple as a result of this project, so if an implementation of MoM or any other algorithm needs to be included, the process would not be as too tedious or time-consuming.

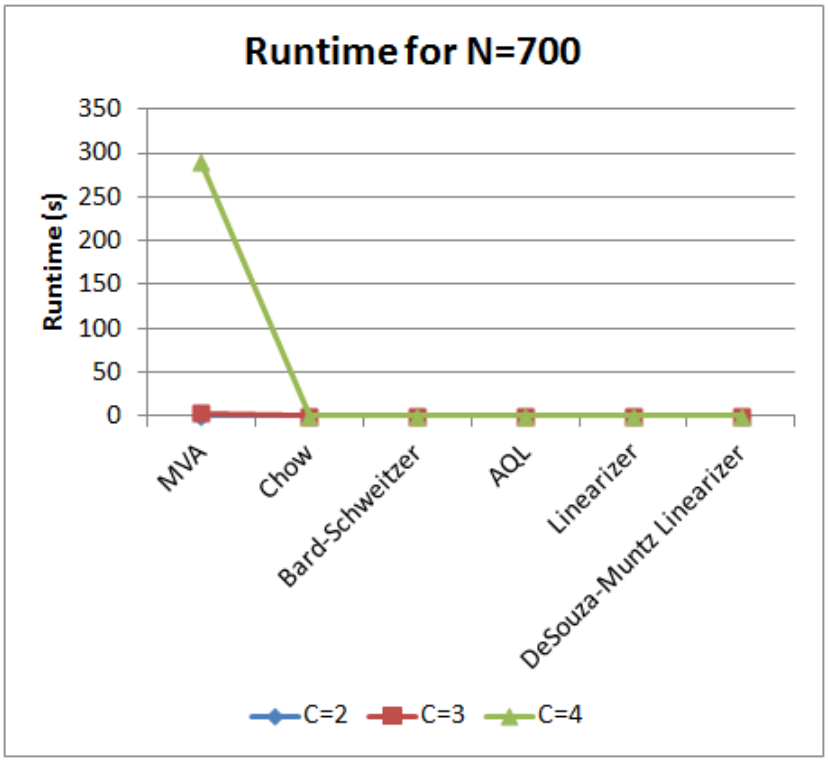


Figure 5.14: Total time needed to evaluate model, using different algorithms, with total population (N) = 700 and $K = 3$ queues for varying number of classes, C.

- As moment analysis has not been incorporated into the GUI yet, users are required to use command-line, which all users may not be fully familiar with. Also, since it takes a model file as input, the model has to be created in JMVA first and saved before we can run moment analysis, which makes the process a little tedious. However, this also makes it more flexible, since we do not need to write our own command-line parser, as we can just use the one we implemented for JMVA and also, the model only needs to be created once in JMVA and can be used with both JMVA and moment analysis.

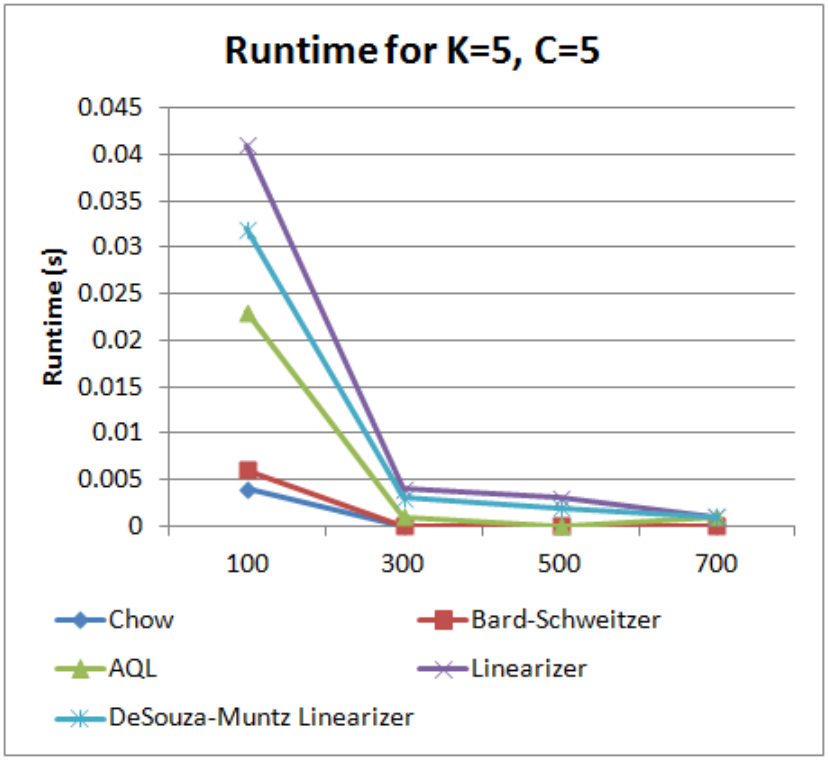


Figure 5.15: Total time needed to evaluate model, using different algorithms, with $K = 5$ queues and $C = 5$ classes for varying total population, N. Although MVA cannot be run on this model, we can still run approximate algorithms with quick evaluations.

Chapter 6

Conclusion

We have presented a library of approximate algorithms, which are used for solving closed queuing network models and are very well-suited to solving large models. These have been implemented and designed in a way to provide seamless integration with the JMVA tool. The design of implemented algorithm API is flexible and makes it easy to include new algorithms or new implementations of existing ones in JMVA. We introduced clear, intuitive ways for users to make use of these algorithms in the evaluation of models and to compare the results of algorithms against results from other algorithms and MVA numerically and graphically. The latter allows user to visually see the difference in accuracy between the approximate algorithms, which may help them decide which algorithm to use for future models.

The most important conclusions drawn from the evaluation of these algorithms was that they provide considerable improvements over MVA, in terms of runtime and memory usage. Although there is a small price of accuracy to pay with Chow and Bard-Schweitzer algorithms, AQL, Linearizer and De Souza-Muntz Linearizer make up for it with nearly accurate estimates (with an error percentage of below 0.5%). We also found that although the approximate algorithms are not as accurate for small populations, they scale extremely well to large models and populations, which is something MVA does not offer. As complexity of AQL, Linearizer and De Souza-Muntz Linearizer depends primarily on the number of classes and stations in the network, we believe they have a clear advantage over Chow and Bard-Schweitzer, considering the practical models these are used with will generally have very large populations (which we already know these algorithms scale well for), but still not as many stations and classes to have a negative impact on runtime. Also, as we found in the stress case evaluation, even with other exact techniques which can solve large models, these algorithms still far surpass them, on the subject of runtime, without sacrificing hugely on accuracy, meaning they are perfect for quick model evaluations, where exact solutions are not necessarily needed. Since these algorithms are not very CPU- or memory-intensive, it means that these could practically be run on any machines, avoiding the need for extremely powerful machines which would be required for MVA.

After the successful integration of these algorithms in JMVA, we focussed our attention to moment analysis, which makes use of these algorithms (Linearizer to be specific). After researching about moments and finding out about their

significance in queuing networks (recall 3.3), we noticed how easily we could compute them using MVA (for small models) and the implemented approximate algorithms (for other models). These provide important characteristics about station queue lengths, such as mean and variance, which can be helpful in better understanding their distribution. Regrettably, we did not have the chance to explore this topic further to establish a way of computing marginal probabilities of stations.

6.1 Future Work

Although not possible for this project, Method of Moments (MoM) algorithm could be included in JMVA in the future. It is another exact algorithm, which solves queuing networks by recursively computing a set of higher order moments of the stations' queue length distributions, using a a system of linear equations involving normalising constants. It provides drastic improvement in terms of cost for an exact analysis compared to the MVA algorithm and does not scale as badly as MVA for bigger models and thus, would be a useful addition to JMVA and its users who require exact solutions. Since it can be parallelised with multi-threads, it could provide even better performance.

Additionally, we could allow users to set maximum number of iterations, along with tolerance, as termination criteria for approximate algorithms in JMVA. This would allow to use either or both measures, as some users might prefer one over the both. The APIs for this have already been established in the algorithm solvers, so only the option in the GUI needs to be added.

Lastly, the work on moment analysis could be extended to include calculation of marginal probabilities (i.e. probability that there are n jobs at a station k) for stations from binomial and power moments. The formulas connecting marginal probabilities with binomial and power moments were provided in section 3.3. As we already covered calculation of moments in this project, we could use these values to get a system of linear equations, which could be solved with Simplex algorithm to obtain a solution for the marginal probabilities. These could then be incorporated into JMVA, along with moment analysis, to provide a simple, intuitive way for users to compute moments and marginal probabilities of stations in a model. Furthermore, other moments, such as skewness, could also be added to moment analysis.

Bibliography

[1] *Java Modelling Tools – Users Manual*, November 2011.

[2] S. Balsamo. Product form queueing networks. *Lecture Notes in Computer Science*, pages 377–402, 2000.

[3] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. Jmt: performance engineering tools for system modelling. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):10–15, March 2009.

[4] Giuliano Casale. On single-class load-dependent normalizing constant equations. *Proc. of QEST*, pages 333–342, September 2006.

[5] Giuliano Casale. Exact analysis of performance models by the method of moments. *Performance Evaluation*, 68(6):487–506, June 2011.

[6] K. Mani Chandy and Doug Neuse. Linearizer: a heuristic algorithm for queueing network models of computing systems. *Communications of the ACM*, 25(2):126–134, February 1982.

[7] We-Min Chow. Approximations for large scale closed queueing networks. *Performance Evaluation*, 3(1):1–12, 1983.

[8] E. de Souza e Silva and R. R. Muntz. A note on the computational cost of the linearizer algorithm for queueing networks. *IEEE Transactions on Computers*, 39(6):840–842, June 1990.

[9] Armin Heindl and Appie van de Liefvoort. Moment conversions for discrete distributions. *Proc. of PMCCS-6 Workshop*, 2003.

[10] Samuel Kounev and Alejandro Buchmann. Performance modeling and evaluation of large-scale j2ee applications, 2003.

[11] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.

[12] Edward D Lazowska, John Zahorjan, and Kenneth C Sevcik. Computer system performance evaluation using queueing network models. *Annual review of computer science*, 1:107–137, 1986.

[13] Michail A. Makaronidis. Efficient analysis of it sizing models. Master's thesis, Imperial College London, 2010.

[14] Georgios Poullaides. Algorithms for computer system performance analysis. Master's thesis, Imperial College London, 2011.

[15] Paul J. Schweitzer. Approximate analysis of multi-class closed networks of queues. *Proceedings of International Conference on Stochastic Control and Optimization*, pages 25–29, 1979.

[16] Giuseppe Serazzi. *Performance Evaluation Modelling with JMT: learning by examples*. Politecnico di Milano - DEI, June 2008.

[17] John Zahorjan, Derek L. Eager, and Hisham M. Sweillam. Accuracy, speed, and convergence of approximate mean value analysis. *Performance Evaluation*, 8(4):255–270, August 1988.

Appendix A

Sample model file

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <model xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="JMTmodel.xsd">
3     <parameters>
4        <classes number="1">
5           <closedclass name="Class1" population="5"/>
6        </classes>
7        <stations number="1">
8           <listation name="Station1" servers="1">
9              <servicetimes>
10                <servicetime customerclass="Class1">
                      0.027385234208403157</servicetime>
11             </servicetimes>
12             <visits>
13                <visit customerclass="Class1">1.0</visit>
14             </visits>
15          </listation>
16       </stations>
17    </parameters>
18    <algParams>
19       <algType name="Bard-Schweitzer" tolerance="1.0E-7"/>
20       <compareAlgs value="true">
21          <whatIfAlg index="0" name="MVA" tolerance="1.0E-7" value="
                1"/>
22          <whatIfAlg index="1" name="Chow" tolerance="0.01" value="1
                "/>
23          <whatIfAlg index="2" name="Bard-Schweitzer" tolerance="1.0
                E-7" value="0"/>
24          <whatIfAlg index="3" name="AQL" tolerance="1.0E-7" value="
                0"/>
25          <whatIfAlg index="4" name="Linearizer" tolerance="1.0E-7"
                value="0"/>
26          <whatIfAlg index="5" name="De Souza-Muntz Linearizer"
                tolerance="1.0E-7" value="0"/>
27       </compareAlgs>
28    </algParams>
29    <whatIf className="Class1" type="Customer Numbers" values="
          5.0;6.0"/>
30    <solutions algCount="2" iteration="0" iterationValue="5.0" ok="
          true" solutionMethod="analytical whatif">
31       <algorithm name="MVA">
32          <stationresults station="Station1">
33             <classresults customerclass="Class1">
```

```
34                  <measure meanValue="5.0" measureType="Number of
                        Customers" successful="true"/>
35                  <measure meanValue="36.516028761702174" measureType=
                        "Throughput" successful="true"/>
36                  <measure meanValue="0.1369261710420158" measureType=
                        "Residence time" successful="true"/>
37                  <measure meanValue="1.0" measureType="Utilization"
                        successful="true"/>
38              </classresults>
39           </stationresults>
40        </algorithm>
41        <algorithm iterations="1" name="Chow">
42           <stationresults station="Station1">
43              <classresults customerclass="Class1">
44                  <measure meanValue="5.0" measureType="Number of
                        Customers" successful="true"/>
45                  <measure meanValue="30.43002396808515" measureType="
                        Throughput" successful="true"/>
46                  <measure meanValue="0.16431140525041893" measureType
                        ="Residence time" successful="true"/>
47                  <measure meanValue="0.8333333333333334" measureType=
                        "Utilization" successful="true"/>
48              </classresults>
49           </stationresults>
50        </algorithm>
51     </solutions>
52
53     ...
54     (solutions from other iterations)
55
56 </model>
```

Listing A.1: A sample input model file with the newly added elements. `algParams` element stores the algorithm preferences, and solutions element the values of performance indices for the model under different algorithms.

Appendix B

XSLT template file for Synopsis panel

```
<!--creates algorithm table-->
<xsl:template match="algParams" mode="description">
   <!-- Check that no open classes exist -->
   <xsl:if test="not($model/parameters/classes/openclass)">
      <table class="param" cellspacing="0">
         <tr><th class="paramtitle" colspan="3">
            Algorithms
         </th></tr>
         <tr class="paramhead">
            <td>Name</td>
            <td>Tolerance</td>
            <td>Iterations</td>
         </tr>
         <xsl:for-each select="$model/algParams/compareAlgs">
            <xsl:choose>
            <!-- Check results are from a what-if analysis and if
                 compare algorithms option was selected -->
            <xsl:when test="$model/whatIf and @value='true'">
               <!-- Go through each selected algorithm and display
                    its name, tolerance and iterations -->
               <xsl:for-each select="$model/algParams/compareAlgs/
                    whatIfAlg">
                  <xsl:if test="@value = 1">
                     <tr>
                        <xsl:attribute name="class">
                           <xsl:if test="position() mod 2=0">line2<
                                /xsl:if>
                           <xsl:if test="position() mod 2=1">line1<
                                /xsl:if>
                        </xsl:attribute>
                        <td width="{$alg-name-width}"><xsl:value-of
                             select="@name"/></td>
                        <xsl:choose>
                           <xsl:when test="@name = 'MVA'"><td width
                                ="{$alg-tol-width}"></td></xsl:when
                                >
                           <xsl:otherwise><td width="{$alg-tol-
                                width}"><xsl:value-of select="
                                @tolerance"/></td></xsl:otherwise>
                        </xsl:choose>
```

```
31                        <xsl:choose>
32                           <!-- Put a hyphen in iterations column
                                  for MVA -->
33                           <xsl:when test="@name = 'MVA'"><td width
                                ="{$alg-iter-width}">-</td></
                                xsl:when>
34                           <xsl:otherwise>
35                              <td width="{$alg-iter-width}">
36                                 <!-- For other algorithms, go
                                       through each what-if execution
                                        and find algorithm
37                                  iterations and display them in
                                       this column, separated by
                                       commas -->
38                              <xsl:for-each select="key('k1', @name
                                    )">
39                                 <xsl:value-of select="
                                       @iterations"/>
40                                 <xsl:if test="position()!=last
                                       ()">
41                                    <xsl:text>, </xsl:text>
42                                 </xsl:if>
43                              </xsl:for-each>
44                           </td>
45                        </xsl:otherwise>
46                     </xsl:choose>
47                  </tr>
48               </xsl:if>
49            </xsl:for-each>
50         </xsl:when>
51         <xsl:otherwise>
52            <!-- For a normal analysis, or a what-if analysis
                   without comapre algorithms selected,
53             get the value from the algorithm box -->
54            <xsl:for-each select="$model/algParams/algType">
55               <tr>
56                  <xsl:attribute name="class">
57                     <xsl:if test="position() mod 2=0">line2</
                           xsl:if>
58                     <xsl:if test="position() mod 2=1">line1</
                           xsl:if>
59                  </xsl:attribute>
60                  <td width="{$alg-name-width}"><xsl:value-of
                        select="@name"/></td>
61                  <xsl:choose>
62                     <xsl:when test="@name = 'MVA'"><td width="
                           {$alg-tol-width}">-</td></xsl:when>
63                     <xsl:otherwise><td width="{$alg-tol-width}"
                           ><xsl:value-of select="@tolerance"/></
                           td></xsl:otherwise>
64                  </xsl:choose>
65                  <xsl:choose>
66                     <xsl:when test="@name = 'MVA'"><td width="
                           {$alg-iter-width}">-</td></xsl:when>
67                     <xsl:otherwise>
68                        <td width="{$alg-iter-width}">
69                        <xsl:for-each select="key('k1', @name)">
70                              <xsl:value-of select="@iterations"
                                    />
71                              <xsl:if test="position()!=last()">
72                                 <xsl:text>, </xsl:text>
73                              </xsl:if>
```

```
74                                    </xsl:for-each>
75                                </td>
76                            </xsl:otherwise>
77                        </xsl:choose>
78                    </tr>
79                </xsl:for-each>
80            </xsl:otherwise>
81            </xsl:choose>
82        </xsl:for-each>
83    </table>
84  </xsl:if>
85 </xsl:template>
```

Listing B.1: The code added to the XSLT template file to display the algorithm information mentioned in Section 4.3.4, i.e. algorithm names, tolerances and algorithm iterations.