MASTER THESIS

PERFORMANCE ANALYSIS FOR EMBEDDED SOFTWARE DESIGN

SJORS HETTINGA

EWI
DESIGN AND ANALYSIS OF COMMUNICATION SYSTEMS (DACS)

EXAMINATION COMMITTEE
dr. Anne Remke
prof. dr. ir. Boudewijn Haverkort
dr. ir. Jan Broenink

UNIVERSITY OF TWENTE.

10-12-2010

Abstract

To reduce the number of implementation errors in the integration phase of a software project, new software development methods are required. The Analytical Software Design (ASD) method of the company Verum provides design strategies combined with model-checking to reduce the number of errors made when developing software.

Currently there is no insight into the performance of a software system, before the implementation phase of a project has been finished. When a system does not meet the performance requirements after integration, improving the software speed in this stage of the project is a very expensive and time consuming process. To provide early stage insight into the performance of a software system, adequate performance models and corresponding analysis tools are needed.

In this thesis a start is made by modelling and analysing the performance of the software generated by the ASD suite. A hierarchical approach is used to model the whole ASD generated software. Queuing theory is applied to model the blocks, that are generated by the ASD suite. Waiting time propagation then composes the model for the whole system from the individual blocks. The focus of this thesis is on making accurate models, with scalable analysis techniques.

Single ASD blocks are modelled as queueing stations and a case study shows that the analysis results match the simulation results accurately. To model the dependencies between different blocks, phase type distributions are used. Also systems composed of multiple blocks are analysed and validated using simulation. The analytic results for larger systems deviate substantially from the simulation, because some dependencies are left out of the model. Recommendations are given to improve the quality of the results in future research. One way to do this, could be to embedded these dependencies into the model by also using phase type distributions for elements that are currently modelled as negative exponential distributions.

Contents

1 Introduction

Philips has been making X-Ray systems since 1933. A lot of developments have taken place since 1933, making X-ray useful for applications exceeding the initially limited photographing of broken bones.

Philips Healthcare (PHC) is a division of Philips and is specialized in medical applications. One of their current products is the *interventional X-Ray (iXR)* system, which is constantly developed to improve and extend its use.

A picture of a current model iXR machine is shown in Figure 1.1. This machine consists of two connected movable arms and a table on which the patient is lying. On the movable arm the X-ray tube and detector are mounted. Besides the original photos, this system can even produce X-ray movies while it rotates the arm to get a clear picture from different angles.

In the Allegio project the software of the iXR systems of Philips Healthcare (PHC) and especially the software of the cardio vascular (CV) systems will be the topic of a case study.

These cardio vascular systems are X-ray systems, used mostly for angioplasty procedures. This procedure reopens obstructed blood vessels, which for example can cause a heart attack. During the procedure doctors insert a tool into the body at a vessel in the groin. This tool is moved through the blood vessels to the place of the obstruction. Doctors use X-ray to track the movement of the tool within the patient's body and to check if the procedure has been successful. This method allows an obstructed vessel to be opened without invasive surgery.

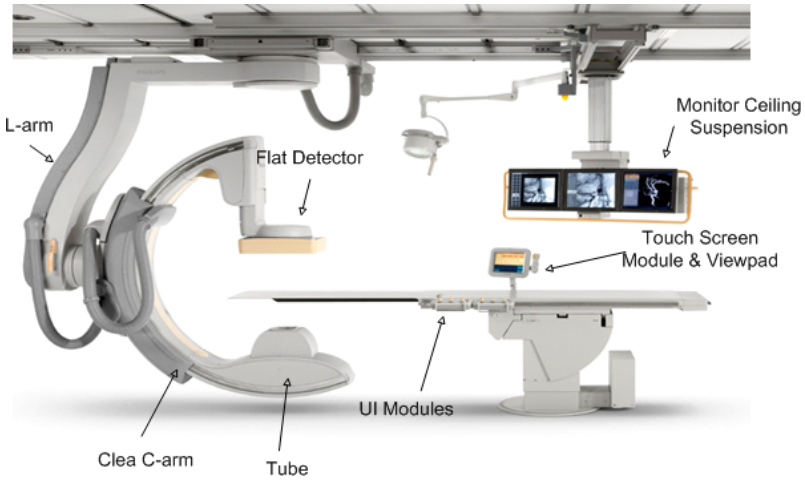


Figure 1.1: The iXR machine

The market demands a constant extension of the features of CV systems, making the systems increasingly complex. Possible new options are the automated tilting of the table as part of the image acquisition process. The growing complexity to increases the development time of new versions, while the market requires the "time to market" to become shorter.

1.1 Allegio project

Philips HealthCare (PHC) and the Embedded Systems Institute (ESI) initiated the Allegio project to find methods to handle the increasing complexity in projects and to shorten the time to market for their products. The partners in the Allegio project are the Embedded Systems Institute (ESI), Philips Healthcare (PHC), Verum, the University of Twente, Delft University of Technology and Eindhoven University of Technology. From the University of Twente the research

groups DACS[1] and DE[2] are involved.

The purpose of the Allegio project is the development and evaluation of new methods to develop software for products. For a better understanding of the problem, an overview is given of the current way of developing new products.

The development of a new product consists of the following four phases:

Requirement definition In this phase the requirements for the new product have to be listed.

Design space exploration Before actually implementing the product, different ways of implementation have to be evaluated.

System design and implementation The design is split into parts, which are implemented individually.

Integration and testing All parts are put together to form the complete system. After the integration is done, the complete system is tested.

The increasing complexity of the software leads to an increasing percentage of the project time being spend in the testing and integration phase. Without reducing the number of errors per 1000 lines of code and with systems becoming more complex, more problems manifest during the integration phase. This complexity makes fixing errors more difficult and makes it harder to find all bugs and errors. To put this trend to a halt, PHC intends to adapt their design and implementation methods to shorten the test and integrating time.

To achieve this goal, PHC wants to adopt the Analytical Software Design (ASD) method of the company Verum. Verum claims that their structured design method with build-in model-checking, reduces the number of errors made by programmers. This method saves time on bug hunting, shortening test and integration phase. The implementation phase is also shortened a little, because in this phase parts are already tested and bugs are fixed. In the Allegio project several aspects of the ASD method will be investigated.

Both the current design method and the ASD method do not take performance into account. However performance issues can also have a major impact on the duration of the software development. Performance is part of the requirements of a software system, but is only taken into account roughly in the design and implementation phase. Only in the integration phase the performance can be checked with the requirements again.

1.2 Topic of this thesis

After combining all the software parts in the integration phase, for the first time in the project the performance of the complete system can be analysed. If the performance does not meet the requirements, modifications in this phase of the project are very time consuming, because the software is already fully implemented and some parts may have to be rebuild from scratch again. The problem localization and correction can lengthen the integration process dramatically and disrupt the whole project planning.

To notice performance problems before the integration phase, an early stage performance evaluation is necessary. The contribution of the DACS group to the Allegio project will be to develop models that can predict the performance of the generated software in an early stage of the project. In this master thesis a start is made by analysing the performance of software generated using the ASD method.

Performance of software is usually expressed in responsiveness, i.e. how quickly the software reacts to different stimuli. In case of CV systems such a stimulus could be, e.g. the doctor asking

[1] DACS = Design and Analysis of Communication Systems
[2] DE = Design Engineering

for an exam. Every response to such a stimulus is called a task. An example of a task could be: to prepare all the subsystems for an examination. The time between sending the stimulus and the system having done all the responses to this stimulus, is called the response time and is the measure of interest in this thesis. The response time is important because long response times result in a system that is not user-friendly.

In practice a number of tasks are present in the system at the same time. Hence the handling and blocking of one task influences the response time of other tasks. For Philips Healthcare the following measures are interesting on the ASD architecture:

- The mean response times of tasks
- The Worst case response times of tasks

With an accurate model and efficient analysis method the performance of tasks can be evaluated on different software architectures, without completely implementing them. Using the results from the analysis, performance issues can be located in an early stage of the development. This makes it possible to incorporate performance in the design decisions of a project.

1.3 Contribution

Within this master project I would like to model the performance of the ASD generated software using an analytical approach. Within this model and analysis method a strong hierarchy is desired to keep the modelling and analysis possible for larger systems. The modelling and analysis will be build up step by step, to keep the problem manageable. When necessary assumptions or approximations will be to make the modelling and analysis possible. A simulator (which simulates the working of the ASD generated software) will be build to validate the analysis results. Using this simulator the analytical results will be validated. Based on the validation results, extensions will be made to the analysis method.

1.4 Outlook

This thesis is further organized as follows: In Chapter 2 the simplified version of the ASD structure used for this study is explained. Chapter 3 lists the requirements for the modelling and analysis method, discusses several modelling and analysis methods for this problem and lists the required background information for this thesis on queueing theory. In Chapter 4 the modelling of a single ASD block is explained, that is developed in this thesis. The simulator build in this project to validate the analytic models is presented in Chapter 5. In Chapter 6 the single block analysis is validated and extended to multiple blocks. Using the results from the validation, Chapter 7 presents a general algorithm to model ASD structures with multiple blocks. Finally in Chapter 8 the conclusion and recommendations are given.

2 Case study

In this chapter the simplified ASD structure used in this thesis is discussed. Firstly it is explained how the ASD method is used to make software (Section 2.1). Secondly, the modelling structure is detailed out in Section 2.2. In the rest of the chapter the elements of the ASD generated software (Section 2.3) and their behaviour are denoted in Section 2.4.

2.1 Software design using ASD

PHC uses the design method ASD by Verum[16], which is a software development package with integrated automatic formal verification possibilities. The software is build using the same tool that does the formal verification. The ASD suite can check for deadlocks (reaching a state from which it cannot exit) and life-locks (keeps running in a single loop, without doing normal operation) in the code.

Developing software using the ASD suite is not just writing lines of code, but is largely based on state diagrams. The generated software is composed of a number of communicating state diagrams. These diagrams consist of states and transitions, to which functions can be attached. The diagrams together with the functions are transformed into a working program by the ASD suite. The basic idea is displayed in Figure 2.1. The states (the ovals) the program can be in are displayed with their possible transitions (the arrows). These transitions can be triggered by calls from other blocks or by the functions themselves. The functions connected to these transitions are executed when a transition is triggered.

In the example program of Figure 2.1 there are three states: Running, Receiving and Waiting. When the program is in the Running state, the program can be switched to the Receiving state by an external call. In the Receiving state, a call can either move the program to the Running- or to the Waiting state. In the Waiting state an incoming call either makes the program stay in the Waiting state or move it to the Running state. When the transition from the Waiting state to the Running state is made the code connected to this transition is executed.

The state diagram structure allows for formal verification, but also provides structure and overview of the software to the designer. The ASD suite can automatically generate an executable from these state diagrams and the corresponding functions. This means the diagrams are part of the code, so they are always up to date. In other design philosophies, where diagrams are only used for documentation, it often happens that, code is changed, however, updating the corresponding diagrams is forgotten.

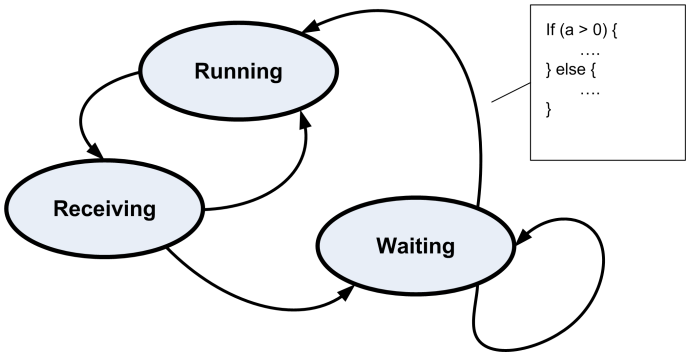


Figure 2.1: A state diagram with code connected to it

In this thesis a simplified version of the ASD structure is be used, that will be explained in the next sections. This simplified version has a stricter structure than the real generated software, which makes analysis easier.

2.2 Y-model for ASD

When modelling the performance of the ASD based software, the performance evaluation of ASD based software can be split up into the following parts:

Architecture This part contains the complete ASD architecture and block as well as the relations between all ASD blocks. The model of the architecture should closely reflect the behaviour of the ASD blocks handling calls. The architecture is the basis for the whole model so errors made in this part could be amplified by the other parts of the mode.

Tasks The performance of the complete task is the measure of interest. A task is, for example, the reaction the system has to perform when a button is pushed. Each task follows its own characteristic path through the software architecture. The flow of a task through the architecture determines which parts of the architecture contribute to the response time of this task.

Resource mapping The software runs on a computer with its own specifications. To be able to make an accurate prediction of the performance, the platform on which the software runs should also be taken into account. The most important influence is the fact that multiple parts in the architecture share the same processor. Possibly, also the scheduling mechanism of the operating system has to be incorporated in the model, because of the performance loss due to switch over times (time to switch between two processes).

Eventually, all these different model parts form the complete model. To avoid enormous calculation times, a smart composition of the parts has to be found. The different modelling aspects are modelled as independently as possible and only at the end they are combined to a single model including all aspects. This, so called Y-model philosophy is displayed in Figure 2.2. In the Y-model philosophy[6][7] all aspects of a system are handled separately and combined at the end to model the complete behaviour.

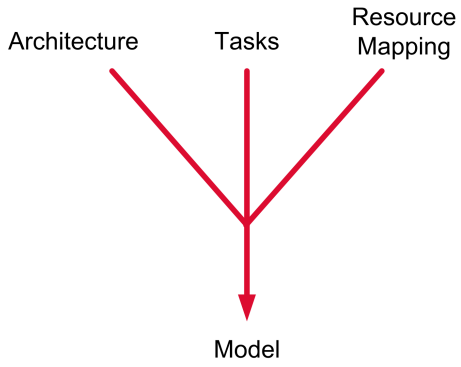


Figure 2.2: Y-model

2.3 ASD block

The state diagram of Figure 2.1 is rather simple, whereas the state diagram of a complete software system is enormous. To perform formal verification, the state diagram has to be kept small enough, otherwise model-checking will take too much time due to the growing number of states (state space explosion).

To keep the model manageable, the developer splits the state diagram of the complete software system into smaller parts. Each part is implemented as a single ASD block. These blocks have a clearly defined interface. Other blocks only see the interface and consider the block itself as a black box. The state space of a single block together with the interfaces of the other blocks is much smaller than the state diagram of the complete software system, making formal verification possible. Blocks communicate between each other using calls.

Transitions in the state diagram of Figure 2.1 can be triggered by calls issued by other blocks. Two types of calls exist: synchronous calls (S) and asynchronous calls (AS). Synchronous calls return to their caller when they have been finished, asynchronous calls do not send returns. The asynchronous calls which an ASD block receives are queued upon arrival. Only one synchronous call can be present in a block at the same time. Asynchronous calls are handled in a First In First Out (FIFO) order. They have priority over synchronous calls. Only one call can be handled at a time and calls in progress are not interrupted (pre-empted). Asynchronous calls have priority over synchronous calls, hence a synchronous call can only be handled when the asynchronous queue is empty. When a synchronous call is in service, upon the arrival of an asynchronous call, the asynchronous call has to wait until the processing is done. A schematic view of a single ASD block is given in Figure 2.3.

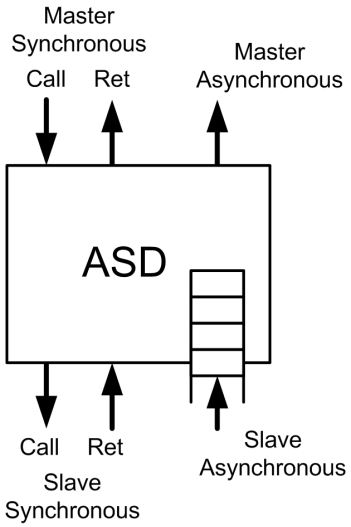


Figure 2.3: Schematic picture of an ASD block

All the discussed effects are displayed in the state diagram in Figure 2.4. In this figure, five different states of an ASD block are shown:

(1) The block is empty: No call in the server and the queues are empty. Both synchronous and asynchronous calls can arrive in this state.

(2) Only AS calls present: An asynchronous call is in the server and possibly a number of asynchronous calls are in the queue. Both synchronous and asynchronous calls can arrive in this state.

(3) Only a S call in server, no AS queue: Only a synchronous call is in the server, no asynchronous calls are present in the block. Because there is already a synchronous call present in he block in this state, only asynchronous calls can arrive in this state.

(4) An S call in server + AS queue: A synchronous call is in the server, but during serving the synchronous call, one or more asynchronous calls have arrived. Because there is already a synchronous call present in he block in this state, only asynchronous calls can arrive in this state.

(5) AS in server, S call waiting + optional AS queue: An asynchronous call is in the server and optional asynchronous calls are in the queue. A synchronous call is also waiting for service. In this situation all the asynchronous calls have to be handled before the synchronous call is served. Because there is already a synchronous call present in he block in this state, only asynchronous calls can arrive in this state.

The non-pre-emptive behaviour of the block allows for a synchronous call to be in service, while one or more asynchronous calls are waiting in the queue.

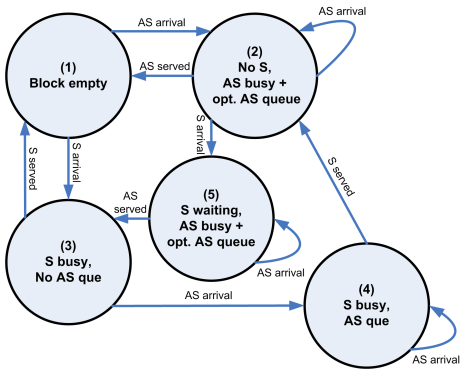


Figure 2.4: Overall state diagram ASD block

2.4 ASD architecture

One block cannot be large enough to describe the whole program, so programs have to consist of multiple blocks. But still the formal verification has to hold for the whole program. The construction of the ASD architecture guarantees that the complete program will be deadlock free, when all blocks of the program are deadlock free itself.

The generated software will have a certain structure, referred to here as the ASD architecture. The ASD architecture is a tree structure composed of a number of communicating ASD blocks as displayed in Figure 2.5. The tree structure determines that every master (parent) can have multiple slaves (children), but every slave has only one master.

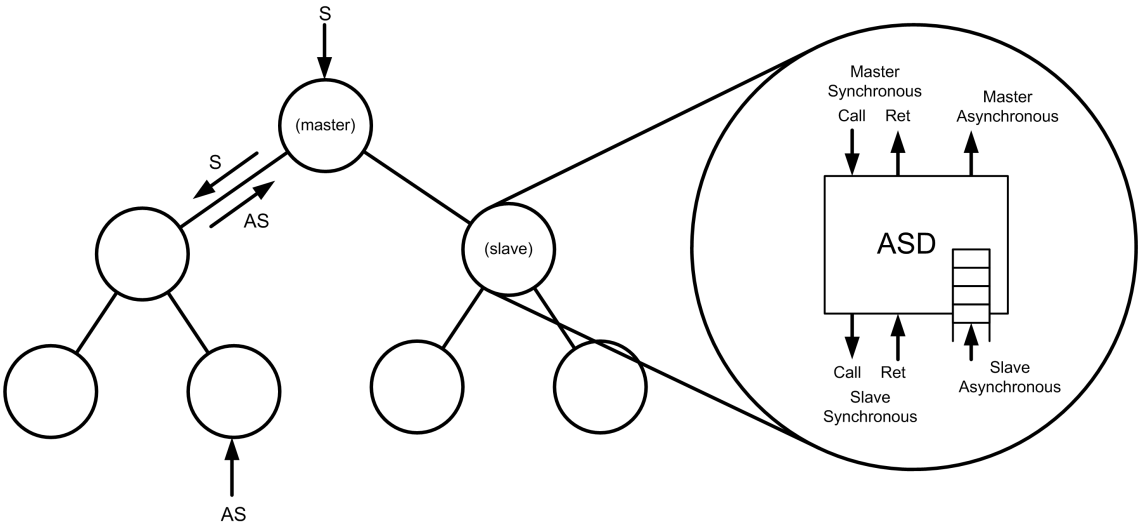


Figure 2.5: ASD tree structure

The ASD blocks can communicate with other blocks using synchronous (S) and asynchronous calls (AS). The synchronous calls can only go top down in the structure, while the asynchronous calls only go bottom up. Hence, when a block wants to send a call to its parent this is done via an AS call, but when it addresses a child, this is done via a S call. This rigid structure is necessary for formal verification by Verum's ASD suite.

A new call can be send to one of its children or to its parent, as part of the response. When a synchronous call is send to another block, the sender remains "blocked" until it has received a return on the call, just like a function call in a program. This effect is illustrated in Figure 2.6.

In this example a master receives a synchronous call, does some processing (P) and then issues a synchronous call to one of his slaves, which does the same. While a slave is processing, the master stays blocked (B). The blocking is removed by the synchronous return. Issuing an asynchronous call however, is non-blocking. The caller does not have to wait until the call has finished, but just continuous with other work.

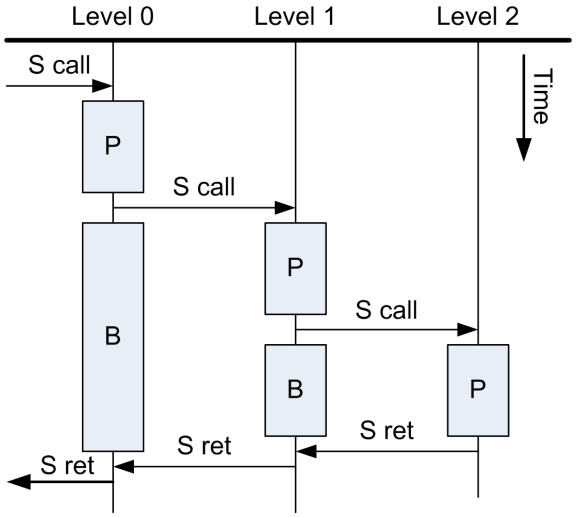


Figure 2.6: Timeline of nested synchronous calls

The tree structure together with the blocking ensures that there can only be one synchronous call at one block, because there is only one master node that can issue a synchronous call to its slave. As soon as a synchronous call is issued at one of the slaves, the master is blocked, so it cannot issue any other synchronous calls.

With the asynchronous calls however, the block continuous serving the next call directly after the previous asynchronous call is served. After finishing service of an asynchronous call, the call directly goes the next block. Also the input of asynchronous calls does not wait for completion of the service. So there can be multiple calls present at each block.

2.4.1 Tasks

Eventually, the whole ASD structure forms a complete program, which has to respond to a number of tasks. Such a task could be, e.g., to configure the X-Ray tube. A task consists of one or more calls and enters the architecture as synchronous or asynchronous call and then flows through the architecture according to a predefined path.

The path depends on the type of task, but is always the same for a certain type of task. Tasks, starting with a synchronous call, can only enter at the top of the tree structure. Tasks, starting with an asynchronous call can only enter at the bottom of the architecture.

Two examples are given in Figure 2.7; In the left example, a task enters as synchronous call at the top and flows though the architecture. The order of the issued calls is addressed by the numbers next to the arrows. From start to end, the top block remains occupied, because it is either busy or has outstanding synchronous calls. In the right example, a task enters at the bottom as asynchronous call. In this example no blocking occurs, because only asynchronous calls take place.

In the previous examples, the tasks consisted of either only synchronous or either only asynchronous calls, but tasks can also consist of both types of calls, as shown in Figure 2.8. The left example shows how synchronous calls trigger asynchronous calls. The order of the synchronous calls and their returns is given by the numbers next to the arrows. Some of the synchronous calls generate asynchronous calls. Synchronous call (3) at block D generates an

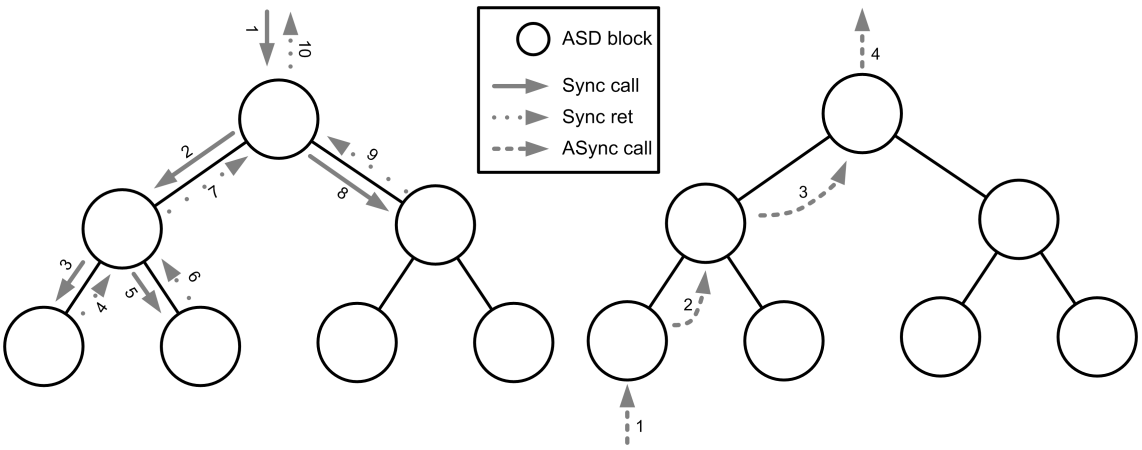


Figure 2.7: Two task examples

asynchronous call (4A). This call is put into the queue of block D. The block continues execution with call 3 and the synchronous return is generated before the asynchronous call 4A is started. The same happens with call 8A as reaction on call 8 at block C. The synchronous calls do not wait until the asynchronous calls are started. The asynchronous calls are indicated by letters, because the exact execution order between the 4X and 8X call series is not defined by the architecture, but depends on the usage of the blocks by other tasks and their service time. When the synchronous part is done, the synchronous return (10) is sent. After call 10 has taken place the block A is available for service again, so call 4D and 8C will arrive also, but due to the asynchronous handling it is not a priori clear which of them arrives first. In this situation, the response time of this task is the time it takes between sending call 1 and finishing the last asynchronous call (either 4D or 8C).

In the right example, asynchronous calls generate synchronous calls. In this case, asynchronous call number 2 generates a synchronous call at block E (call number 3). Hence, the master (block B) is blocked until the slave (block E) is finished, although the master (block B) is handling an asynchronous call. After the return call (number 4), a new asynchronous call is issued at block A. Finally call 6 finishes the task. So in this example,the order of the calls is predefined, despite the fact asynchronous calls are involved.

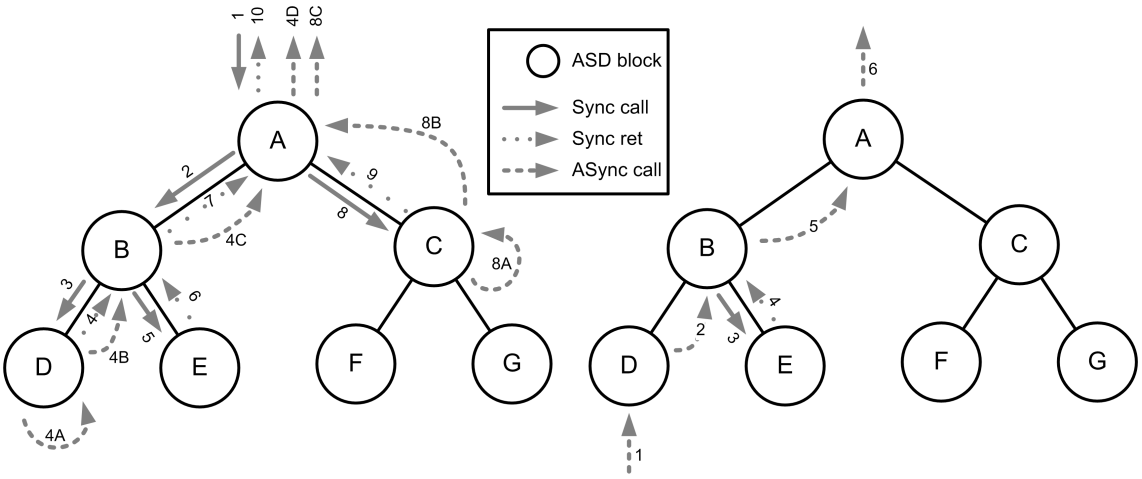


Figure 2.8: Two more complex task examples

3 Performance analysis and queueing models

The purpose of this thesis is to find a suitable modelling formalism together with an efficient analysis method to model and analyse the ASD structure. Before going into detail, firstly the requirements for the solution have to be stated in Section 3.1. Secondly, different modelling formalisms together with their analysis techniques are discussed in Section 3.2. Starting in Section 3.3 the required background information about already existing techniques is listed.

3.1 Requirements for performance analysis

An adequate model has to be made together with an usable analysis method for the performance analysis. To be useful in practice a number of requirements have to be met:

Scalability The whole software system will be huge (approximately 17 million lines of code). Based on the earlier systems designed at Philips, it can be assumed that the complete system will consist of up to 1000 blocks. The model and the analysis algorithm have to be able to handle these sizes within a reasonable calculation time. This means, the modelling formalism and the analysis technique have to be scalable.

Accurate The results of the model should reflect the reality close enough to make useful design decisions. The problem is that more accuracy results in longer calculation time, so approximations have to be made. The challenge is to choose the right approximations, so the model is accurate enough to be useful and small enough to obtain results quickly.

The primary measure of interest is the response time. Besides that also waiting time and service time are interesting, because information on waiting time and service time can help to find the bottleneck in the system.

3.2 Related work

When looking at the available modelling and analysis methods, there are a few useful approaches. These methods are listed below with a short description. More detail is given in later chapters.

3.2.1 Waiting time propagation

This method is based on queueing theory. A single ASD block is modelled as a queueing station, but the block cannot be modelled totally independent. Due to the blocking, the response time of the parent depends on the response time of the children. The bottom blocks have no children, so they do not have these dependencies and their response time can be calculated immediately. Their response times then influence the response time of their parents and hence is propagated upwards.

With this method the performance of several blocks sharing one processor cannot easily be analysed, because the load of every block influences the performance of all the others. Then, simple waiting time propagation is not possible anymore. Assigning fixed time shares can be used to overcome this. The propagation method works only with exponentially distributed processing times.

This thesis focusses on the waiting time propagation method.

3.2.2 Discrete event simulation

When a process is event based, it can be simulated efficiently using a discrete event simulator. In contrast to a discrete time simulator used for, e.g. simulating differential equations, this event simulator only processes the events and skips the parts where no events happen.

In the processes of this thesis the inter-event times are randomly distributed. However, the distribution of these variables is defined. Due to this defined distribution, several measures (like queue length) have expected values.

The simulator is used to derive these expected values. In order to do so, for each randomly distributed variable, it picks random numbers according to the specified distribution using a random number generator. When a lot of events have been simulated, the estimated average values approach the expected values. The simulator is used to validate the expected values derived from the analysis. In order to obtain accurate estimations, long simulations are required, making discrete event simulation only useful to analyse models of small systems. Typical values to validate are response time of calls.

The discrete event simulation will be used to validate the waiting time propagation method.

3.2.3 Modular performance analysis

The basic idea of modular performance is that different parts in a system have their own characteristic performance curves [2][17]. These curves are typically described as mathematical functions. Each element in the chain modifies the input curve with its own curve and sends the resulting output curve to the next component. With only a forward path, this technique can be used nicely to estimate best and worst case latencies of the system under investigation. This technique is often applied to perform jitter analysis on real-time systems. Because all the system properties are described as curves, any distribution can be evaluated. When circular dependencies occur in the model, as with processor sharing, this analysis technique has difficulties, because the calculus behind the propagation of the performance curves cannot handle cyclic dependencies. A way to solve this is to assign fixed time shares to each block on front.

In this thesis modular performance analysis will not be used, however, it could be evaluated in a future project.

3.3 Background

The concept of waiting time propagation is based on queueing theory. The following sections provide the necessary background information on existing queueing theory and is based on [9]. First the queues are introduced together with some important properties in Sections 3.4 upto 3.9. Next several analysis techniques are discussed for the presented queues (Sections 3.10 upto 3.13).

3.4 Queueing station

A queueing station consists of a queue and a server and is used by customers. A graphical representation of a queueing station is given in Figure 3.1. Customers arrive to the queue and wait there before they are put into service. They are handled, e.g., in a *first come first served order* and leave the server when their service is completed. When a customer arrives, it is either handled directly in case the server is free or it is put in the queue if the server is occupied.

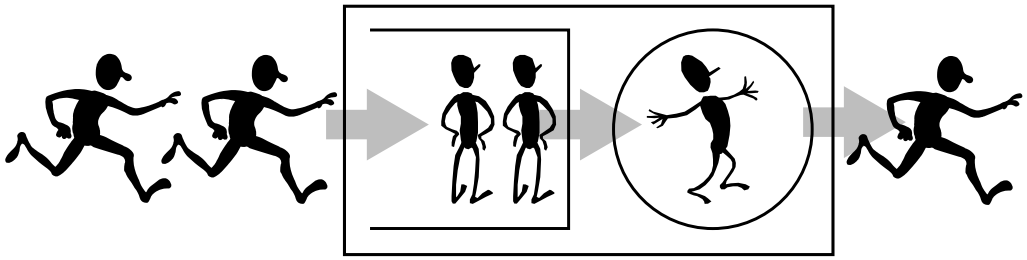


Figure 3.1: Graphic representation of a queueing system

The time the customer spends in the server is denoted as service time. The inter-arrival time is the time between the arrival of two succeeding customers. A graphical representation of the

inter-arrival time is given in Figure 3.2. In this example, the inter-arrival times (IA1 and IA2) are not identical, which is the case when there is variance in the inter-arrival times. Queueing occurs because inter-arrival times are shorter than the service times. When the system is long term stable (average inter-arrival times are longer than the average service times), variance in the service and inter-arrival times can make queueing occur. In general it holds that the larger the variance in the inter-arrival and service time, the more queueing occurs.

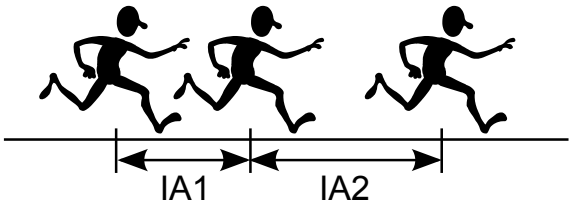


Figure 3.2: Visualisation of inter-arrival times

The time between entering the queue and leaving the server is called the response time. This response time (R) is composed of the waiting time (W), i.e. the time the customer spends in the queue, and the service time (S), i.e. the time the customer spends in the server. Hence, for their expected values ($E[\ldots]$) Equation 3.1 holds:

$$E[R] = E[W] + E[S]. \tag{3.1}$$

3.5 Kendall notation

To compactly describe a queueing station, the *Kendall Notation* is used. This notation consists of the following 6 identifiers separated by vertical bars,

Arrivals | Service | Servers | Buffersize | Population | Scheduling.

"Arrivals" describes the customer arrival process. "Service" indicates the service process. The "Servers" tells how many servers there are and "Population" denotes the size of the customer population. "Scheduling" describes the scheduling scheme that is applied.

Often, buffer size, population and scheduling method are not listed in the description of a queueing station, as with an M|M|1 queue. In that case the buffer size and population are assumed to be infinite. When the scheduling method is not listed, the scheduling scheme is *first come first serve* (FCFS).

The "Arrivals" and "Service" may have different values. The most common value is M (Memoryless or Markovian): the inter-arrival or service times are random according to the negative exponentially distribution. Also the G (General) is used when the times may have an arbitrary distribution.

In this chapter the focus is on the M|M|1 queue, meaning:
- Negative exponentially distributed inter-arrival times
- Negative exponentially distributed service times
- One server
- Infinite buffer size
- Infinite population
- FCFS scheduling scheme

3.6 Little's law and utilisation

Assuming that on average λ customers arrive per time-unit at a queueing station and that a customer spends on average $E[R]$ time in the systems, during this period on average $\lambda \times E[R]$

customers arrive. According to Little's law [11], the average number of customers in the system is $E[N] = \lambda E[R]$.

Variants to this law hold for the average number of customers in the queue $E[N_q] = \lambda E[W]$ and for the average number of customers in the server $E[N_s] = \lambda E[S]$.

Assuming that there is only one server, $E[N_s]$ lies on the interval $[0,1]$ (between empty and busy). $E[N_s]$ also indicates the probability that the server is busy. Therefore $E[N_s]$ is often called the *utilisation* and is denoted as $\rho = \lambda E[S]$.

There are some important remarks to make for Little's Law:
- This law only expresses the relation between the average values for the arrival rate, residence time and number of customers in the system.
- It holds for arbitrary distribution of the arrivals or service time.
- This law holds for single queueing stations but also for networks of queueing stations.

For finite queues a different version of Little's law exists, but this is not discussed here.

3.7 The Markov property

Little's law is based on average values, but more advanced analysis methods need to take the distributions into account.

A queueing station can also be seen as a continuous-time and discrete-state stochastic process. A stochastic process is a collection of random variables $\{X(t) \mid t \in \mathscr{R}^+\}$, where $X(t)$ can take any value of the so-called state space $\mathscr{I}$. In case of a countable infinite population and buffer, the state space equals $\mathscr{I} = \{0,1,2,\ldots\}$.

In case of the M|M|1 queueing station the state space is discrete, because only whole customers are allowed. The state space is also infinite, because the queue has an infinite size. The time $t \in \mathscr{R}^+$ is continuous, because customers can arrive and be served at any instant in time.

When both the inter-arrival and service times are negative exponentially distributed, a special property holds called the *Markov property*. The Markov property states that the next state of the stochastic process only depends on the current state and is independent of the past. In case of an M|M|1 queue this means, that the time until the next customer arrives does not depend on the time that has elapsed since the last customer has arrived. The only continuous distribution that satisfies this *memoryless* property is the negative exponential distribution.

Because both, service times and inter-arrival times, are negative exponentially distributed in the M|M|1 queue, the time it takes to have n customers in the queue only depends on the current number of customers in the queueing station.

The Markov property in continuous time states that for non-negative times $t_0 < t_1 < \ldots < t_{n+1} \in \mathscr{R}^+$ and the corresponding states $x_0, x_1, \ldots, x_{n+1} \in \mathscr{I}$ Equation 3.2 holds:

$$\Pr\{X(t_{n+1}) = x_{n+1} \mid X(t_0) = x_0, \ldots, X(t_n) = x_n\} = \Pr\{X(t_{n+1}) = x_{n+1} \mid X(t_n) = x_n\}. \tag{3.2}$$

This equation states that the probability to be in state x_{n+1} (at time t_{n+1}), only depends on the current state of the queueing station x_n at time t_n and not on what happened before time t_n.

3.8 The PASTA property

For queueing stations with negative exponentially distributed inter-arrival times, the number of arrivals in a given interval is Poison distributed. For such an arrival process the so called PASTA property holds. PASTA abbreviates *Poison Arrivals See Time Averages* and states that when a new customer arrives, it sees the queue as in equilibrium. Hence, if a customer arrives according to a Poison arrival process, he experiences the average queue length.

3.9 CTMC

A continuous time and discrete state space stochastic process, for which the Markov property holds, is called a *Continuous Time Markov Chain* (CTMC). The CTMC can be graphically displayed as state transition diagram. In Figure 3.3, the state transition diagram of the CTMC underlying an M|M|1 queueing station is displayed. In this diagram, the states are displayed as vertices and the edges show the transitions from state i to j. The edges indicated with λ, model the arrival of customers, while serving of customers is modelled by edges equipped with μ. The rates of the edges (λ and μ) are parameters of the negative exponential distribution. In this CTMC state 0 stands for an empty system and state i means there are i customers in the system, i.e. there are $i-1$ customers in the queue.

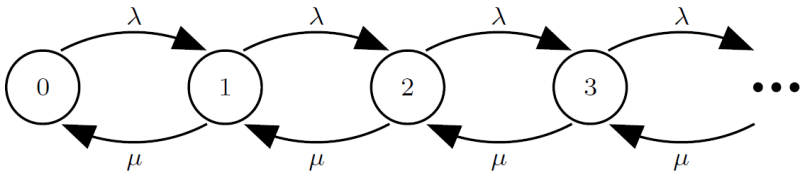


Figure 3.3: The CTMC of an M|M|1 queue

Besides the graphical representation of Figure 3.3, the CTMC of the M|M|1 system can also be described by its generator matrix **Q**. This matrix contains the same information as the figure, where in general row i indicates the ingoing rates into the other states contributed by state i. Element $q_{i,j}$ indicates the rate for a transition from i to j. The sum of each row has to be zero.

The first row in the **Q** matrix (Equation 3.3) represents the outgoing transitions from state 0. Element $[0,1]$ in the matrix is λ because there is a transition from state 0 to state 1 with rate λ. The element $[0,0]$ is $-\lambda$ because total sum of outgoing rates from state 0 is λ, so as incoming rate this becomes $-\lambda$. The second row lists the outgoing transitions from state 1. Element $[1,0]$ is μ due to the transition of rate μ from state 1 to state 0. The transition with rate λ from state 1 to 2 causes element $[1,2]$ to be λ. The sum of outgoing rates is $\lambda+\mu$, so element $[1,1]$ is $-(\lambda+\mu)$.

$$\mathbf{Q} = \begin{bmatrix} -\lambda & \lambda & 0 & 0 & 0 & \cdots & 0 \\ \mu & -(\lambda+\mu) & \lambda & 0 & 0 & \cdots & 0 \\ 0 & \mu & -(\lambda+\mu) & \lambda & 0 & \cdots & 0 \\ 0 & 0 & \mu & -(\lambda+\mu) & \lambda & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \end{bmatrix} \tag{3.3}$$

The state of a system is a random variable, hence, the system can be evaluated using probabilities. A vector $\vec{p(t)}$ can be defined, which indicates for every state i as $p_i(t)$ the probability of being in that state at time t. The complete vector is shown in Equation 3.4.

When the system starts empty $p_0(0) = 1$ and the probabilities of being in other states are zero, . While time evolves, the change of the state is defined by differential Equation 3.5. The changing behaviour of the state probability vector $\vec{p}$ over time is called transient behaviour. When the system is stable, eventually, it will reach a steady-state. In steady-state, the state probability vector does not change any more, resulting in: $\frac{\partial \vec{p(t)}}{\partial t} = 0$.

$$\vec{p(t)} = \begin{bmatrix} p_0(t) & p_1(t) & p_2(t) & \ldots & p_\infty(t) \end{bmatrix} \tag{3.4}$$

$$\frac{\partial \vec{p(t)}}{\partial t} = \vec{p(t)}\mathbf{Q} \tag{3.5}$$

The steady-state probabilities can be used to calculate the expected waiting times $E[W]$ and

all kind of related measures. The steady state probabilities can be calculated by solving Equation 3.6 under the condition of Equation 3.7 (the sum over all probabilities equals 1).

$$\vec{p(t)}\mathbf{Q} = \vec{0} \tag{3.6}$$

$$\sum_{i=0}^{\infty} p_i = 1 \tag{3.7}$$

3.10 Non-preemptive priority scheduling

The queues discussed so far cannot handle classes of customers differently. In practice however different priorities are often applied to different classes. With non-preemptive priority scheduling customers with higher priority are always handled before customers of lower priority levels, but the work on the customer in service is never interrupted (= non pre-emptive). A graphical representation of the non-pre-emptive priority scheduling is given in Figure 3.4. In the queueing systems each priority level has his own queue, corresponding arrival rate and service rate, but they share a single server.

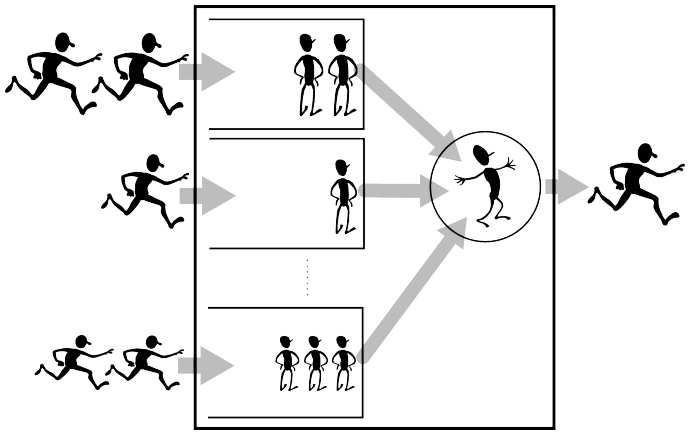


Figure 3.4: Different priority levels using a single server

For non-pre-emptive priority scheduling systems, where customers (at each priority level) have negative exponential distributed inter-arrival times and an arbitrary distributed service time (M|G|1) an analysis method for the waiting times of each priority is available. The waiting time for a customer, arriving to priority level r is composed of the following parts:

- The work that has to be done on the customer that is currently in service.
- The time needed to serve the customers already in the queues with higher or equal priorities $k = 1, \ldots, r$ when the customer arrives.
- The time needed to serve the customers with higher priorities $k = 1, \ldots, r-1$ which arrive during the waiting period of this customer.

This can be written into the following equation:

$$W_r = T_P + \sum_{k=1}^{r} T'_k + \sum_{k=1}^{r-1} T''_k. \tag{3.8}$$

In this equation W_r is the waiting time for a customer of priority level r. T_P is the remaining service time of the customer in service when this customer arrives. T'_k is the time needed to serve the customers of priority k that are already in the queue when this customer arrives. T''_k is the time it takes to serve the customers of priority k that arrive while this customer waits for service. Equation 3.8 can be rewritten into Equation 3.9 to find the expected values,

$$E[W_r] = E[T_P] + \sum_{k=1}^{r} E[T'_k] + \sum_{k=1}^{r-1} E[T''_k]. \tag{3.9}$$

To actually compute the values for $E[W_r]$, the terms of Equation 3.9 have to be filled in. As the inter-arrival times are negative exponentially distributed, the PASTA property holds. This means that each arriving customer finds all queues to be in equilibrium.

$E[T_P]$: The remaining service time of the customer in service depends on the type of customer and the probability of having a customer of this class in the queue, which is ρ_r. The remaining processing time for a class r customer is $E[S_r^2]/2E[S_r]$. By combining these two expressions, the remaining service time of the customer in service is given by Equation 3.10,

$$E[T_P] = \sum_{k=1}^{P} \rho_k \frac{E[S_k^2]}{2E[S_k]} = \frac{1}{2} \sum_{k=1}^{P} \lambda_k E[S_k^2]. \tag{3.10}$$

$E[T'_k]$: Lists the amount of work an arriving customer of class k finds in front of him. Because of the PASTA property, an arriving job sees the queues in equilibrium. Using Little's Law the number of customers in the queue can be calculated as follows: $E[N_{q,k}] = \lambda_k E[W_k]$. These customers require on average $E[S_k] = 1/\mu_k$ service, resulting in:

$$E[T'_k] = \frac{E[N_{q,k}]}{\mu_k} = \frac{\lambda_k E[W_k]}{\mu_k} = \rho_k E[W_k]. \tag{3.11}$$

$E[T''_k]$: Represents the time it takes to process the work that arrives in queue k, while a customer of lower priority waits for service. During the waiting time of a class r customer, on average $\lambda_k E[W_r]$ customers arrive in queue k, where each customer requires on average $1/\mu_k$ service. This results in Equation 3.12:

$$E[T''_k] = \frac{\lambda_k E[W_r]}{\mu_k} = \rho_k E[W_r]. \tag{3.12}$$

By substitution these results into Equation 3.9, Equation 3.13 is obtained.

$$\begin{aligned} E[W_r] &= \sum_{k=1}^{P} \frac{\lambda_k}{2} E[S_k^2] + \sum_{k=1}^{r} \rho_k E[W_k] + \sum_{k=1}^{r-1} \rho_k E[W_r] \\ &= \sum_{k=1}^{P} \frac{\lambda_k}{2} E[S_k^2] + \sum_{k=1}^{r-1} \rho_k E[W_k] + \sum_{k=1}^{r} \rho_k E[W_r] \\ &= \sum_{k=1}^{P} \frac{\lambda_k}{2} E[S_k^2] + \sum_{k=1}^{r-1} \rho_k E[W_k] + E[W_r] \left(\sum_{k=1}^{r} \rho_k \right) \end{aligned} \tag{3.13}$$

By bringing $E[W_r]$ outside the equation, this becomes Equation 3.14. When $\sigma_r = \sum_{k=1}^{r} \rho_k$ is used, Equation 3.15 is acquired.

$$E[W_r] = \frac{\sum_{k=1}^{P} \frac{\lambda_k}{2} E[S_k^2] + \sum_{k=1}^{r-1} \rho_k E[W_k]}{1 - \sum_{k=1}^{r} \rho_k} \tag{3.14}$$

$$= \frac{\sum_{k=1}^{P} \frac{\lambda_k}{2} E[S_k^2] + \sum_{k=1}^{r-1} \rho_k E[W_k]}{1 - \sigma_r} \tag{3.15}$$

For the queue with the highest priority ($r = 1$) this is Equation 3.16, for $r = 2$ (second priority) this results in Equation 3.17. The generalized version is Equation 3.18.

$$E[W_1] = \frac{E[T_P]}{1-\sigma_1} = \frac{E[T_P]}{1-\rho_1} = \frac{\sum_{k=1}^{P} \lambda_k E[S_k^2]}{2(1-\rho_1)} \tag{3.16}$$

$$E[W_2] = \frac{E[T_P] + \rho_1 E[W_1]}{1-\sigma_2} = \ldots = \frac{E[T_P]}{(1-\sigma_2)(1-\sigma_1)} \tag{3.17}$$

$$E[W_r] = \frac{E[T_P]}{(1-\sigma_r)(1-\sigma_{r-1})} \tag{3.18}$$

Equation 3.18 is called Cobham's formula [3], [4]. Using this function for non-pre-emptive priority scheduled M|G|1 queue, the waiting time for each priority class can be derived, based on the first and second moment of the service times and the arrival rates.

3.11 Quasi birth-death processes

Recall the CTMC model of an M|M|1 queue as displayed in Figure 3.3. In an M|M|1 queue customers enter the system with rate λ and leave the system at rate μ. This process is also denoted birth-death process. The state-space of the M|M|1 queue is infinite, but it is very regular, as each state only has transitions to its direct neighbours. As shown in the generator matrix $\mathbf{Q}$ of the M|M|1 queue in Equation 3.19, except for the first column, the next row is just a shifted version of the previous one. This regular structure makes it possible to derive the steady state probabilities, although the system has an infinite state space.

$$\mathbf{Q} = \left[\begin{array}{c|cccccc} -\lambda & \lambda & 0 & 0 & 0 & \cdots & 0 \\ \mu & -(\lambda+\mu) & \lambda & 0 & 0 & \cdots & 0 \\ 0 & \mu & -(\lambda+\mu) & \lambda & 0 & \cdots & 0 \\ 0 & 0 & \mu & -(\lambda+\mu) & \lambda & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \end{array}\right] \tag{3.19}$$

This regularity can be generalized and extended to more complex CTMCs. The states of an M|M|1 queue can be extended to levels, each level consisting of more then one state. Transitions are still only allowed to neighbouring levels and to states within the same level, this process is called a quasi birth death (QBD) process. Quasi birth death (QBD) processes have a starting level 0 and repeating levels $i \in \mathcal{N}$. QBD processes are infinite in one dimension and have a finite size in the other dimension.

Because each level consists of multiple states, scalar values for the transition rates between levels do not work, instead matrices have to be used and also transitions within a level have to be defined. A block schematic view of the levels and corresponding transitions is displayed in Figure 3.5. The levels are displayed as blocks, the text with the arrow states the name of the corresponding transition matrix. The leftmost level has a different structure and is called the boundary level. All the other levels have the same structure and continue to infinity and are therefore called repeating levels.

Figure 3.6 shows the possible states and transitions within a level. In the level the states of the CTMC (circles) with the transitions (arrows) are shown. The ovals behind the transitions indicate in which matrix these transitions are described. To keep the image clear only the transitions on the right are marked, but on the left side this marking holds as well. So the transitions (arrows in the figure) indicated by an oval with dotted lines are described by matrix $\mathbf{A_0}$, transitions indicated by with ovals with solid lines are described by matrix $\mathbf{A_1}$ and transitions indicated by ovals with dashed lines are described by matrix $\mathbf{A_2}$.

For a QBD process a generator matrix $\mathbf{Q}$ and a state probability vector $\vec{p}$ can be defined. Each level has its own state probability sub-vector $\vec{z}_i$. The state vector of the complete CTMC is composed of all these state vectors, as displayed in Equation 3.20. The generator matrix $\mathbf{Q}$ of

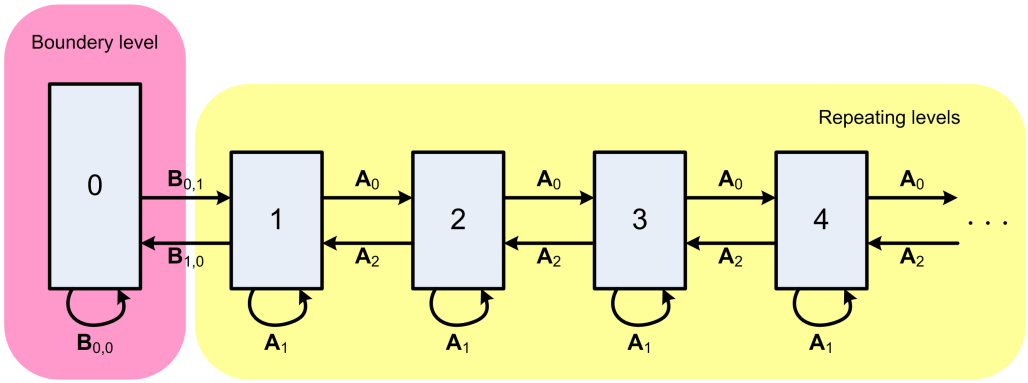


Figure 3.5: QBD level transitions

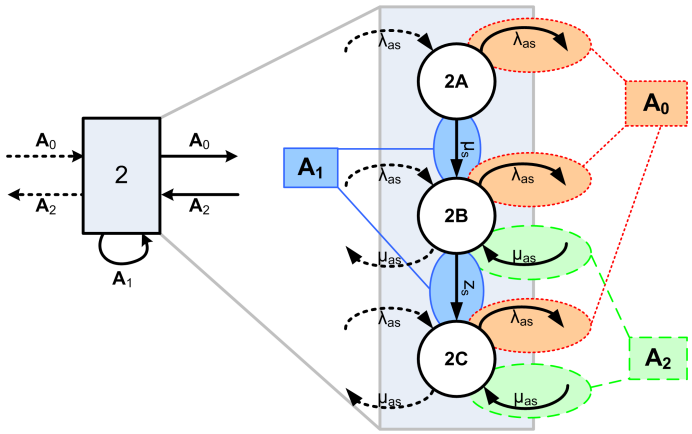


Figure 3.6: QBD level transitions

the QBD process consists of the matrices that describe the transitions within the levels and between the levels. The composition of the generator matrix is displayed in Equation 3.21.

$$\vec{p} = \begin{bmatrix} \vec{z_0} & \vec{z_1} & \vec{z_2} & \dots & \vec{z_i} & \vec{z_{i+1}} & \dots & \vec{z_\infty} \end{bmatrix} \tag{3.20}$$

$$\mathbf{Q} = \begin{bmatrix} \mathbf{B_{00}} & \mathbf{B_{01}} & 0 & 0 & 0 & \cdots & 0 \\ \mathbf{B_{10}} & \mathbf{A_1} & \mathbf{A_0} & 0 & 0 & \cdots & 0 \\ 0 & \mathbf{A_2} & \mathbf{A_1} & \mathbf{A_0} & 0 & \cdots & 0 \\ 0 & 0 & \mathbf{A_2} & \mathbf{A_1} & \mathbf{A_0} & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \end{bmatrix} \tag{3.21}$$

For the levels $i \geq 1$, the size of the state vector is the same. The size of the state vector for level 0 could deviate from that. Hence, matrices $\mathbf{B_{01}}$ and $\mathbf{B_{10}}$ may be rectangular instead of square and $\mathbf{B_{00}}$ may be of different size then $\mathbf{A_1}$. Matrices $\mathbf{A_0}$, $\mathbf{A_1}$ and $\mathbf{A_2}$ always have to be square and of the same size, because the state vectors they operate on are all of the same size.

When the M|M|1 is interpreted as QBD and the state sub-vectors are $\vec{z}_i = p_i$, the matrices take the following form:

$$\begin{aligned}
\mathbf{B_{00}} &= [-\lambda], \\
\mathbf{B_{01}} &= [\lambda], \\
\mathbf{B_{10}} &= [\mu], \\
\mathbf{A_0} &= [\lambda], \\
\mathbf{A_1} &= [-(\lambda+\mu)], \\
\mathbf{A_2} &= [\mu].
\end{aligned}$$

3.11.1 Matrix geometric solution

In this thesis the steady-state probabilities of a QBD process are required. For finite state-space CTMC the steady-state probabilities can be derived by solving equations 3.6 and 3.7. For QBD processes, Equation 3.6 cannot be solved straigth forward, because both $\vec{p}$ and $\mathbf{Q}$ are infinite in size.

To derive the steady-state probabilities for a QBD process, the matrix geometric method [14] exists. This method is the matrix version of the scalar one, used to calculate the infinite sum of a geometric series.

For the matrix geometric solution a matrix $\mathbf{R}$ is defined as in Equation 3.22. The matrix $\mathbf{R}$ relates the steady-state probabilities of two neighbouring levels. This equation holds for all levels $i \geq 2$. The boundary matrices $\mathbf{B_{00}}$, $\mathbf{B_{01}}$ and $\mathbf{B_{10}}$ influence level 0 and 1 and therefore sub-vectors $\vec{z}_0$ and $\vec{z}_1$ have to be treated differently.

$$\vec{z}_i = \vec{z_{i-1}}\mathbf{R} \tag{3.22}$$

By using the definition of the $\mathbf{Q}$ matrix in Equation 3.21, Equation 3.23 can be derived for all levels $i \geq 2$. Together with Equation 3.22 this can be rewritten into Equation 3.24 and further into Equation 3.25. When $\mathbf{R}$ is brought to the other side of the equal sign, this becomes Equation 3.26. As the resulting equation is not linear, solving Equation 3.26 is not straight forward. The easiest way is to use successive substitution [10], which solves it by iterating Equation 3.27. As intial value $\mathbf{R}(0) = \mathbf{A_0}\,\mathbf{A_1}^{-1}$ can be taken.

$$\vec{z}_i\mathbf{A_0} + \vec{z_{i+1}}\mathbf{A_1} + \vec{z_{i+2}}\mathbf{A_2} = \vec{0} \tag{3.23}$$

$$\vec{z}_i\mathbf{A_0} + \vec{z}_i\mathbf{R}\mathbf{A_1} + \vec{z}_i\mathbf{R}^2\mathbf{A_2} = \vec{0} \tag{3.24}$$

$$\vec{z}_i(\mathbf{A_0} + \mathbf{R}\mathbf{A_1} + \mathbf{R}^2\mathbf{A_2}) = \vec{0} \tag{3.25}$$

$$\mathbf{R} = (\mathbf{A_0} + \mathbf{R}^2\mathbf{A_2})\mathbf{A_1}^{-1} \tag{3.26}$$

$$\mathbf{R}(k+1) = (\mathbf{A_0} + \mathbf{R}^2(k)\mathbf{A_2})\mathbf{A_1}^{-1} \tag{3.27}$$

Next the steady-state probabilities have to be derived. When the steady state sub-vectors $\vec{z}_0$ and $\vec{z}_1$ are known, the other probabilities can easily be calculated using equation 3.22. From Equations 3.21, Equations 3.28 and 3.29 can be derived.

$$\begin{bmatrix} \vec{z}_0 & \vec{z}_1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{B_{00}} \\ \mathbf{B_{10}} \end{bmatrix} = \vec{0} \tag{3.28}$$

$$\begin{bmatrix} \vec{z}_0 & \vec{z}_1 & \vec{z}_2 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{B_{01}} \\ \mathbf{A_1} \\ \mathbf{A_2} \end{bmatrix} = \vec{0} \tag{3.29}$$

Equation 3.29 can be rewritten into Equation 3.30, because $\vec{z}_2\mathbf{A_2} = (\vec{z}_1\mathbf{R})\mathbf{A_2}$.

$$\begin{bmatrix} \vec{z}_0 & \vec{z}_1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{B_{01}} \\ \mathbf{A_1 + RA_2} \end{bmatrix} = \vec{0} \tag{3.30}$$

Combining equations 3.28 and 3.30, Equation 3.31 can be constructed.

$$\begin{bmatrix} \vec{z}_0 & \vec{z}_1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{B_{00}} & \mathbf{B_{01}} \\ \mathbf{B_{10}} & \mathbf{A_1 + RA_2} \end{bmatrix} = \begin{bmatrix} \vec{0} & \vec{0} \end{bmatrix} \tag{3.31}$$

To derive $\vec{z}_0$ and $\vec{z}_1$ Equation 3.31 has to be solved. Because this equation originates from the **Q** matrix, it does not have a unique solution. Therefore, an additional equation is needed. So the equation $\sum_{n=1}^{\infty} p_i = 1$ is added to find a unique solution, which is written as function of **R**, $\vec{z}_0$ and $\vec{z}_1$ in Equation 3.32. The derivation is based on the standard closed-form for the infinite sum of a geometric series.

$$\sum_{i=0}^{\infty} \vec{z}_i \vec{1} = \vec{z}_0 \vec{1} + \sum_{i=1}^{\infty} \vec{z}_i \vec{1} = \vec{z}_0 \vec{1} + \vec{z}_1 \left(\sum_{i=0}^{\infty} \mathbf{R}^i \right) \vec{1} = \vec{z}_0 \vec{1} + \vec{z}_1 (\mathbf{I} - \mathbf{R})^{-1} \vec{1} = 1 \tag{3.32}$$

Matrix **R** has a rank one fewer than the number of rows. Since matrix **R** is derived using iteration, the linear relations might be distorted, due to small approximation errors, resulting in a rank equal to the number of rows. To reduce the influence for these errors, $\vec{z}_0$ and $\vec{z}_1$ can be calculated in a robust way using Equation 3.33. In this equation $\mathbf{B}^+$ is the pseudoinverse of **B**.

$$\begin{bmatrix} \vec{0} & \vec{0} & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{B_{00}} & \mathbf{B_{01}} & \vec{1} \\ \mathbf{B_{10}} & \mathbf{A_1 + RA_2} & (\mathbf{I} - \mathbf{R})^{-1} \vec{1} \end{bmatrix}^+ = \begin{bmatrix} \vec{z}_0 & \vec{z}_1 \end{bmatrix} \tag{3.33}$$

Since the steady-state probabilities can be derived for $\vec{z}_0$ and $\vec{z}_1$, the steady-state probabilities for $\vec{z}_i$ where $i \geq 2$ can be derived, using Equation 3.34;

$$\vec{z}_i = \vec{z}_1 \mathbf{R}^{(i-1)}. \tag{3.34}$$

Stability

The matrix geometric solution is based on calculating the **R** matrix, which is computed by iteration. However, iteration only works when the matrix converges to a unique solution. This matrix only converges when the CTMC, to which the matrix geometric solution is applied, is stable. An unstable or marginally stable system does not have steady-state probabilities.

A test for the stability is described by Nelson [14]. This test calculates a steady-state vector $\vec{\pi}$ based on the generator matrix $\mathbf{A} = \mathbf{A_0} + \mathbf{A_1} + \mathbf{A_2}$, by solving the equation $\vec{\pi}\mathbf{A} = \vec{0}$ under the condition that $\sum_i \pi_i = 1$. When the system is stable the average flow from level i to $i+1$ should be smaller then the flow from i to $i-1$. Hence, for a stable system $\vec{\pi}\mathbf{A_0}\vec{1} < \vec{\pi}\mathbf{A_2}\vec{1}$ holds.

3.12 Phase-type distributions

The analysis methods presented earlier rely on the Markov property, which holds for the negative exponential distribution. A generalisation of the negative exponential distribution is the phase-type distribution. In Figure 3.7 a CTMC is shown with a single absorbing state (a state with no outgoing transitions). When the initial probability distribution is $p(0) = [1,0]$, the time until the absorbing state 1 is reached is negative exponentially distributed.

When this CTMC is generalized by replacing state 0 by multiple states, the time until the absorbing state is reached is phase-type(PH) distributed. A graphical representation of a general phase-type distribution is shown in Figure 3.8. For this CTMC the state space is defined as $\mathscr{S} = \{1, \ldots, m, m+1\}$, where state $m+1$ is the absorbing state and states 1 to m are transient. For this generalized CTMC a generator matrix **Q** can be defined as in Equation 3.35;

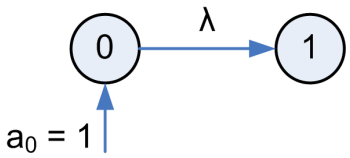


Figure 3.7: The exponential distribution as phase-type

$$\mathbf{Q} = \begin{bmatrix} \mathbf{T} & \vec{T}_0 \\ \vec{0} & 0 \end{bmatrix}. \tag{3.35}$$

Where the transition matrix $\mathbf{T}$ of size $m \times m$ and $\vec{T}_0$ is a column vector with only non-negative elements. To make $\mathbf{Q}$ a proper generator matrix the sum of the rows has to be 1, i.e. $\mathbf{T} \cdot \vec{1} + \vec{T}_0 = \vec{0}$. Hence, $\mathbf{T}$ itself is not a proper generator matrix and vector $\vec{T}_0$ can be derived from matrix $\mathbf{T}$. The initial state vector is given by $(\vec{a}, a_{m+1})$ and has to sum up to 1.

A phase-type distribution is fully described by $\vec{a}$ and $\mathbf{T}$. Both $\vec{T}_0$ and a_{m+1} can be derived from them.

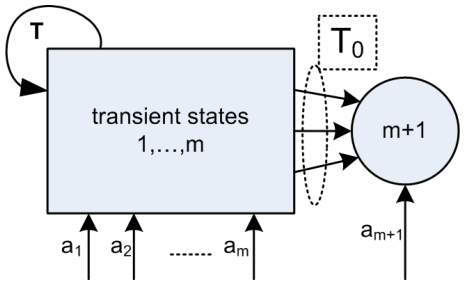


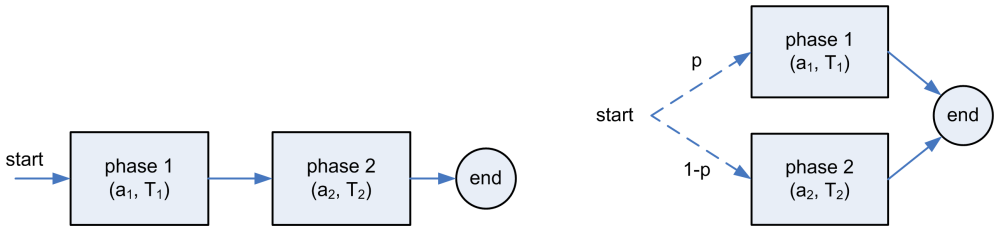
Figure 3.8: Graphical representation of a general phase-type distribution

3.12.1 Manipulation and properties

The i-th moment of a PH-distribution is given by Equation 3.36,

$$E[X^i] = (-1)^i \, i! \, (\vec{a} \, \mathbf{T}^{-i} \, \vec{1}). \tag{3.36}$$

Phase-type distributions can be combined to form a new replacing phase-type distribution. The first basic combination is convolution (two distributions after each other), as schematically shown in Figure 3.9(a). The second denoted is mixture (choosing one of two distributions), as schematically shown in Figure 3.9(b).



(a) Convolution (b) Mixture

Figure 3.9: Combining phase-type distributions

When considering two phase-type distributions $(\vec{a}_1, \mathbf{T_1})$ and $(\vec{a}_2, \mathbf{T_2})$, the convolution of these distributions $(\vec{a}, \mathbf{T})$ can be calculated using Equations 3.37 and 3.38.

$$\vec{a} = \left[\vec{a}_1, (1 - \vec{a}_1\vec{1})\vec{a}_2\right] \tag{3.37}$$

$$\mathbf{T} = \begin{bmatrix} \mathbf{T_1} & -\mathbf{T_1}\vec{1}\,\vec{a}_2 \\ \mathbf{0} & \mathbf{T_2} \end{bmatrix} \tag{3.38}$$

To calculate the mixture, also the probability p of taking phase 1 has to be taken into account. The mixture of both distributions can be calculated using Equations 3.39 and 3.40.

$$\vec{a} = \left[p\vec{a}_1, (1-p)\vec{a}_2\right] \tag{3.39}$$

$$\mathbf{T} = \begin{bmatrix} \mathbf{T_1} & \mathbf{0} \\ \mathbf{0} & \mathbf{T_2} \end{bmatrix} \tag{3.40}$$

3.13 Moment matching

Real systems are often very large and analysis is therefore computationally complex. Simplifying this structure by analysing one part at a time can speed up the analysis. When these systems consists of several sub-systems doing event handling, the response time of the event handling by a sub-system has an own distribution. Due to queueing, this experienced response time can have a complex distribution.

In the systems, investigated in this thesis, all basic random variables are assumed to be negative exponentially distributed. Hence, compositions of these random variables result in phase-type distributions. Although, considering a sub-system as a complex distribution, can reduce the complexity of the total system, analysis with complex distributions remains difficult. To speed up the analysis sacrificing a little accuracy, the distribution can be simplified.

At first the exact distribution has to be specified. The exact distribution is often hard to derive, deriving the moments (first ($E[X]$), second ($E[X^2]$) , third ($E[X^3]$), ...) is often much easier. The simplified phase-type distribution has to match these moments. A number so called *moment matching* algorithms exist for this task. In this thesis the algorithm of Osogami [15] is used.

This algorithm matches a fixed CTMC structure as shown in Figure 3.10, to the provided moments. This CTMC structure is . The $(1-p)$ path goes directly to the absorbing state representing a waiting time of zero. With probability p an Erlang part is started. This Erlang phase is used to make the distribution fit the provided moments when the ratio $\frac{E[X^2]}{E[X]^2}$ is small. The last part is a 2-phase Coxian distribution, which is used to give the distribution the right ratio between $E[X^2]$ and $E[X^3]$.

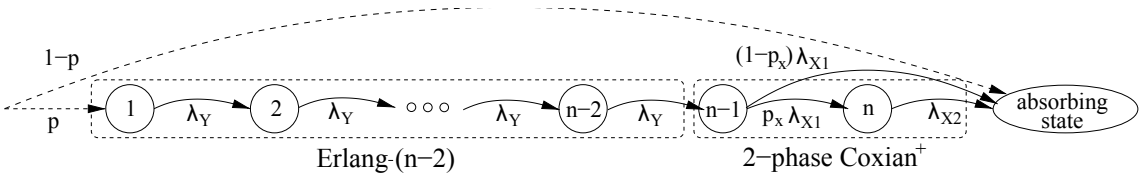


Figure 3.10: Phase structure for the moment matching, source: [15]

The matching algorithm finds values for λ_{X1}, λ_{X2}, λ_Y, p_X and the number of phases in the Erlang part (n) to fit the phase-type distribution of Figure 3.10 have the same moments as the moments provided. The algorithm outputs the distribution that is found as a phase-type distribution with a vector $\vec{a}$ and a matrix $\mathbf{T}$.

When considering the waiting time of an M|M|1 queue, there is a positive probability of having a waiting time of zero (when the server is empty). This behaviour cannot be derived from the moments, as a zero waiting time does not add to the moments. To increase the accuracy of the matching, the probability of having having to wait can be added to the matched distribution by setting the value for p.

The advantages of this moment matching algorithm are:

- It has been proven that the number of required phases by this algorithm is at most 1 more than the minimum number of required phases to match the original distribution.
- It is well explained and documented.
- It uses 3 moments, where some other algorithms use only 2 moments, resulting in higher accuracy.

Detailed information on the matching algorithm can be found in [15].

4 Single block analysis

An ASD based system has to execute different tasks. An example of a task can be the response of a system to someone pushing a button. The performance of the system is characterised by the time it takes for the system to respond to such a task. The response time of a task is specified as the time between starting it and finishing it completely.

This chapter starts with an introduction to the behaviour of an ASD block in Section 4.1. Then the analysis of a single block is discussed in Sections 4.2 and 4.3. The chapter ends with the method developed for derivation of waiting times for different calls to be executed on the same block in Sections 4.4 and 4.5.

4.1 Single ASD block characteristics

As listed in the previous section, the response time of a task is composed of the response times of multiple blocks. Therefore, the response time of each block is required.

A program generated by ASD consists of several ASD blocks. Each type of task follows its own characteristic path through the ASD structure. An example of the timeline of a task is shown in Figure 4.1. The task starts with a synchronous call at block A in an ASD architecture. After finishing at block A, a synchronous call is done on block B, after this one has returned, it is followed by a synchronous call at block C. The return of block C initiates a synchronous call to be processed on block B.

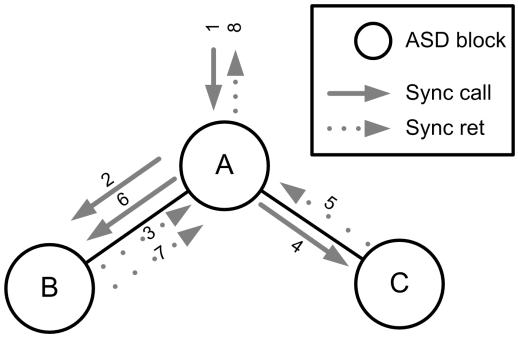


Figure 4.1: Path through the architecture

Figure 4.1 gives the path the tasks takes through the architecture, but in this thesis we are more interested in the response time of the task. Therefore a timeline of the situation is shown in Figure 4.2. In this figure for each block a line in present with the events happening at a block. The blocks with a “P” indicate processing that is going on, the blocks with a “B” indicate blocking. As long there is no block between two events, no time elapses, i.e., no time elapses between the return indicated by 5 and the call indicated by 6.

So the task starts with processing on block A, next block B starts processing and block A is blocked. While block C starts processing, block A remains blocked, but block B is free again and after block B has done the final processing the task finishes. However the blocking may interfere with other task, from this timeline it can be seen that the processing at the individual blocks determines to response time.

By stripping all the elements that not contribute to the response time, the timeline of Figure 4.2 can be simplified to the time line of Figure 4.3, which only indicates the processing parts.

This thesis only discusses the analysis of tasks consisting of either synchronous or asynchronous calls. Combinations of asynchronous calls and synchronous calls within one task are not investigated yet, to start simple.

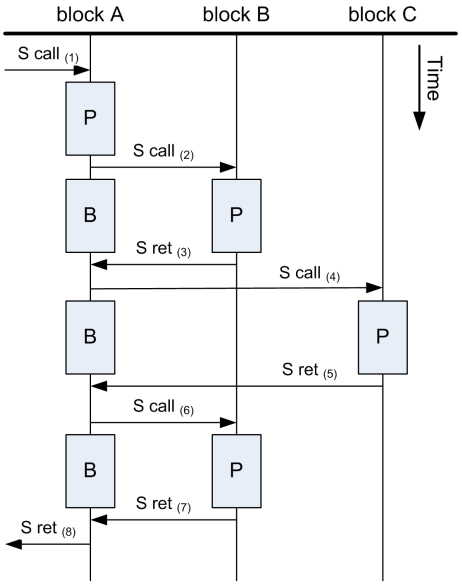


Figure 4.2: Timeline of the path of Figure 4.1

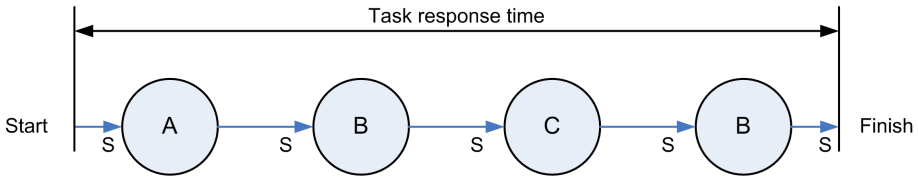


Figure 4.3: Composition of the response time of a task

4.1.1 Structural behaviour

As discussed in Chapter 2, synchronous calls introduce blocking. When a block issues a synchronous call to its slave, the block itself is blocked until the slave block has finished processing the call.

Asynchronous calls however, do not cause blocking. As soon as a block finishes processing an asynchronous call and issues a call to another block, the block is free to process new calls.

These effects are illustrated in Figure 4.4. A task with only synchronous calls is currently processed at the grey block. The parent node is blocked until the execution at the grey block finishes. Meanwhile an asynchronous call arrives at the grey block, which belongs to another task.

The bottom blocks in the tree, as shown in Figure 4.4, cannot be blocked, because they have no slaves to issue synchronous calls to. In this chapter a model is build that does not incorporate blocking. So the results of this chapter only adequately model the behaviour of the bottom blocks in the tree.

4.1.2 Block behaviour

The behaviour of an ASD block can be modelled as a special queueing system, as displayed in Figure 4.5. Both types of calls, synchronous and asynchronous, have their own queue and asynchronous calls have priority over synchronous calls. The asynchronous calls arrive at the top queue (the black jobs), that is modelled to be infinite, according to a Poisson process with a specified rate. If the server is busy jobs are queued, otherwise they are served immediately. When their service is finished they leave the block.

In the queuing station the synchronous calls are shown in grey. The queueing station can only hold one synchronous job, that is either waiting for service, in service, or experiencing its think

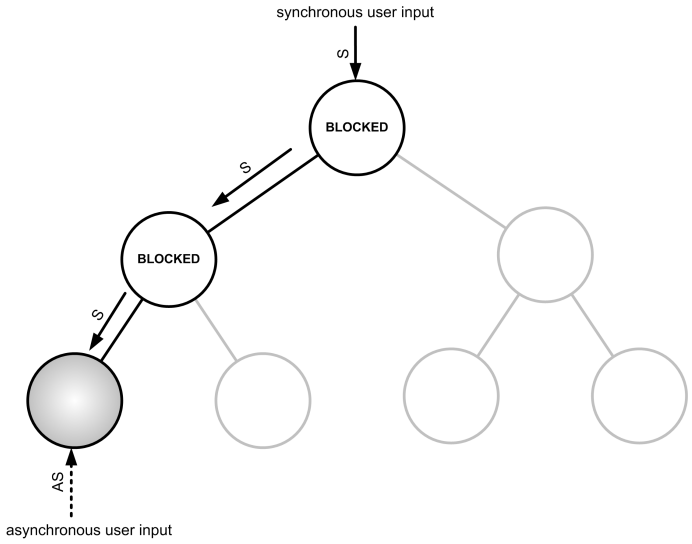


Figure 4.4: Blocking with synchronous calls

time. Hence, this is modelled as a closed system. The think time models the time it takes between the finishing of a synchronous call and the arrival of the next synchronous call. Think time represents the time the user "thinks" between the moment a synchronous call finishes and a new synchronous call is put into the system. Making the think time 0 would make the system always busy. When modelling multiple blocks, this think time can also represent the service time of other blocks.

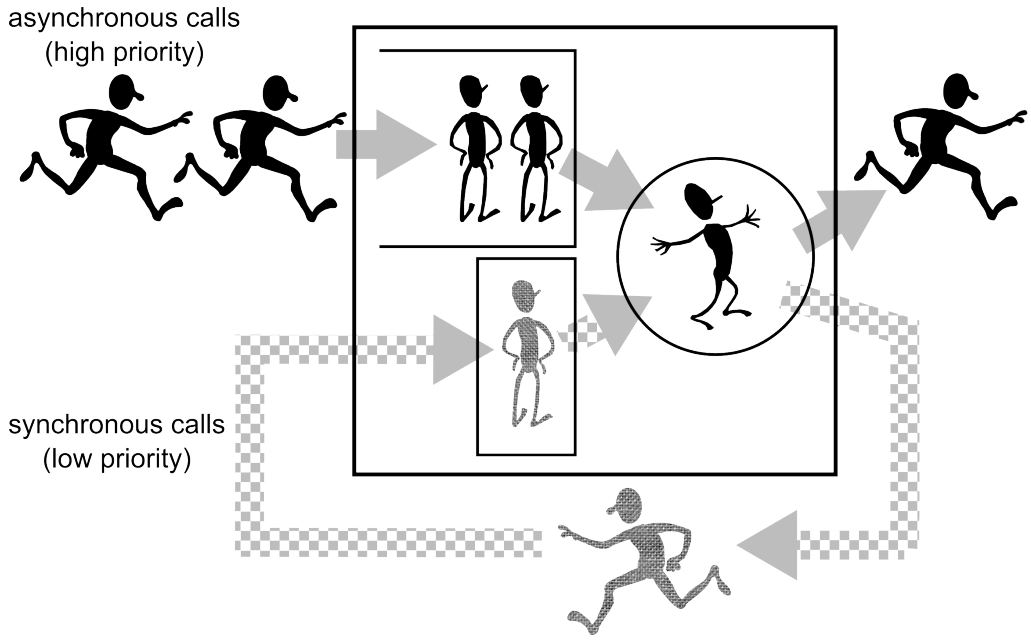


Figure 4.5: Queue representation of a single ASD block

Since at a single block there cannot be more than one synchronous call present, the inter-arrival time of a synchronous call consists of three parts: The think time, waiting time and service time. This is graphically shown in Figure 4.6. At the top line the inter-arrival time of a synchronous call is shown, which consists of its think time, the time it has to wait for all asynchronous calls to be processed and its service time. The later is shown in the second line of the figure.

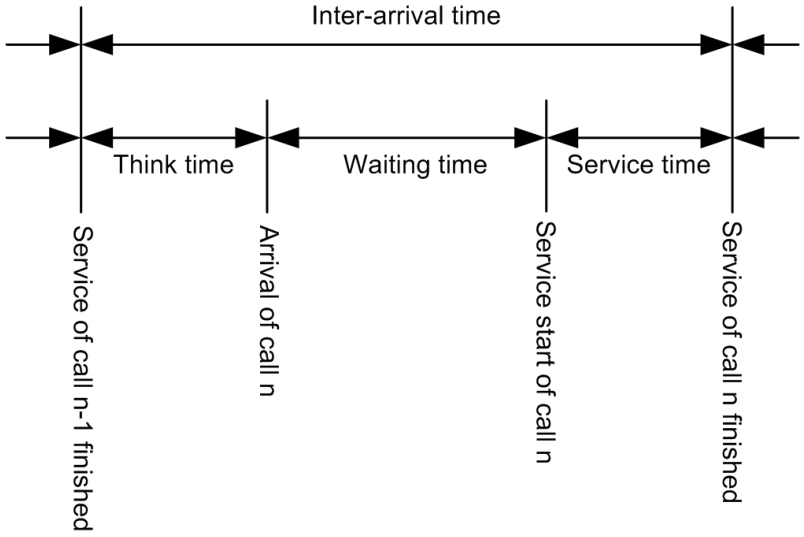


Figure 4.6: Timeline for the synchronous calls

4.1.3 M|G|1 with vacations

At first sight the behaviour of the asynchronous calls appears to match the M|G|1 model with vacations [5], in which the server takes vacations when the queue is empty. During this vacation, the server is unavailable. The duration of the vacations has a certain distribution. When mapping this to the ASD block, the vacations would model the behaviour of the synchronous calls.

There are two main types of these vacation systems, which are described in more detail [5] and [9]:

M|G|1 with multiple vacations A server that returns from vacation and finds the server empty takes another vacation.

M|G|1 with a single vacations When the server finds the queue empty, it takes only one vacation. When it returns from its vacation and finds the server still empty it does not take another vacation.

Both variants of the vacation model do not completely resemble the behaviour of a single ASD block. The first assumes that two synchronous calls can follow each other directly with zero think time, which is not true in an ASD block. The single vacation model is also not modelling the behaviour of the ASD block correctly, since with a low arrival rate for the asynchronous calls it is possible that multiple synchronous calls (vacations) have to be processed between two asynchronous calls.

4.1.4 Outlook on the applied analysis methods

The analysis of the queuing station of Figure 4.5 is based on Cobham's formula as discussed in Section 3.10, which can handle an arbitrary distributed service time. Cobham's formula assumes Poisson arrivals, however this is not the case in our model of an ASD block due to the think time and the restricted population of one synchronous call. To overcome this problem, several measures have to be derived from the underlying CTMC, which can only be derived when all the transitions are negative-exponentially distributed.

Therefore the service times of the calls is assumed to be negative exponentially distributed. Also inter-arrival time for the asynchronous calls is assumed to be negative exponentially distributed. For the synchronous calls the think time (the time between leaving the server and re-entering the block again) is assumed to be negative exponentially distributed.

The think and service times of the synchronous call are negative exponentially distributed. However, the waiting time of the synchronous calls is phase type distributed, as it consists of the service times of the asynchronous calls, which are negative exponentially distributed. The inter-arrival times for synchronous calls is than a convolution of the think time, waiting time and service time distributions, which results in another phase type distribution. The expected inter-arrival time for the synchronous calls $E[IA_s]$ (Equation 4.1) is the sum over the expected think time $E[Z_s]$, the expected waiting time $E[W_s]$ and the expected service time $E[S_s]$.

$$E[IA_s] = E[Z_s] + E[W_s] + E[S_s] \tag{4.1}$$

4.2 Waiting time analysis

To calculate the total waiting time for the two categories of calls, the basic equation for non-pre-emptive priority scheduling [3] can be used, given in Equation 3.9. Adjustments will be made to this formula to cope with the arrival distribution of the synchronous calls.

Asynchronous calls have the highest priority, so they only have to wait for asynchronous calls in the queue. Calls that are already in service are not interrupted. So when a synchronous call is in the server upon arrival of an asynchronous call, it has to wait until the processing of the synchronous call is finished.

The waiting time for the asynchronous calls can be derived by filling in the terms $E[T_P]$, $\sum_{k=1}^{r} E[T'_k]$ and $\sum_{k=1}^{r-1} E[T''_k]$ of Equation 3.9.

The term $E[T_P]$ accounts for the work in the server when an asynchronous call arrives. At the arrival of an asynchronous call, either a synchronous or an asynchronous call can be in the server. According to Equation 3.10 the contribution by the asynchronous calls is $\frac{1}{2}\lambda_{as}E[S_{as}^2] = \frac{\lambda_{as}}{\mu_{as}^2}$ and the synchronous calls contribute $\rho_s \frac{E[S_s^2]}{2E[S_s]} = \frac{1}{E[IA_s]\mu_s^2}$. The remaining service time is then given by $E[T_{P\,as}]$, which is the weighted sum as displayed by Equation 4.2,

$$E[T_{P\,as}] = \rho_{as} \cdot \frac{1}{\mu_{as}} + \rho_s \cdot \frac{1}{\mu_s} = \frac{\lambda_{as}}{\mu_{as}^2} + \frac{1}{\mu_s^2 \cdot E[IA_s]}. \tag{4.2}$$

The second term $\sum_{k=1}^{r} E[T_k^{'}]$ describes the work an arriving call finds in the queues of the same and higher priority. Since, the asynchronous calls have the highest priority, they only have to wait for the processing of calls from their own class. This is given by Equation 4.3,

$$\sum_{k=1}^{r} E[T_{k\,as}^{'}] = \rho_{as} \cdot E[W_{as}]. \tag{4.3}$$

As asynchronous calls have the highest priority, no jobs of higher priority can arrive while an asynchronous call waits for service. Meaning $\sum_{k=1}^{r-1} E[T_k^{''}] = 0$ in this case.

The waiting time of an asynchronous call is expressed as Equation 4.4, which is derived by filling in the derived terms in Equation 3.9. This results in the waiting time for the asynchronous calls given by Equation 4.5.

$$E[W_{as}] = \frac{\lambda_{as}}{\mu_{as}^2} + \frac{1}{\mu_s^2 \cdot E[IA_s]} + \rho_{as}E[W_{as}] \tag{4.4}$$

$$E[W_{as}] = \frac{\frac{\lambda_{as}}{\mu_{as}^2} + \frac{1}{\mu_s^2 \cdot E[IA_s]}}{(1-\rho_{as})} \tag{4.5}$$

Because the inter-arrival time of the asynchronous calls is exponentially distributed, the PASTA property holds. So for the expected utilization and queue length at arrival, the time average can be taken.

The synchronous calls have low priority, so when a call arrives at a block it has to wait until the block is empty, before it goes into service. Because each block contains at most one synchronous call, it only has to wait for the service of asynchronous calls currently present and those arriving during waiting.

The waiting time for the synchronous calls can also be derived by filling in the elements of Equation 3.9.

Upon the arrival of a synchronous call, only an asynchronous call can be in the server, because the ASD structure only allows one synchronous call at a block. The remaining processing time of an asynchronous call at an arbitrary moment during processing is described by $E[S_{as}^2]/2E[S_{as}]$, which equals $1/\mu_{as}$. Multiplying this by the probability of finding the server busy at arrival, the contribution of the current call in the server to the waiting time can be obtained.

For synchronous calls the PASTA property does not hold, because the inter-arrival times are not negative exponentially distributed. This means the arriving calls do not see the queue in equilibrium. Hence, the probability of finding a call in service at the arrival of a new call, is no longer equal to the utilization. The probability of an arriving synchronous call finding the server busy, has to be calculated otherwise and will be called r_{as}, resulting in Equation 4.6 for $E[T_{Ps}]$. Because the PASTA property does not hold, this r_{as} is not equal to the time average utilization ρ_{as}. The derivation of r_{as} will be discussed later on.

$$E[T_{Ps}] = r_{as} \cdot \frac{E[S_{as}^2]}{2E[S_{as}]} = \frac{r_{as}}{\mu_{as}} \tag{4.6}$$

A remark has to be made when using Equation 4.6 in case $E[S_{as}]$ is much larger than $E[Z_s]$. In Equation 4.6 the remaining work is given by $E[S_{as}^2]/2E[S_{as}]$. This formula is based on the independence between the moment of arrival of a synchronous call and the moment at which the processing of an asynchronous call started. However, depending on the parameter choices, the assumption of independence may not be correct. After a synchronous call is served, two things happen; (i) the first asynchronous call from the queue is put into service, and (ii) the think time of the synchronous call is initiated. In case the expected service time of an asynchronous call is much greater than the think time of a synchronous call $E[S_{as}] \gg E[Z_s]$, this dependency is strong. This is because, upon arrival of the next synchronous call, the remaining service time of the asynchronous call is larger than the average value $E[S_{as}^2]/2E[S_{as}]$. Otherwise, the error induced by assuming this independence is negligible.

Next, the expected amount of work which an arriving synchronous call finds in the queues of equal and higher priority has to be derived. Recall, that the PASTA property does not hold for synchronous calls, the system is not in equilibrium at the arrival of a synchronous call, so the queue length at the arrival of a synchronous call $E[N_{qA}]$ is not equal to the time average. Therefore $E[N_{qA}]$ is not so easy to calculate and its computation will be discussed later on. By assuming $E[N_{qA}]$, the time required to process the queue that is found upon the arrival of a synchronous call can be calculated using Equation 4.7.

$$\sum_{k=1}^{r} E[T_k^{'}] = E[N_{qA}] \cdot E[S_{as}] = \frac{E[N_{qA}]}{\mu_{as}} \tag{4.7}$$

Since asynchronous calls have priority over synchronous calls, any asynchronous call arriving while the synchronous call waits for service, is processed before the synchronous call goes into service. While a synchronous call is waiting, on average $\lambda_{as}E[W_s]$ asynchronous calls arrive. These calls will require each $E[S_{as}]$ processing time composing Equation 4.8.

$$\sum_{k=1}^{r-1} E[T_k''] = \lambda_{as} E[W_s] E[S_{as}] = \rho_{as} E[W_s] \tag{4.8}$$

All these components combined lead to Equation 4.9, which can be rewritten into Equation 4.10.

$$E[W_s] = \frac{r_{as}}{\mu_{as}} + \frac{E[N_{qA}]}{\mu_{as}} + \rho_{as} E[W_s] \tag{4.9}$$

$$E[W_s] = \frac{r_{as} + E[N_{qA}]}{\mu_{as}(1-\rho_{as})} \tag{4.10}$$

For the expected inter arrival time ($E[IA_s]$), synchronous waiting time ($E[W_s]$) and asynchronous waiting time ($E[W_{as}]$) equations are derived, but none of them can be calculated while values for r_{as} and $E[N_{qA}]$ are missing. These values can be derived from the underlying CTMC, as explained in the next section.

4.3 CTMC model of an ASD block

The underlying CTMC of the queue representation of a single ASD block (c.g. Figure 4.5) is displayed in Figure 4.7.

In the states of the CTMC, the first number denotes the amount of synchronous calls in the queue and the second number the amount of asynchronous calls in the queue. The letters *AS* mean, that there is an asynchronous call in the server, *S* that means a synchronous call is in the server and an *E* denotes the empty server.

The first row (states 0,n,S) lists all states with a synchronous call in the server, the second row (states 0,n,AS and 0,0,E) describes the states without a synchronous call present in the block and the bottom row (states 1,n,AS) lists the states with a synchronous call waiting. The columns denote the amount of asynchronous calls present in the block. In the first column (level 0), no asynchronous call is present in the block and in the i-th column (level $i-1$) $i-1$ asynchronous calls are present the block. The CTMC is infinite as the buffer of asynchronous calls is assumed to be infinite.

In this CTMC the following transition rates are used:

$z_s = \frac{1}{E[Z_s]}$ The think rate for synchronous calls.

$\mu_s = \frac{1}{E[S_s]}$ The service rate for synchronous calls.

$\lambda_{as} = \frac{1}{E[IA_{as}]}$ The arrival rate for asynchronous calls.

$\mu_{as} = \frac{1}{E[S_{as}]}$ The service rate for asynchronous calls.

In case the system is empty (state 0,0,E) either a synchronous call can arrive (transition with rate z_s) or an asynchronous call can arrive with rate λ_{as}, this is represented by the outgoing transitions. In case a synchronous call is in service and the asynchronous queue is empty (state 0,0,S), either an asynchronous call can arrive with rate λ_{as} or the service of the synchronous call is finished (transition with rate μ_{as}).

In general, if there is no synchronous call in the system, a new one can arrive with rate z_s and when it is in the server is can be served with rate μ_s. In every state a new asynchronous call can arrive with rate λ_{as} and when an asynchronous call is currently in the server it can be served with rate μ_{as}.

From this CTMC directly an expected waiting time can be calculated, however this requires the CTMC to be finite. In Section 4.5 a method of calculating the expected waiting time for synchronous calls and the second and third moments for waiting time of synchronous calls directly

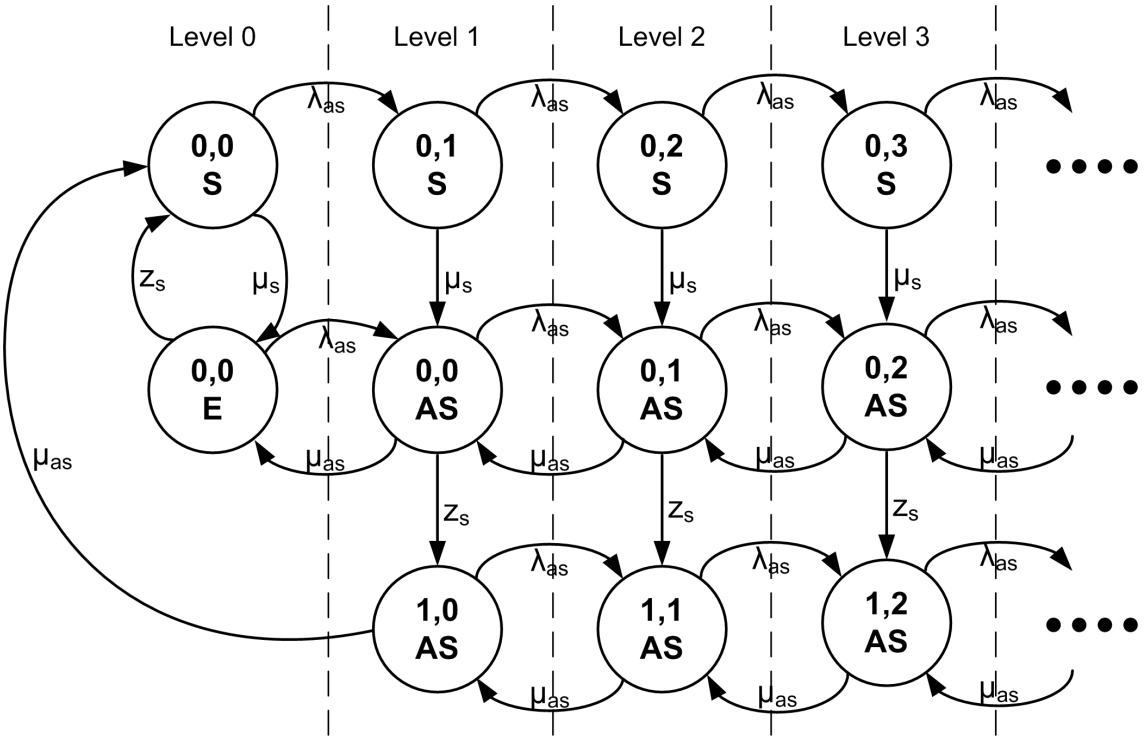


Figure 4.7: ASD CTMC levels

from this CTMC is discussed in Section 4.5. The adapted version of Cobham's formula allows analysis based on the infinite CTMC's and is therefore used as the algorithm to calculate the expected waiting time for synchronous calls. Section 4.4 explains how the required measures for this formula (r_{as} and $E[N_{qA}]$) can be calculated from the CTMC. The adjusted Cobham's formula does not give the second and third moments, so these are directly computed from the CTMC.

4.4 Deriving measures of interest

The values for r_{as} and $E[N_{qA}]$ can be obtained from the CTMC model of a single ASD block. The probability that there is an asynchronous call in service and no synchronous call in the system, i.e. the utilization r_{as}, can be calculated from the steady state probabilities. In the following the steady state probabilities are denoted as $\vec{p}$, with entries $p_{0,n,S}$ for states in the upper row of the CTMC, entries $p_{0,n,AS}$ and $p_{0,0,E}$ for the middle row and entries $p_{1,n,AS}$ for the lower row. Note that $n \in \mathcal{N}$.

To apply the matrix geometric method, first the levels in the CTMC have to be defined. The division into levels of the CTMC of an ASD block is shown in Figure 4.7. Except for level 0 all other levels have the same structure.

For this CTMC the state sub-vectors for the different levels are defined by Equation 4.11 and 4.12. Together they form the complete steady state vector $\vec{p}$, as shown in Equation 4.13.

$$\vec{z_0} = \begin{bmatrix} p_{0,0,S} & p_{0,0,E} \end{bmatrix} \tag{4.11}$$

$$\vec{z_i} = \begin{bmatrix} p_{0,i,S} & p_{0,i-1,AS} & p_{1,i-1,AS}, i > 0 \end{bmatrix} \tag{4.12}$$

$$\vec{p} = \begin{bmatrix} \vec{z_0} & \vec{z_1} & \vec{z_2} & \dots & \vec{z_i} & \vec{z_{i+1}} & \dots & \vec{z_\infty} \end{bmatrix} \tag{4.13}$$

The corresponding block matrices can be derived from the CTMC and listed are below. Together they form the **Q** matrix as displayed in Equation 3.21.

$$
\begin{aligned}
\mathbf{B_{00}} &= \begin{bmatrix} -(\lambda_{as}+\mu_s) & \mu_s \\ z_s & -(\lambda_{as}+z_s) \end{bmatrix} \\
\mathbf{B_{01}} &= \begin{bmatrix} \lambda_{as} & 0 & 0 \\ 0 & \lambda_{as} & 0 \end{bmatrix} \\
\mathbf{B_{10}} &= \begin{bmatrix} 0 & 0 \\ 0 & \mu_{as} \\ \mu_{as} & 0 \end{bmatrix} \\
\mathbf{A_0} &= \begin{bmatrix} \lambda_{as} & 0 & 0 \\ 0 & \lambda_{as} & 0 \\ 0 & 0 & \lambda_{as} \end{bmatrix} \\
\mathbf{A_1} &= \begin{bmatrix} -(\lambda_{as}+\mu_s) & \mu_s & 0 \\ 0 & -(\lambda_{as}+\mu_{as}+z_s) & z_s \\ 0 & 0 & -(\lambda_{as}+\mu_{as}) \end{bmatrix} \\
\mathbf{A_2} &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & \mu_{as} & 0 \\ 0 & 0 & \mu_{as} \end{bmatrix}
\end{aligned}
$$

Matrix $\mathbf{B_{00}}$ describes the transition within level 0. The $\mathbf{A_1}$ matrix describes the transitions within the repeating levels. Block matrix $\mathbf{A_0}$ describes the transitions to the next level and matrix $\mathbf{A_2}$ the transitions to the previous level. Because z_0 and z_i do not necessarily have the same size, matrices $\mathbf{B_{01}}$ and $\mathbf{B_{10}}$ describe the transitions from level 0 to level 1 and back.

Looking at the $\mathbf{B_{00}}$ matrix the upper left term is $-(\lambda_{as}+\mu_s)$, because from state $p_{0,0,S}$ their are two outgoing transitions with rates λ_{as} and μ_s. The μ_s goes to state $p_{0,0,E}$ as listed in the top right element of this matrix. The λ_{as} transition goes to the next level, to state $p_{0,1,S}$ and is listed in the top left element of the $\mathbf{B_{01}}$ matrix. With the bottom row of the $\mathbf{B_{00}}$ matrix this goes in an almost similar way.

Using the matrix geometric method, the steady state probabilities for this CTMC can be derived. This method outputs the steady state probability sub-vectors $\vec{z}_0$ and $\vec{z}_1$ and the matrix **R**. From these results, all other the steady state probability sub-vectors can be calculated using Equation 3.34.

Once, the steady state probabilities are known, the measures needed for the calculation of the waiting times (r_{as} and $E[N_qA]$) and can be derived. First the probability that there is no synchronous call in the system is needed (the probability of being in the middle row in the CTMC). This can be calculated using Equation 4.14. In the first step of this derivation all relevant steady state probabilities are accumulated. They are then expressed in terms of the steady state sub-vectors in the second step. Using the recursive relationship between successive sub-vectors, as in Equation 3.34, this is rewritten using the **R** matrix. By applying the sum over the geometric series, finally Equation 4.14 is derived.

$$\begin{aligned}
\Pr(\text{no S call in system}) &= p_{0,0,E} + \sum_{n=0}^{\infty} p_{0,n,AS} \\
&= \vec{z}_0 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \sum_{i=1}^{\infty} \vec{z}_i \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \\
&= \vec{z}_0 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \vec{z}_1 \left(\sum_{i=0}^{\infty} \mathbf{R}^i \right) \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \\
&= \vec{z}_0 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \vec{z}_1 (\mathbf{I} - \mathbf{R})^{-1} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad (4.14)
\end{aligned}$$

The probability that an arriving synchronous call finds the server busy (r_{as}) can be calculated by dividing the steady state probability of having an asynchronous call in the server by the probability of having no synchronous call in the block. As shown in Equation 4.15, the first is computed by accumulating the steady state probabilities of all states $p_{0,n,AS}$ for $n \in \mathcal{N}$. Because of the denominator that ensures that no synchronous is present, only states from the second row have to be taken into account. Hence, the equation can be rewritten to the second line as the counter probability of having an empty server under the same condition. The steady state probability of having an empty server can be rewritten, using the sub-vector $\vec{z}_0$. This results in Equation 4.16.

$$\begin{aligned}
r_{as} &= \frac{\sum_{n=0}^{\infty} p_{0,n,AS}}{\Pr(\text{no S call in system})} \qquad (4.15) \\
&= 1 - \frac{p_{0,0,E}}{\Pr(\text{no S call in system})} \\
&= 1 - \frac{\vec{z}_0 \begin{bmatrix} 0 \\ 1 \end{bmatrix}}{\Pr(\text{no S call in system})} \qquad (4.16)
\end{aligned}$$

The expected number of asynchronous calls in queue an arriving synchronous call sees ($E[N_q A]$) can be calculated using Equation 4.17. The derivation starts by multiplying the number of customers n by the probability n asynchronous calls in the queue under the condition of having no synchronous calls in the block. This is written as function of $\vec{z}_i$ in the second line. Next it is split in two, making it a function of r_{as}.

$$
\begin{aligned}
E[N_{qA}] &= \sum_{n=0}^{\infty} n \cdot \frac{p_{0,n,AS}}{p(\text{no S call in system})} \\
&= \frac{\sum_{i=2}^{\infty}(i-1)\cdot\vec{z}_i \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}}{\Pr(\text{no S call in system})} \\
&= \frac{\left(\sum_{i=1}^{\infty} i\cdot\vec{z}_i\right)\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}}{\Pr(\text{no S call in system})} - \frac{\left(\sum_{i=1}^{\infty}\vec{z}_i\right)\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}}{\Pr(\text{no S call in system})} \\
&= \frac{\vec{z}_1(\mathbf{I}-\mathbf{R})^{-2}\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}}{\Pr(\text{no S call in system})} - r_{as}
\end{aligned} \tag{4.17}
$$

More details on the last step in the derivation are given in Equation 4.18.

$$
\begin{aligned}
\sum_{i=1}^{\infty} i\cdot\vec{z}_i &= \vec{z}_1\left(\sum_{i=1}^{\infty} i\mathbf{R}^{i-1}\right) \\
&= \vec{z}_1\left(\sum_{i=1}^{\infty}\frac{\partial}{\partial\mathbf{R}}\mathbf{R}^{i}\right) \\
&= \vec{z}_1\frac{\partial}{\partial\mathbf{R}}\left(\sum_{i=1}^{\infty}\mathbf{R}^{i}\right) \\
&= \vec{z}_1\frac{\partial}{\partial\mathbf{R}}\left((\mathbf{I}-\mathbf{R})^{-1}-\mathbf{I}\right) \\
&= \vec{z}_1(\mathbf{I}-\mathbf{R})^{-2}
\end{aligned} \tag{4.18}
$$

Since the values for r_{as} and $E[NqA]$ have been derived, the adjusted Cobham's formula can be applied to calculate the expected waiting times of a single ASD block. These formula's are applied and validated in Section 6.1.

4.5 Moments of the waiting time distribution

In later analysis, not only the expected waiting time for the synchronous calls ($E[W_s]$) is required, but also the second and third moment ($E[W_s^2]$ and $E[W_s^3]$) are required. Cobham's formula only provides an expression for the first moment of the waiting time. The second and third moment have to be derived differently.

First, we follow the path that is taken by a synchronous call in the CTMC model, from arrival until it enters the server. In Figure 4.8 the CTMC model of an ASD block is shown.

A synchronous call enters the system with rate z_s, modelled by a transition to the third row. The CTMC then remains in the lowest row until all asynchronous calls are served and the synchronous call is put into the server. This is modelled the transition to state $(0,0,S)$ (grey in the figure), that is equipped with rate $\mu_a s$.

When a synchronous call arrives, the server is either empty or busy. In case the synchronous call arrives to an empty system, its waiting time is zero and hence, does not contribute to the second and third moment $E[W_s^2]$ and $E[W_s^3]$.

A synchronous call that arrives at a busy server, is modelled by a transition in the CTMC from

the middle row to the bottom row (transition with rate z_s has taken place). The call has to wait until the asynchronous queue becomes empty, i.e. the CTMC being in state [0,0,S] (grey state).

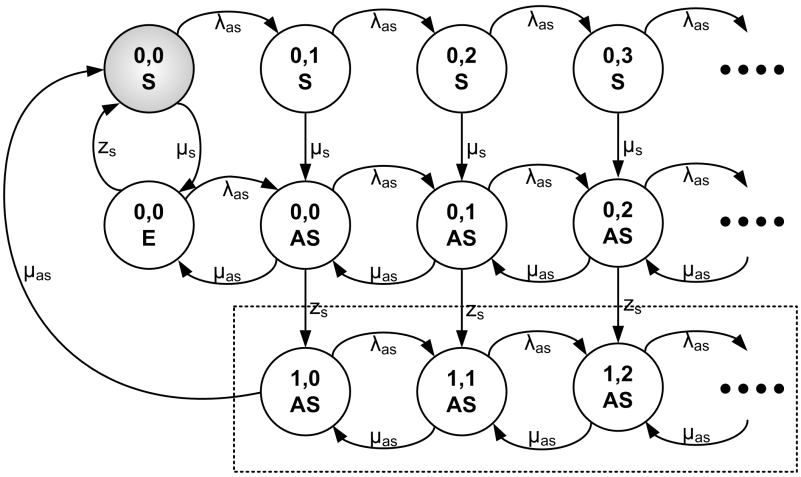


Figure 4.8: Waiting trajectory of a synchronous call in the CTMC

To compute the higher moments of the synchronous waiting time, only the part of the CTMC involved in the waiting process of a synchronous call has to be taken into account. This part is shown in Figure 4.9.

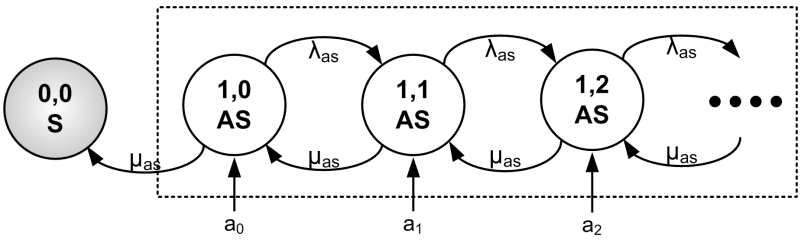


Figure 4.9: CTMC part involved in the waiting time of a synchronous call

This CTMC represents a phase type distribution with an infinite number of phases. The values a_i indicate the state of the asynchronous queue when the the synchronous call arrives, which is the initial distribution in this CTMC. This a_i is calculated as the probability of having i asynchronous calls in the queue under the condition that no synchronous call is currently in the system, as listed in Equation 4.19.

$$a_i = \frac{p_{0,i,AS}}{p(\text{no S call in system})} \tag{4.19}$$

Using Equation 3.36, the i-th moment of a phase type distribution can be calculated from the vector $\vec{a}$ and the matrix $\mathbf{T}$. In this case both the $\vec{a}$ vector and the $\mathbf{T}$ matrix are of infinite size, however Equation 3.36 can only deal with finite matrices and vectors.

By truncating the CTMC as shown in Figure 4.10, the size of the $\vec{a}$ vector and the $\mathbf{T}$ matrix is limited, i.e. limiting the number of asynchronous calls that can be in the queue. When the number of phases taken into account is sufficient, the error made by the truncation is neglectable. More information on the required number of phases is given in Section 4.5.1.

The truncated $\mathbf{T}$ matrix is given by Equation 4.20. The first row starts with $-\lambda_{as} - \mu_{as}$ indicating the outgoing transitions from state (1,0,AS). With rate λ_{as} new tasks arrive resulting in a transition to state (1,1,AS). Note, that $\mathbf{T}$ is not a proper probability matrix as it does not take

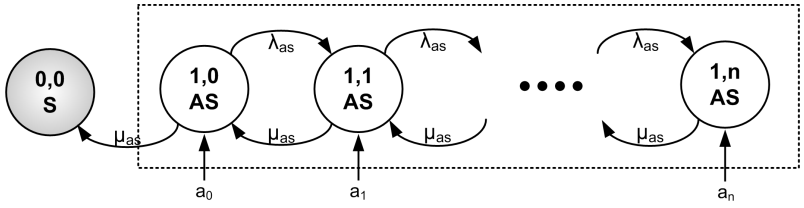


Figure 4.10: Truncated CTMC of synchronous waiting times

into account the absorbing state (0,0,S). This reflected in the diagonal element of the first row, which does not equal the negative row sum.

For the other states expect for state $(1,n,AS)$, there is a rate of λ_{as} going to the next state and μ_{as} going to the previous state. The number of phases is truncated, so no calls can arrive in the last state $(1,n,AS)$, therefore in the last row there is only transition to the previous state possible.

$$\mathbf{T} = \begin{bmatrix} -\lambda_{as}-\mu_{as} & \lambda_{as} & 0 & 0 & 0 & 0 \\ \mu_{as} & -\lambda_{as}-\mu_{as} & \lambda_{as} & 0 & 0 & 0 \\ 0 & \mu_{as} & -\lambda_{as}-\mu_{as} & \lambda_{as} & 0 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & \mu_{as} & -\lambda_{as}-\mu_{as} & \lambda_{as} \\ 0 & 0 & \cdots & 0 & \mu_{as} & -\mu_{as} \end{bmatrix} \tag{4.20}$$

From Equation 3.36 the equations 4.21, 4.22 and 4.23 for the waiting time moments $E[W_s]$, $E[W_s^2]$ and $E[W_s^3]$ can be derived.

$$E[W_s] = -\vec{a}\mathbf{T}^{-1}\vec{1} \tag{4.21}$$
$$E[W_s^2] = \vec{a}\mathbf{T}^{-2}\vec{1} \tag{4.22}$$
$$E[W_s^3] = -\vec{a}\mathbf{T}^{-3}\vec{1} \tag{4.23}$$

When $E[W_s]$ is calculated using Equation 4.21, the effect of the truncation can be analysed by comparing it with the $E[W_s]$ acquired using the steady state analysis. The error caused by the truncation is larger in the calculation of $E[W_s^2]$ and $E[W_s^3]$, because in these cases the $\mathbf{T}^{-2}$ respectively $\mathbf{T}^{-3}$ are used instead of $\mathbf{T}^{-1}$.

4.5.1 Required number of phases

In the graph of Figure 4.11 a comparison is presented between the waiting times produced by Cobham's formula and the results by truncated phase-type approach. Cobham's formula does not use truncation, so it results are independent on the number of phases, which results in a horizontal line. The phase type version however, becomes better as the number of included phases increases. For this graph, the following values are used: $z_s = 0.1$, $\mu_s = 0.5$, $\mu_{as} = 0.333$ and for λ_{as} three different values are used (resulting in three line sets): 0.1, 0.2 and 0.35. The corresponding utilizations are 0.42, 0.60 and 0.83.

Besides the first moment, the second and third moments are also calculated using the truncation method. Under the same conditions as used with Figure 4.11, the second and third moments are plotted against the number of phases taken into account in Figure 4.12 and Figure 4.13.

All graphs show that as the number of phases taken into account becomes larger, the estimated moments of the waiting times by the truncation method approach a certain asymptote, representing the values with an infinite number of phases. From these graphs it can be seen that the

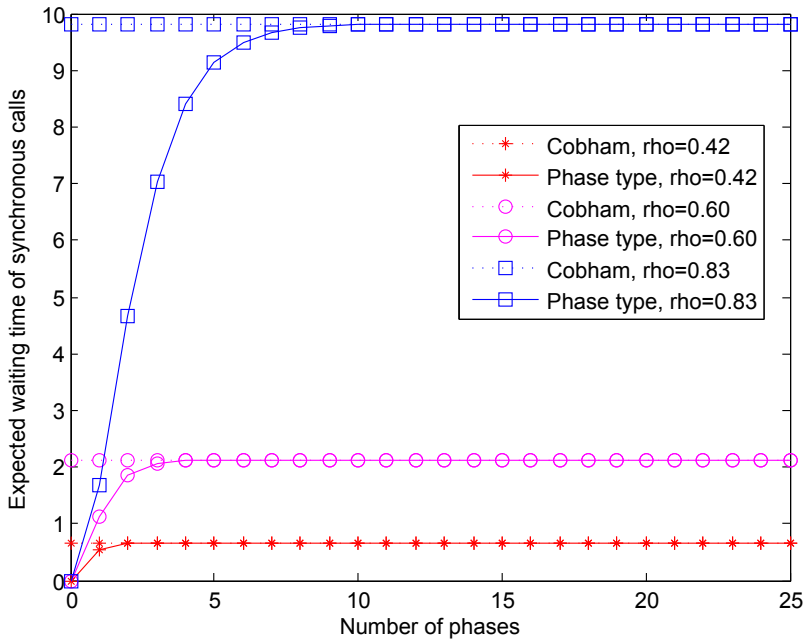


Figure 4.11: Waiting times calculated using Cobham versus phase-type

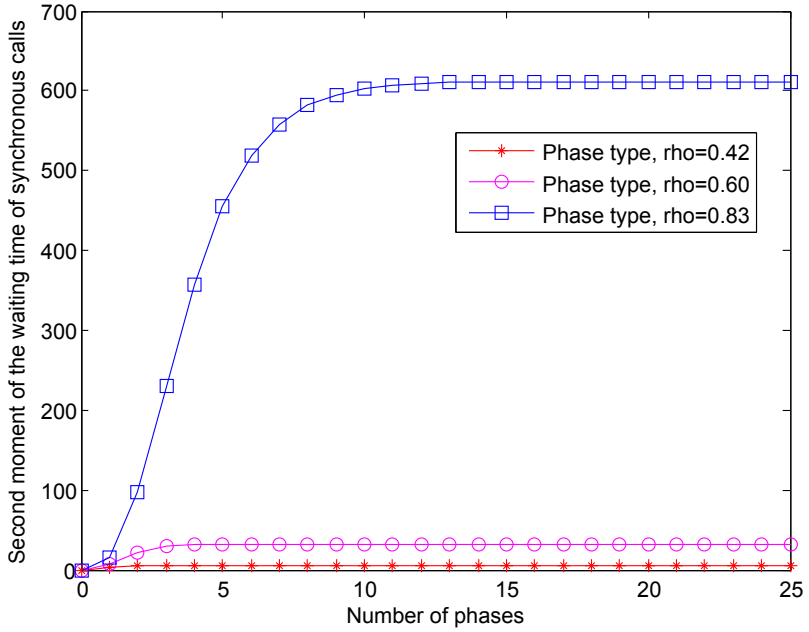


Figure 4.12: Second moment waiting times using phase-type

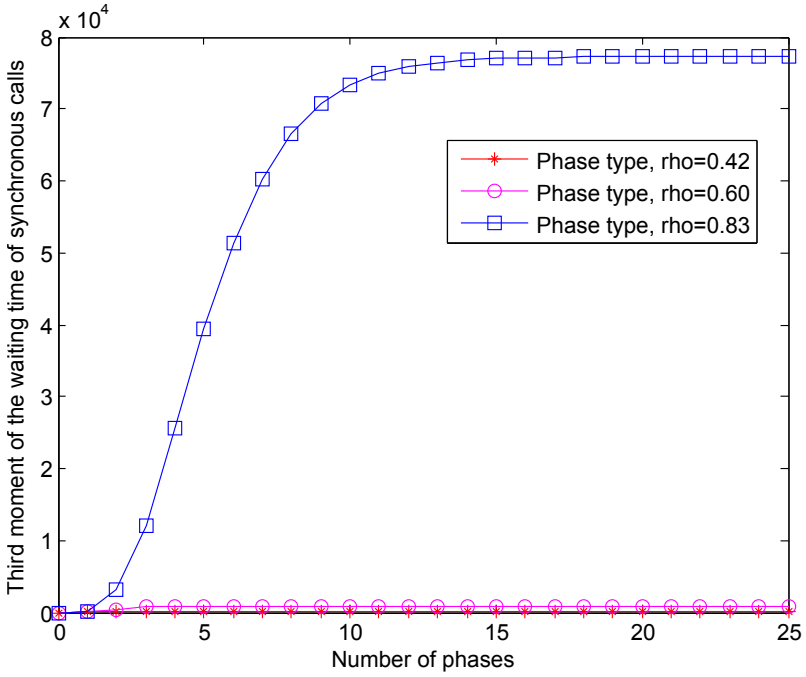


Figure 4.13: Third moment waiting times using phase-type

number of phases required to get a good estimate for the moments, is dependant on the utilization. Also the estimation of the third moments requires more phases to approach the final value as the estimation for the first moment.

For the number of phases required to get a good estimate for the moments the following rule of thumb can be derived:
Estimate the number of phases required for the expected waiting time to approach the value obtained using adapted Cobham's formula within a predefined margin and multiply this number of phases by a factor 3.

4.5.2 Related and alternative methods

The phase type method for the calculation of the second and the third moment of the waiting time, as presented in this section is not the only available method for deriving moments of the waiting time. Several other methods are presented in [10], to calculate the *first passage times* in quasi birth death process, which resembles the expected waiting times in this situation. These methods can be adapted to deliver the second and third moments as well.

Although these techniques work based on QBD processes, they only deliver the time it takes to travel from level $i \in \mathcal{N}$ to level $j \in \mathcal{N}$. Hence only a finite number of starting levels can be taken into account, which also truncates the CTMC in a certain way, resulting in an approximation. Nevertheless these techniques are probably faster to calculate, although they are harder to implement. So if fast analysis is important, these techniques can be used instead of the phase type approach, as presented here.

There also exists a technique called the "uniformization method" [8] to calculate not only the moments of a distribution, but the complete probability density function of a distribution. In this thesis the distribution is not required, so it would only be used to calculate the moments. For this purpose the "uniformization method" is not a good option, because it is an approximation and only works for a finite number of phases, where the phase type method is faster to calculate and delivers exact results.

5 Simulation

To validate the results acquired using analysis, a discrete-event simulator has been built. This simulator should behave like a real ASD-generated software. To be able to simulate the complete system, the blocks and their structure together with a number of tasks are inserted into the simulator. A model for processor scheduling is also built in, but can be switched off to validate the analytical results.

This chapter starts with the explanation of actual simulator in Section 5.1. Then the calculation of the confidence interval based on the simulation results is presented in Section 5.2, followed by the random number generator in Section 5.3. Finally, the implementation processor scheduling together with its validation test is discussed in Section 5.4.

5.1 Simulator description

For this thesis a dedicated simulator has been build. This simulator contains a virtual ASD system according to the specification of the simplified ASD structure as listed in Sections 2.3 and 2.4. The ASD-architecture is currently configured by directly changing the source code of the simulator. The architecture is then described by denoting for each block what its children are, as presented in Figure 5.1. By denoting for all the blocks what their children are, the complete architecture is described. In this figure Block A has two children, which are Block B and C.

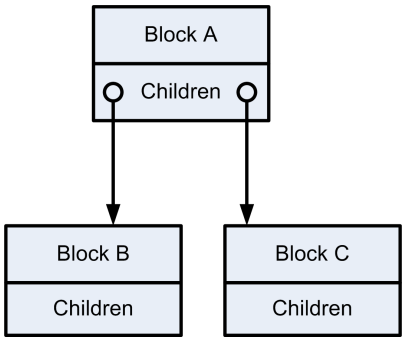


Figure 5.1: Children allocation

This virtual system is fed with several tasks, that are also configured by changing the source code of the simulator. A task is defined by an object, with the structure as shown in Figure 5.2. Each task contains the arrival / think rate and a list of calls the task consists of. Each call is of a certain type (synchronous or asynchronous), has an expected service time and states the block it has to be executed on. Although each call can be synchronous or asynchronous, the simulator can only simulate tasks either consisting completely of synchronous call or either only from asynchronous calls, because combinations would introduce parallelism, which makes simulation and analysis more difficult.

In the model the following entities are random variables:
- Think time of synchronous tasks
- Inter-arrival time of asynchronous calls
- Service time of synchronous calls
- Service time of asynchronous calls.

These variables are, as in the analytical model negative exponentially distributed, where the rate of the distribution is defined by the task or the call.

Since the system is fully defined now, the simulation can be performed. The simulation starts with an empty system and the tasks are scheduled for arrival. For the asynchronous tasks this is done by picking a sample from a negative exponential distribution with rate equal to the arrival

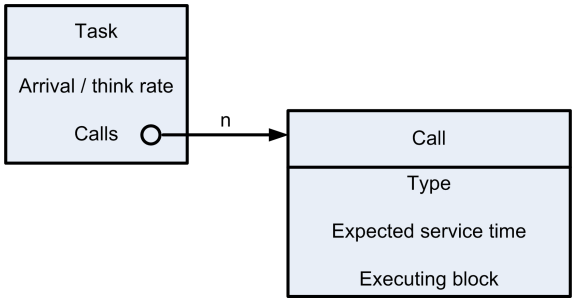


Figure 5.2: Task definition

rate.

For the synchronous tasks however, this is more complicated, because only one synchronous call can arrive. First the task to arrive has to be chosen. The probability of being chosen as task (p_i) is given by its think rate divided by the sum over the think rate of all synchronous tasks, as displayed in Equation 5.1. To decide which synchronous task has to arrive, first the interval $[0,1]$ is divided into segments. Each segment corresponds to a task and has the size of the probability of being chosen for arrival p_i. Second a uniform-distributed value between 0 and 1 is picked and it determined in which segment the picked value lies, as displayed in Figure 5.3. In this example Task 2 is chosen to arrive as first task.

$$p_i = \frac{z_i}{\sum_{j=1}^{k} z_j} \tag{5.1}$$

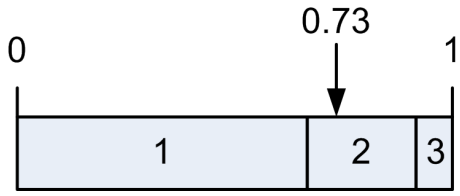


Figure 5.3: Implementation probabalistic choise

The think for this synchronous task is determined by drawing a sample from a negative exponential distribution with as rate the sum of all think rates.

Since the moment of arrival of the tasks is determined, the simulation can start. The simulator is implemented as a discrete-event simulator, meaning it processes the events according to their order and jumps between events, as displayed in Figure 5.4. The bend arrows in this figure indicate jumps in time which the simulator takes, as it only processes the events.

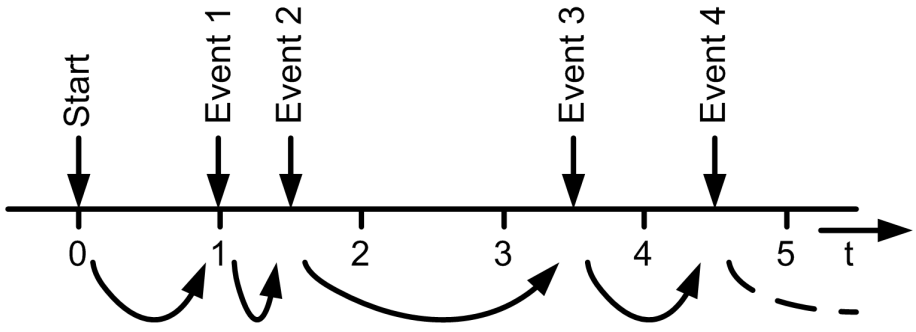


Figure 5.4: Discrete event simulator

To carry out the actual simulation, the simulator has to process these events and handle them in the right way and right order. To keep track of all the events, the simulator contains a list of all events and their moment of occurrence. Each type of event requires a different reaction. The different events and the required response are described below.

Arrival of an AS task The task arrives with its first call at a certain block. If the block is idle and not blocked, the call is directly put into service, otherwise it is queued. Also a new instance of this task is scheduled for arrival, as described in the initiation. This new arrival is added to the list of events.

Arrival of a S task The first call of this task arrives at a certain block, and if the block is empty, the call is directly put into service, otherwise it is queued. Note that: No new synchronous task is scheduled for arrival, as this is only done at the completion of a synchronous task.

Processing of an AS call finishes The call finishes at a certain block. From the task description it is determined if a call to another block has to be done or if the task is finished. At the block the new call has to be a "Arrival of next call" function is called. A "process next call" function is executed on the current block to fill the server again.

Processing of a S call finished This event requires a more complex reaction and is therefore explained by pseudo code in Listing 5.1. The reaction is dependant on the whether the next call has to be executed by a parent of this block or by a child of this block.

```
if (new S call in task?) {
  if (new S call to child of this block?} {
    This block is blocked until further notice.
  } else {
    if (new  S call to this block?) {
      Execute a ''Arrival of next call'' function on this block
    } else {
      Remove blockade of this block
      Execute a 'process next call' function on this block
      Start the same sequence at the parent block with the question:
        'new S call to child of this block?'
    }
  }
} else {
  Remove blockades issued by this call
  Execute a 'process next call' function at all the blocks that where previously blocked
  Schedule the arrival of a new S task
}
```

Listing 5.1: Pseudo code: "Processing of a S call finished" event

Because some event may have to execute the same procedure, these procedures are described once by the following functions:

Process next call If there are still AS calls in the queue, the first call is put into service, otherwise a S call can enter service. When both queues are empty, the block does nothing. When a call goes into service, it is removed from the queue and its service time is determined by picking a sample from a negative exponential distribution with according to the specification for this call. Using this service time, the moment this call finishes is calculated and listed in the event list.

Arrival of next call Directly after finishing the previous call of a task the new call of this task arrives at the next block. When this new call arrives at a block it is directly put into service if the block is idle and not blocked. The procedure of putting a call into service is more detail described by function "Process next call". When the block is not idle or blocked, the new call tasks place in the queue of its type of task.

The simulation continues until the simulated time (the time a real system would encounter) reaches a predefined end value. The occurrence times of the events are logged for statistical analysis in the end.

5.2 Statistical measures

To compare the simulation results with the analytical results, some statistical measures have to be calculated. The statistics obtained during the simulation are:

- Average waiting time per block for both asynchronous and synchronous calls
- Average queue length of asynchronous calls, upon arrival of a synchronous call
- Utilisation per block
- Response time of a complete task
- Arrival rate of tasks

Because of the large number of dependencies, the simulation is started with an empty system, as already mentioned earlier. The computation of statistical measures only starts after a predefined simulated time. Hereby the impact of the start-up effect on the statistics is reduced. The information as summarized in this section is discussed in more detail in [9].

With these statistical measures, a confidence interval is needed (a window in which the real expected value will be with a certain probability). The confidence interval is obtained, by running the complete simulation multiple times. For each run the measures listed above are determined and saved. The results are assumed to be Student-$n-1$ distributed, where n is the number of measurements.

The normal distribution is a special form of the Student distribution i.e. the normal distribution is the Student distribution for an infinite number of runs, so these distributions are very similar. A sketch of the probability density function (PDF) of the unit Student-10 distribution is shown in Figure 5.5. This graph shows the values on the x-axis and the corresponding probability density on the y-axis. The exact shape and PDF depends on the number n, but this sketch is made for $n = 10$.

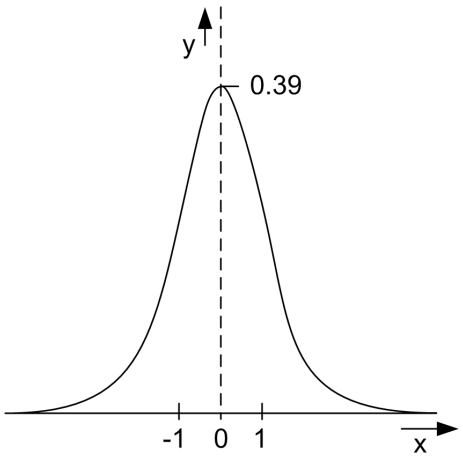


Figure 5.5: Sketch of the PDF of the Student distribution

The distribution obtained by simulation is never the unit Student distribution, but a transformed variant, as displayed in Figure 5.6. This PDF has a mean X and is stretched by a factor

S in the horizontal direction. Because this is a PDF, the area under the function should be 1, hence in the vertical direction the function is shrinked by a factor S.

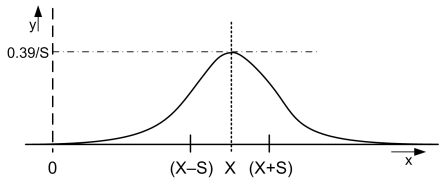


Figure 5.6: Real distribution based on the Student distribution

For further analysis, a the distribution of the simulation results should be mapped onto the unit Student-n distribution, therefore the average ($\tilde{X}$) and sample variance ($\tilde{S}$) need to be calculated from the obtained simulation results. The sample variance $\tilde{S}_{n-1}$ is calculated using Equation 5.2. In this equation, n is the number of samples and X_i, is the obtained value from simulation run number i.

$$\tilde{S}^2_{n-1} = \frac{1}{n-1}\sum_{i=1}^{n}\left(X_i - \tilde{X}\right)^2 \tag{5.2}$$

Using the sample variance and average, a value a from the real distribution can be mapped onto a corresponding value Z in the unit Student-n distribution as displayed in Equation 5.3.

$$Z = \frac{\tilde{X} - a}{\tilde{S}_{n-1}/\sqrt{n}} \tag{5.3}$$

A confidence interval is described by the values z_{min} and z_{max}, where $z_{min} = -z_{max}$. When a confidence interval of 90% is required, z_{min} and z_{max} have to be chosen such that the area under graph between z_{min} and z_{max} equals 0.9. Mapping a confidence interval to the Student distribution is displayed in Figure 5.7

Direct calculation of z_{min} and z_{max} is not possible, but z_{max} can be derived using the cumulative probability function (CDF) of the unit Student distribution. The CDF describes the probability (β) of having a value smaller than z, as shown in Equation 5.4. In a graphical perspective, the CDF gives the area under the graph of the PDF up to value z as shown in Figure 5.8.

$$Pr\{|Z| \leq z\} = \beta \tag{5.4}$$

The confidence interval does not contain the left part that is contained in the CDF, as can be seen by comparing Figure 5.7 and 5.8. Due to the symmetry in the PDF the value for β required to get the z_{max} with a given confidence interval(conf) is calculated by $\beta = \frac{1-\text{conf}}{2} + \text{conf} = \frac{1}{2} + \frac{1}{2}\text{conf}$.

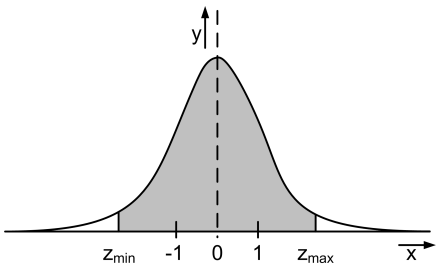


Figure 5.7: Confidence window expressed in the unit PDF

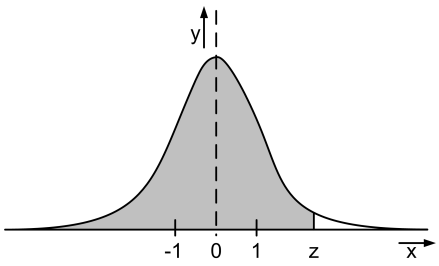


Figure 5.8: Cummalative probability density function

Normally, the values for z for a given value β are obtained using look-up tables. To fully automate the process, the simulator uses the Colt library [1] to calculate this z value for a given value for β. This Colt library is made for Java by the researchers of CERN. It contains a lot of mathematical functions, that are not standard available with Java. It contains functions for calculations with matrices and several functions for statistical analysis, including random number generators.

By transforming this z value from the unit distribution to the original distribution based on Equation 5.4, the confidence interval can be calculated, using Equation 5.5 and 5.6. Together these describe the confidence interval $[a_{min}, a_{max}]$.

$$a_{min} = \tilde{X} - z \cdot \tilde{S}/\sqrt{n} \tag{5.5}$$

$$a_{max} = \tilde{X} + z \cdot \tilde{S}/\sqrt{n} \tag{5.6}$$

Finally the measures as listed at beginning of this section together with their confidence windows are required. After the simulation is performed 10 times, for each run the measures of interest are determined. After 10 runs 10 values have been obtained for each measure. From these 10 values the confidence window is constructed as detailed out above. For each measure of interest the method of calculation is listed below:

Average waiting time per block for both asynchronous and synchronous calls For each call at a block account waiting time by taking the difference between the moment of arrival at that block and the moment it goes into service. Sort these waiting times by call type and take the average per type.

Average queue length of asynchronous calls, upon arrival of a synchronous call When a synchronous call arrives at a block, store length of the queue for asynchronous calls. Take the average over all these queue lengths to calculate the average.

Utilisation per block Sum (per block) up all the time the block is empty (sum over all times between becoming empty and start working again) and divide it over the total simulation length.

Response time of a complete task For each task calculate the time between arrival and finishing the last call. Finally take the average over all these times.

Arrival rate of tasks Account for each task the number of arrival during a simulation and divide it by the simulation time.

5.3 Random number generator

The random number generator used in the simulator is the Mersenne Twister [13]. Again, the implementation of the Colt library [1] is used. This random number generator uses a large period of $2^{19937} - 1 (= 10^{6001})$ and passes many stringent statistical tests.

To seed the random number generator a random value is used, that is generated by the Java random number generator. The Java random number generator is seeded by Java itself.

5.4 Processor scheduling

The real ASD generated software will run on a computer with a limited number of processors. However in the generated software multiple processes can be active at the same time. The scheduler of the operating system will handle this by assigning turn by turn the processor to one process.

While it is very hard to incorporate all effects caused by the scheduler into the analysis, it can be easily incorporated into the discrete event simulator. The effects of scheduling are built into the simulator using a simplified model, having only one processor. The application of scheduling can be switched on and off easily for validation of the analysis results. When the scheduling is switched off, each process is simulated as it runs on its on own "processor". Two scheduling models have been implemented; round robin-scheduling (RR) and processor sharing (PS).

In a normal computer, each process receives a small amount of processor time. This is closely modelled by the Round Robin (RR) scheduling. With RR-scheduling, when a task is selected for service, all other tasks are postponed till the end of the window. At the end of the window, the next task is selected for service. This scheduling mechanism is displayed in Figure 5.9(a), in which the time is on the horizontal axis and the two tasks are indicated with the different patterns. A task obtains the full processor, while the other wait again for its turn. When looking on a large scale it looks like everyone gets a certain share.

The implemented algorithm for RR-scheduling has a variable window size and is given in pseudo-code below.

```
// a function to handle the windowed processor sharing
void updateWindow() {
  activeTask = selectNextTask();

  // get the window size, stop when a task finishes
  windowSize = min(tasks[activeTask].finish - Tglobal,
       C_WINDOW_SIZE);

  // walk all other the tasks
  foreach task {
    if (task != tasks[activeTask]) {
      task.finish = task.finish + windowSize;
    }
  }
}
```

With the processor sharing model, each task gets a share of the processor, that is equally divided over the number of active tasks, as if they can use only a part of the processor and can run simultaneously. PS is displayed in Figure 5.9(b). In this case the two tasks (different patterns) run in parallel each using half the processor, each getting only half of the processing power. When one task has finished, the other one can use the full processor.

The RR-scheduling generates many events, which need to be handled by the simulator. This makes the time the simulation takes longer, especially when the window size becomes small. To overcome this problem, processor sharing is also implemented in the simulator. With PS all the active tasks receive an equal amount of calculation time, hence a task runs besides two others, receives one third of the normal processor time, lengthening its calculation time by a factor three. The shares have to be newly allocated every time a new call enters service or finishes, which generates much less events as RR-scheduling.

(a) Round robin scheduling (b) Processor sharing

Figure 5.9: Round Robin scheduling versus processor sharing

5.4.1 Validation of PS versus RR scheduling

In theory the round robin scheduling and processor sharing should provide the same results when the window size approaches zero. This is in fact the case with the simulator.

This is validated using a case, which is presented in Figure 5.10. In this case a synchronous task will go downwards and an asynchronous task will go upwards. The corresponding parameters are given in Table 5.1. Both calls indicated with A are executed on block A and the call indicated with B at block B. This means that the different tasks have to share blocks.

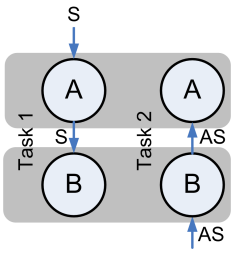


Figure 5.10: Case processor sharing task flow

Task number	Arrival / think rate	Expected service time
1	$z_s = 0.1$	$E[S_sA] = 3$ $E[S_sB] = 2$
2	$\lambda_{as} = 0.2$	$E[S_{as}B] = 2$ $E[S_{as}A] = 1$

Table 5.1: Case processor sharing task definition

The simulation is performed for a simulated time of 10^7 and all results before time 10^6 are not taken into account in the average values, to reduce the influence of the start-up. The 90% confidence window is constructed from 10 runs. Both a processor sharing (PS) run is done and RR-scheduling run, with a window of 0.1.

The results of these runs are presented in Table 5.2. The Min and Max in this table are the upper and lower bounds of the confidence window. In the simulation the utilization of block A is 0.723 (including time blocked, waiting for block B), and of block B the utilization is 0.636.

The PS provides a slightly higher response time for both tasks. With task 1 the confidence windows do overlap, but the averages are not within each others confidence windows. For the second task however, the averages do lie within each others confidence windows. So the RR-scheduling and processor sharing simulator give almost similar results for small window sizes.

The minor differences in both simulations are probably because both simulations are not performed with the same set of generated random numbers. Running the simulation multiple times would probably give a result where the windows do overlap.

Task number	Task response						Difference
	Full PS			Windowed PS = 0.1			
	Min	Avg	Max	Min	Avg	Max	%
1	11.983	11.994	12.005	11.999	12.008	12.018	0.12
2	15.485	15.501	15.517	15.478	15.505	15.531	0.02

Table 5.2: Case PS task responses

6 Validation

Using the discrete-event simulator, it is possible to validate the analytical results against the simulation results. A number of test cases are used to validate the analytical results. For all the simulations in this and the next chapter, the following settings have been used: The simulation is run until $t = 10^7$ and the values obtained before $t = 10^6$ are not used to calculate the average values, to suppress the start-up effects. The simulation is run ten times to compute a 90% confidence interval based on the average values of each run.

This chapter starts with the validation of the single block analysis in Section 6.1. Next, combinations of multiple blocks are discussed in Sections 6.2, 6.3 and 6.4.

6.1 Case 1: A single ASD block

In this case the modelling of a single ASD block is validated. This block receives both a synchronous and an asynchronous task. A schematic view is given in Figure 6.1. The parameters for the tasks are defined in Table 6.1. The expected service time for a synchronous call at block A is denoted as $E[S_s A]$. The expected the service time for an asynchronous call at block A is given by $E[S_{as} A]$. The think rate for the synchronous task is listed as z_s and the arrival rate for an asynchronous task is notated as λ_{as}.

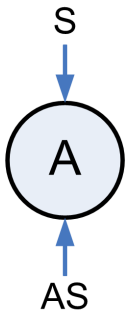


Figure 6.1: Single ASD block

Task number	Arrival / think rate	Expected service time
1	$z_s = 0.1$	$E[S_s A] = 3$
2	$\lambda_{as} = 0.4$	$E[S_{as} A] = 2$

Table 6.1: Case 1 task definition

In Figure 6.2 the simulation results of the task response times have been plotted with the confidence window acquired, using simulation. The lines indicate the analytical results and the confidence window is indicated using error bars. The lowest line indicates the response time for AS calls. The second line the response time for synchronous calls and the upper line indicates the utilization, for which the scale is on the second vertical axis.

The utilization plotted here, is the analysis result and the non-linearity is caused by the think time of the S calls. Due to this think time, the waiting time of the S calls is part of the inter-arrival time. So when the utilization increases, the waiting time for S calls increases, enlarging the inter-arrival time for S calls, "reducing" the utilization.

Since the simulation has a tight confidence window, the error bars plotted with the response times can hardly be distinguished in the plot, but all the results are well within the confidence window. To give a more detailed view, we zoom in on the response time of an asynchronous call for arrival rate of 0.2, in Figure 6.3.

In Table 6.2 task response times of the simulation and analysis are listed next to each other. The "min" and "max" listed with the simulation results are the upper and lower bound of the confidence window. The comparison has been performed for several values of λ_{as}. The results from Table 6.2 also show that the analysis results lie well within the confidence windows, as obtained by simulation. This also holds for high utilizations.

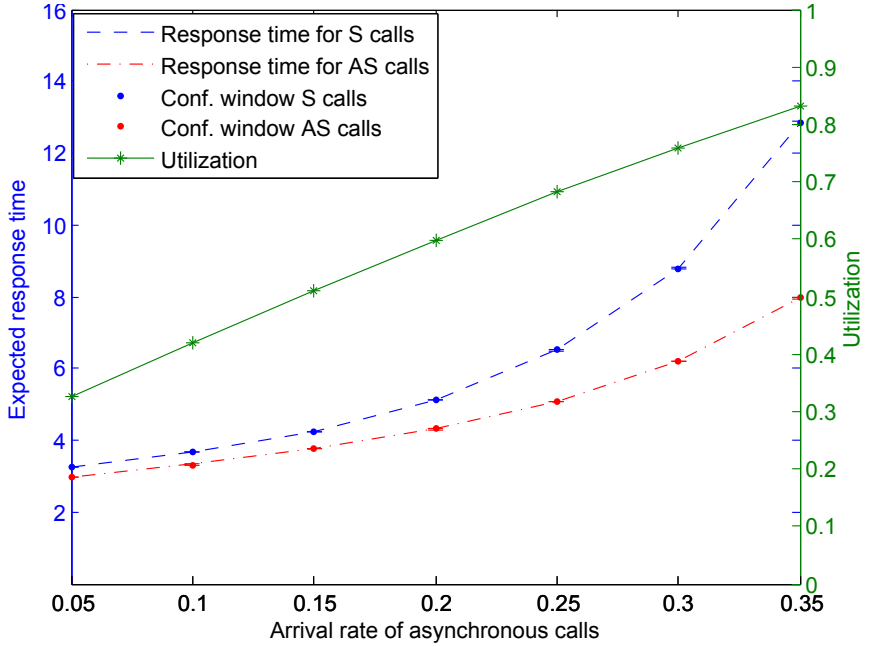


Figure 6.2: Waiting times against arrival rate

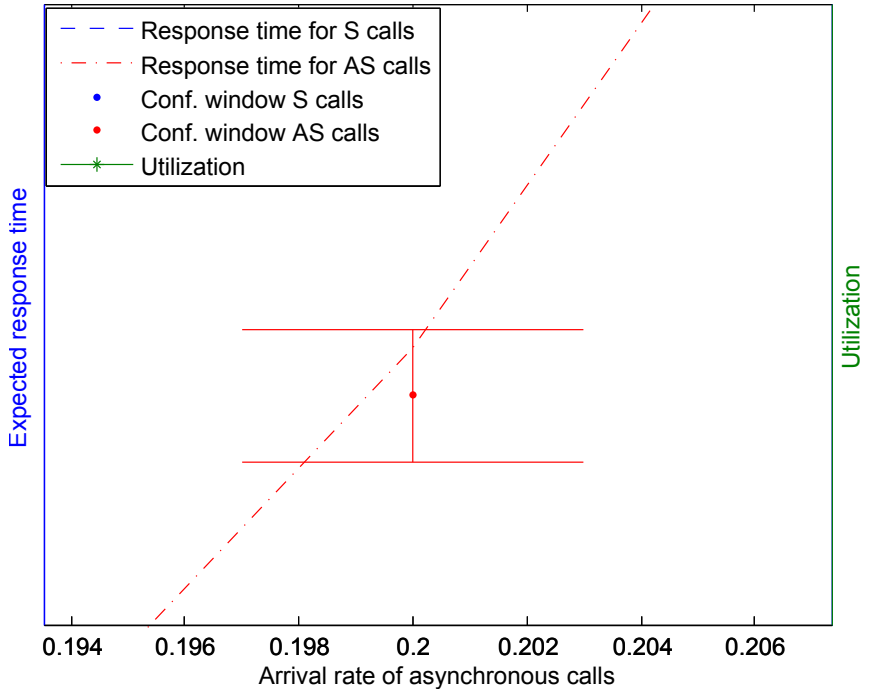


Figure 6.3: Zoom of Figure 6.2

Parameter	Simulation results				Analytical results	
	Response time AS		Response time S		Response time	
	min	max	min	max	AS	S
$\lambda_{as} = 0.1$	3.321	3.326	3.655	3.660	3.324	3.659
$\lambda_{as} = 0.4$	11.382	11.458	21.215	21.351	11.437	21.312
$\lambda_{as} = 0.45$	21.387	21.613	47.278	47.817	21.563	47.586

Table 6.2: Case 1 results

6.2 Multiple blocks

In the previous section only a situation with a single block has been analysed. Real systems consist at least of multiple blocks. To start with the analysis of multiple blocks, first two cases will be analysed, where only one task runs in the system.

In Figure 6.4 the task of first example is shown. This task has only synchronous calls, which have to be processed each on a different block. The expected service time at each block is listed in Table 6.3. In Figure 6.5 the task of the second example is shown. This task fully consists of asynchronous call, each to be executed on a different block. The corresponding expected service times are listed in Table 6.4.

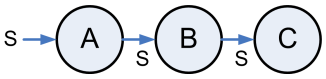


Figure 6.4: Task structure

Task number	Think Rate	Expected service time
1	$z_s = 0.1$	$E[S_{sA}] = 3$ $E[S_{sB}] = 2$ $E[S_{sC}] = 4$

Table 6.3: Multiple blocks task definition 1

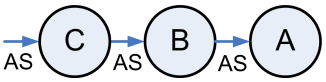


Figure 6.5: Task structure

Task number	Arrival rate	Expected service time
1	$\lambda_{as} = 0.1$	$E[S_{asA}] = 4$ $E[S_{asB}] = 2$ $E[S_{asC}] = 3$

Table 6.4: Multiple blocks task definition 2

Analysis of both examples is straightforward. For synchronous calls, the specification of ASD guarantees a maximum of one synchronous call to be at each block, so no waiting occurs. Therefore the response time of the task in example 1 is just the sum over all service times, which is 9. This is graphically shown in Figure 6.6. In this graph the analytical result is plotted as a line and the acquired confidence window with error bars. The utilization of block A (including blocking time) is also plotted, which is calculated with the analysis. As shown from the graph the analysis and simulation match on the response time. The utilization is strongly non-linear, because the utilization is not only dependant on the think rate (which is varied on the horizontal axis), but also on the response time (which is fixed).

The analysis with the asynchronous calls is done by analysing each block using the single block analysis and accumulating the response times of each block. This is possible, because the blocks do not have dependencies. The analysis results and simulation results are shown in Figure 6.7, in a similar way as the previous example. This graph also shows that the analysis and simulation match for the response time. With this graph clearly a linear relation between the arrival rate and utilization can be seen, because the utilization is only dependent on the arrival rate, which is varied on the horizontal axis. Also the analysis results for the response time lie within the confidence window obtained using simulation.

These two examples were rather simple, however when combinations of tasks have to be analysed, more complicated analysis is required. Combining multiple blocks does not influence the handling of asynchronous calls, because they are handled per block and do not influence the operation of other blocks. However, synchronous calls induce blocking, i.e, they do influence the operation of other blocks.

To illustrate the behaviour of a task in time, the situation of Figure 4.4 is shown here. In this figure a synchronous call is being processed at the grey block and its hierarchical parents are blocked, because they have issued a synchronous call, which is in progress. At the grey block also an asynchronous call is arriving.

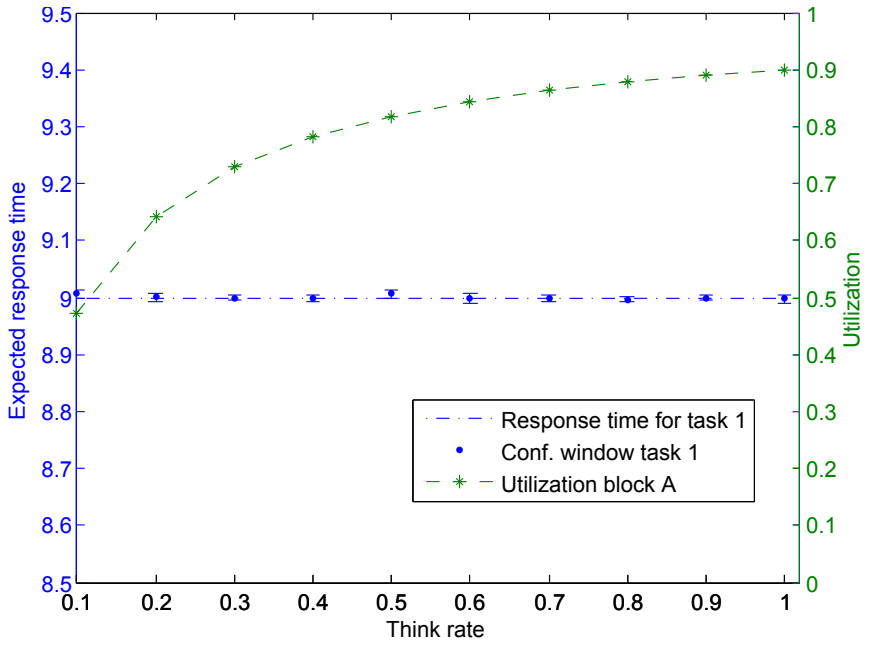


Figure 6.6: Only synchronous calls

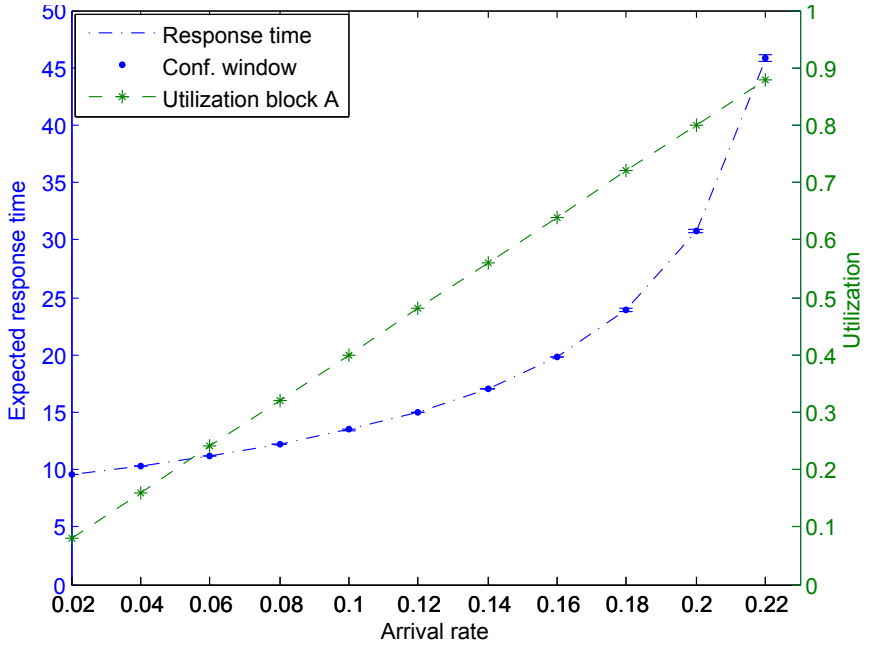


Figure 6.7: Only asynchronous calls

The flow of the task containing synchronous calls is illustrated in the time diagram of Figure 6.8, which shows the status of the three involved blocks at each step. The task first (step 1) arrives at block A. Then block A calls block B (step 2), block B starts processing and block A is blocked (step 3). The same holds for block C (step 4). When block C finishes, block B is processing again (step 5). After block B ended, it is free again and block A finishes the task (step 6).

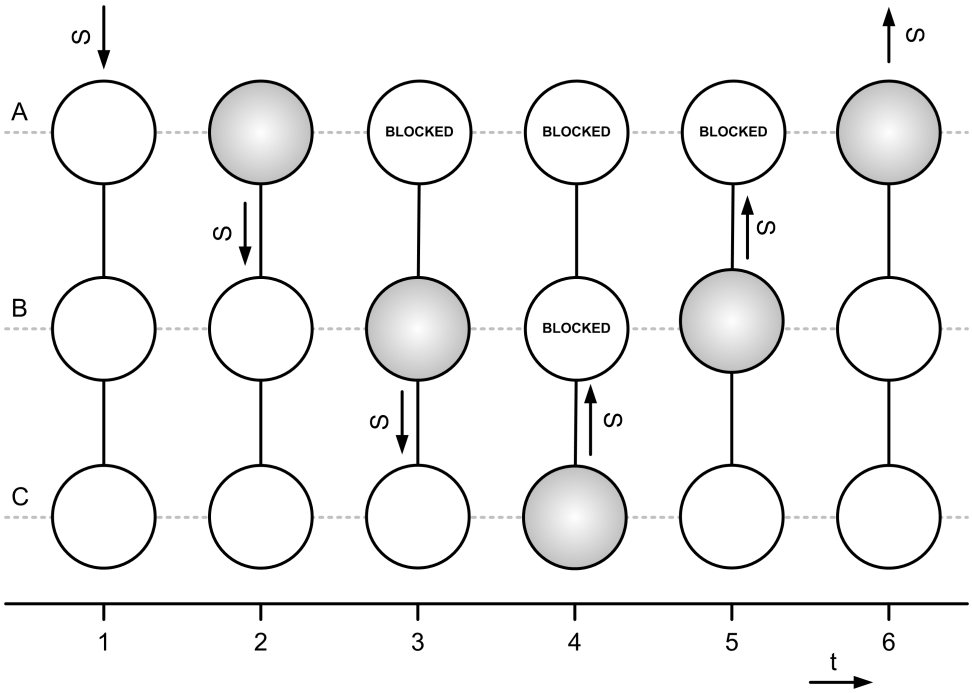


Figure 6.8: Time diagram of call flow

From a modelling perspective, the blocking means, that the time spend waiting for the called blocks, is actually part of the service time of the caller. Hence, the experienced service time of block B ($E[S^*_{sB}]$) is the time it actually does service by itself ($E[S_{sB}]$) plus the time the call waits at block C ($E[W_{sC}]$) plus the time it is served by block C ($E[S_{sC}]$). This is denoted in Equation 6.1.

$$E[S^*_{sB}] = E[S_{sB}] + E[W_{sC}] + E[S_{sC}] \tag{6.1}$$

To fully embed the influence of block C on block B, the CTMC of block C should be a part of the CTMC of block B. For two blocks this is already quite challenging to solve, because this results in a CTMC that is infinite in two dimensions. For even more blocks this becomes even more complex, so a different analysis method is required.

To keep the CTMC's small enough, we analyse them separately and propagate their influence to the influenced blocks. This methods keeps the state-space smaller, but is clearly is less accurate. At first only the expected values for waiting and service time will be propagated to the other blocks.

6.3 Case 2: Two calls, multiple blocks

Since the analysis matches the simulation for a single block, the results when combining multiple blocks can be reviewed. For this purpose a new case is defined as in Figure 6.9. The parameters for this case are denoted in Table 6.5. In this case a task enters the system with a synchronous call at block A and does a call to block B next. Another task enters the system as an asynchronous call at block A.

To calculate the response times, first block B is modelled as single block, then the service time at block A for the synchronous calls is adapted to include the response time of block B using

Equation 6.2. In this case the waiting time at block B is 0. Because block B only handles synchronous calls, there cannot be other calls in the block when a call arrives, resulting in a waiting time of 0. Therefore the experienced service time at block A is just the sum of both expected service times and becomes $E[S_sA^*] = 5$. Using the existing single block model, the service-time of block A and B and the waiting time at block B are together modelled as a single negative exponential distribution with an expected value of 5, although this combination is not negative exponential distributed.

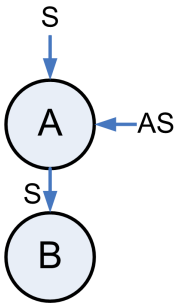


Figure 6.9: Structure case 2

$$E[S^*_{sA}] = E[S_{sA}] + E[W_{sB}] + E[S_{sB}] \tag{6.2}$$

Task number	Arrival / Think Rate	Expected service time
1	$z_s = 0.1$	$E[S_sA] = 3$ $E[S_sB] = 2$
2	$\lambda_{as} = 0.4$	$E[S_{as}A] = 2$

Table 6.5: Case 2 task definition

In Figure 6.10 the analytic task response times are plotted with lines and the confidence window of the simulation results with the error bars. From this graph it is clear that the analysis results are not within the confidence window of the simulation. In Table 6.6 also both the analysis and simulation results for the response times of the tasks of case 2 are displayed. The columns indicated with “min” and “max” represent the lower and upper bounds of the confidence interval. Looking at these measures, it clearly shows that the analysis over estimates the response times of both tasks and the results do not lie within the confidence interval obtained by simulation. This is only a simple example, enlarging the case could lead to even larger differences. So a further analysis is required to find the source of the problem.

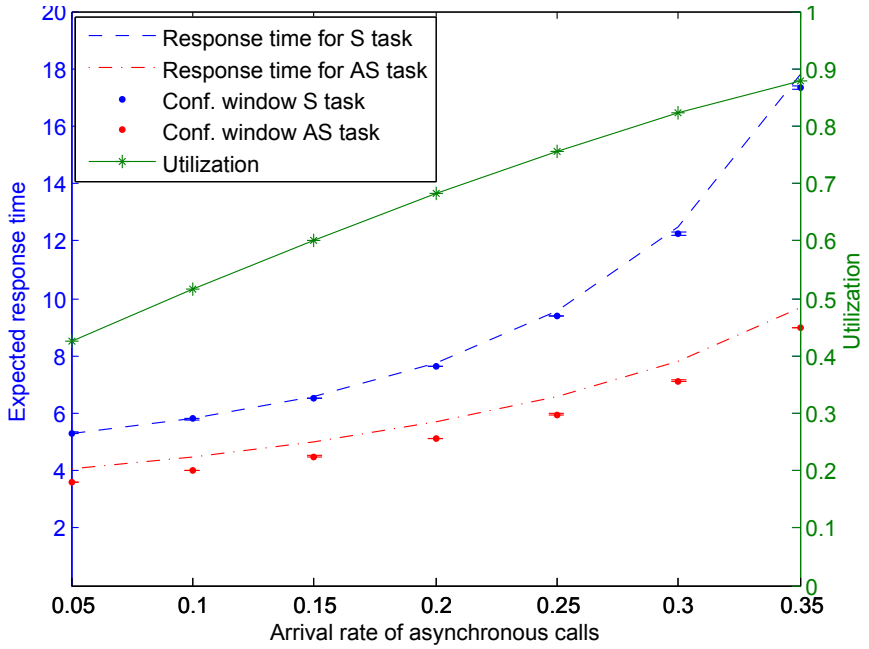


Figure 6.10: Task response times for case 2

Parameter	Simulation results				Analytical results	
	Response time AS		Response time S		Response time	
	min	max	min	max	AS	S
$\lambda_{as} = 0.1$	4.000	4.008	5.787	5.796	4.477	5.8108
$\lambda_{as} = 0.4$	12.456	12.509	28.001	28.130	13.220	28.8197
$\lambda_{as} = 0.45$	22.649	22.838	60.991	61.488	23.431	62.857

Table 6.6: Case 2 results

To provide more insight into the problem, a number of internal values retrieved from the simulation are listed in Table 6.7 with the situation of $\lambda_{as} = 0.4$. To keep overview confidence intervals are not given here. From these values it can be seen that especially the expected number of asynchronous call at the arrival of a synchronous call ($E[N_{qA}]$) deviates a lot, which is 5.5% in this situation. This deviation is caused by the incorrect assumption in the analytical model, that the experienced service time of block A is negative exponentially distributed. In fact the experienced service time of block A models two connected blocks, so the experienced service time of the sychronous calls at block A is hypo-exponentially distributed (convolution of two negative exponential distributions). This hypo-exponential distributions has two phases, the first phase is the service at block A, the second phase is the service time of block B.

Parameter	Simulation results	Analytical results
$E[IA]forS$	38.168	38.75
$E[N_{qA}]@A$	1.571	1.658

Table 6.7: Case 2 results

To incorporate this hypo-exponential distribution, the CTMC model of block A has to be adapted. Also the first and second moments of the service times have to be calculated differently. This means that the Equation 4.2 used for calculation of the remaining time of a call in service is not true anymore, because this equation is also based on negative exponentially distributed service times.

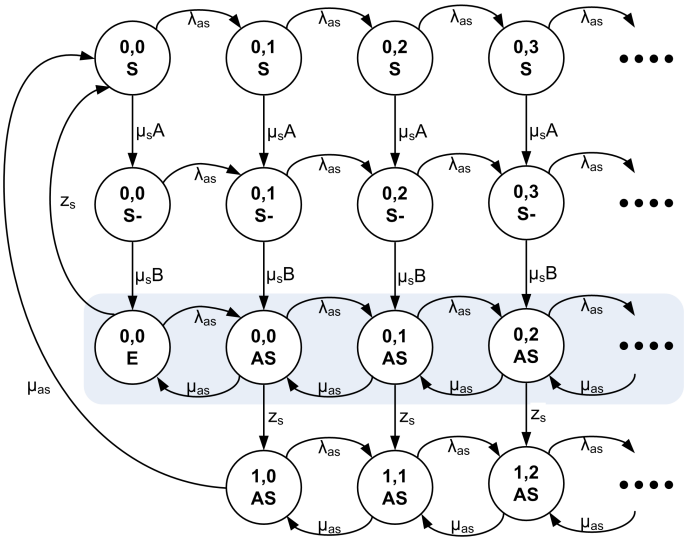


Figure 6.11: Adapted CTMC for block A of case 2

The adapted CTMC for block A is displayed in Figure 6.11, where a complete new row of states is introduced, with respect to the standard CTMC of an ASD block as displayed in Figure 4.7. This new CTMC, explicitly models the two phases of the service for the synchronous call. Also

all the matrices for the matrix geometric solution have to be derived again, which has not been displayed here. The states used to calculate the r_{as} (the probability of an arriving synchronous call, finding the server busy) and the $E[N_{qA}]$ (the expected queue length at the arrival of a synchronous call) do not change and are indicated with the rectangle on the background in the CTMC in Figure 6.11.

To calculate the second moment of the service time of the synchronous call for block A Equation 6.3 has to be used. This is the applied version of the second moment of the hypo-exponential distribution. A proof for this equation is given in Appendix B.

$$E[S_s^2] = \frac{2}{\mu_A^2} + \frac{2}{\mu_B^2} + \frac{2}{\mu_A \cdot \mu_B} \tag{6.3}$$

After applying this new CTMC together with the adapted second moment, the analytic task response times as displayed in Figure 6.12 (with a think rate z_s is 0.1) and Table 6.8 lie within the confidence interval retrieved from the simulation. The simulation values are the same as in Table 6.6. Figure 6.12 shows that the analytic and simulation results are close together and Table 6.6 confirms that the analytic results are within the confidence window obtained by simulation.

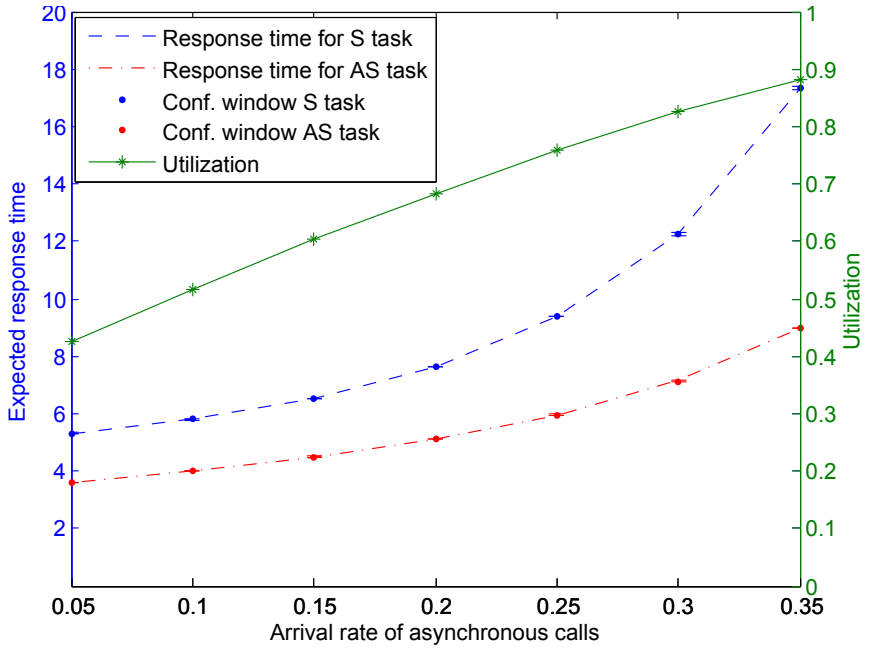


Figure 6.12: Case 2: Task responses using hypo-exponential service time

Parameter	Simulation results				Analytical results	
	Response time AS		Response time S		Response time	
	min	max	min	max	AS	S
$\lambda_{as} = 0.1$	4.000	4.008	5.787	5.796	4.004	5.791
$\lambda_{as} = 0.4$	12.456	12.509	28.001	28.130	12.495	28.078
$\lambda_{as} = 0.45$	22.649	22.838	60.991	61.488	22.668	61.220
$\lambda_{as} = 0.1$ $z_s = 10$	6.248	6.256	6.230	6.234	6.252	6.231

Table 6.8: Case 2 results adapted algorithm

6.4 Case 3: Two calls, multiple blocks, oriented differently

Since case 2 can be modelled and analysed well, lets look at a slightly different case as presented in Figure 6.13 and Table 6.9. The only difference with case 2 is that the asynchronous task arrive in this case at block B instead of block A. In this case, a task starts with a the synchronous call at block A, after it is served, it issues a synchronous call at block B. Hence, the think time for the synchronous calls at block B is no longer negative exponentially distributed, but is hypo-exponentially distributed.

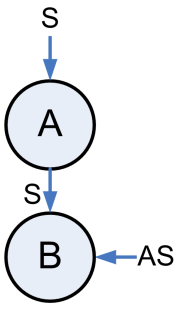


Figure 6.13: Structure case 3

Task number	Arrival / Think rate	Expected service time
1	$z_s = 0.1$	$E[S_{sA}] = 3$ $E[S_{sB}] = 2$
2	$\lambda_{as} = 0.4$	$E[S_{asB}] = 2$

Table 6.9: Case 3 task definition

In case 2 block B also had an hypo-exponentially distributed think time, but because block B did not have any waiting time, no error in the response time of the tasks was made by modelling this think time incorrectly. When the think time is modelled as negative exponential, the experienced think time ($E[Z^*_{sB}]$) for block B can be calculated using Equation 6.4, which accumulates, the task think time $E[Z_{sT}]$, the waiting time for synchronous calls at block A and the service time of the synchronous call at block A.

$$E[Z^*_{sB}] = E[Z_{sT}] + E[W_{sA}] + E[S_{sA}] \tag{6.4}$$

In Table 6.10 the resulting task responses are listed when modelling all parts as negative exponentially distributed. The "min" and "max" columns in this table indicate the lower and upper bound of the confidence interval. These results have also been plotted with think rate (z_s) of 0.1 in Figure 6.14. As visible in the table, the analytical results lie only slightly out of the confidence window when modelling the think rate as negative exponential, so this is also barely visible in Figure 6.14. A rather extreme case ($\lambda_{as} = 0.1$ and $z_s = 10$) is taken to show that in certain conditions both response times fall out of the confidence window.

Parameter	Simulation results				Analytical results	
	Response time AS		Response time S		Response time	
	min	max	min	max	AS	S
$\lambda_{as} = 0.1$	2.817	2.822	5.604	5.611	2.821	5.598
$\lambda_{as} = 0.4$	10.590	10.633	22.789	22.870	10.620	22.236
$\lambda_{as} = 0.45$	20.573	20.834	48.740	49.145	20.694	47.600
$\lambda_{as} = 0.1$ $z_s = 10$	3.380	3.385	5.559	5.562	3.261	5.568

Table 6.10: Case 3 results

To get the analytical results to match the simulation, the hypo-exponential distributed think time has to be incorporated in the model. The new CTMC (in which this hypo-exponential think time is included) is displayed in Figure 6.15. In this CTMC the states with the "+" symbol indicate that no synchronous call is in block B, but there is one in block A. These states have to be used to calculate the r_{as} and the $E[NqA]$.

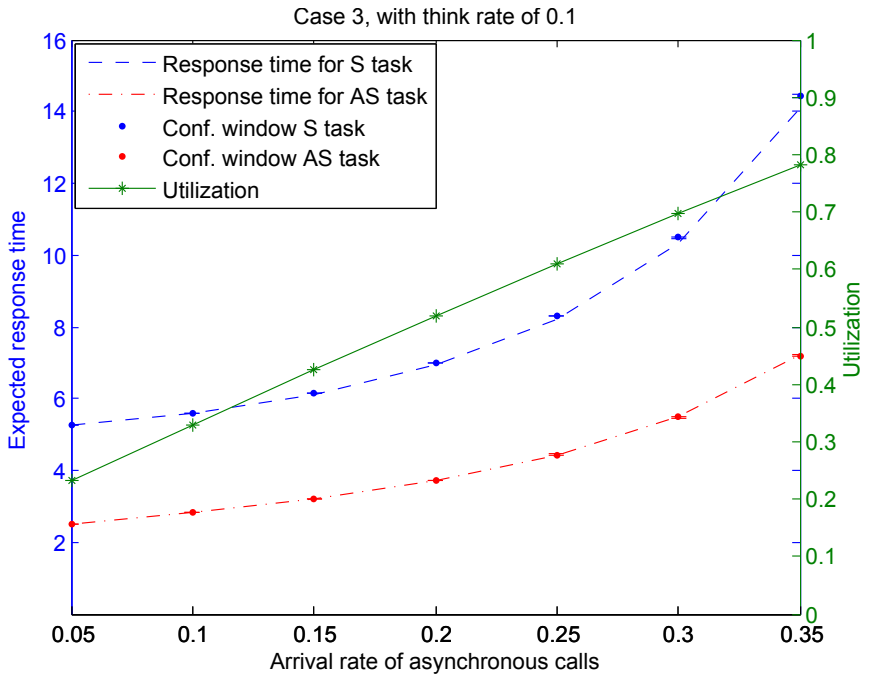


Figure 6.14: Case 3: Task response

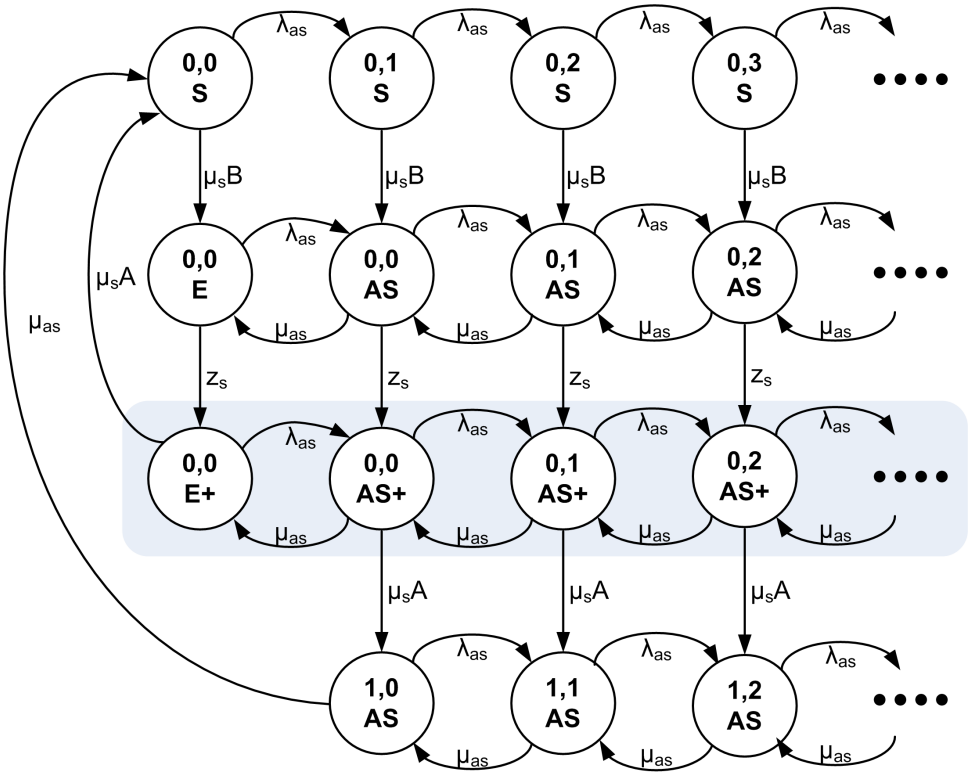


Figure 6.15: Adapted CTMC for block B of case 3

Using the CTMC from Figure 6.15 for the analysis, the new analytical results together with the same simulation results as shown in Table 6.11. These results are also plotted in the plot in Figure 6.16. From the graph it is clear that with the hypo-exponential think time, the simulation and analysis results for the response times are close together and the table shows that the analytic results lie within the confidence window.

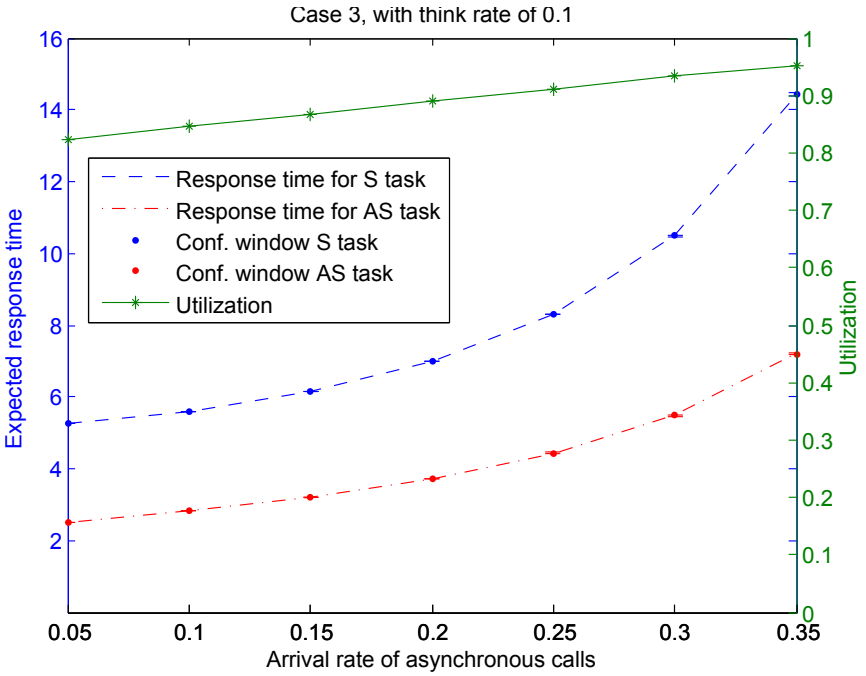


Figure 6.16: Case 3: Task response with hypo-exponential think times

Parameter	Simulation results				Analytical results	
	Response time AS		Response time S		Response time	
	min	max	min	max	AS	S
$\lambda_{as} = 0.1$	2.817	2.822	5.604	5.611	2.820	5.607
$\lambda_{as} = 0.4$	10.590	10.633	22.789	22.870	10.610	22.816
$\lambda_{as} = 0.45$	20.573	20.834	48.740	49.145	20.679	48.913
$\lambda_{as} = 0.1$ $z_s = 10$	3.380	3.385	5.559	5.562	3.383	5.561

Table 6.11: Case 3 results adapted algorithm

7 Generalizing analysis of multiple ASD blocks

A method to manually construct the analysis with multiple blocks has been presented in Chapter 6. For practical use, this process has to be automated. This automated analysis method should construct the complete model from a list of tasks and the block architecture. This chapter presents this automated approach.

The chapter starts with a global overview of the developed waiting time propagation algorithm to model multiple block in Section 7.1, next Section 7.2 presents a method to acquire the dependencies between blocks. In Section 7.3 the changes to the analysis methods of Chapter 6, required to analyse systems with multiple blocks. In Sections 7.4 and 7.5 the automated algorithm is applied to cases and compared with simulation results. Section 7.6 states a method to determine the required number of iterations for the waiting time propagation. Finally a summary of all the cases and algorithms presented is listed in Section 7.7.

7.1 Waiting time propagation: an iterative approach

The analysis should output the expected response time of all the tasks. To derive these values the following steps have to be taken:

1. Investigate per block which tasks have to be processed. With this accounting action for each block two lists are made, one for all synchronous tasks and one for all asynchronous tasks using the block.

   Repeat until the changes in waiting times are small enough (Section 7.6)

   (a) Build the phase-type distributions for the service time of synchronous calls and the think time for synchronous calls. More information on this process is given in Section 7.2.

   (b) (Re)compute the waiting times per block based on the determined phase-type distributions. This process is explained in Section 7.3 .

   Until

2. Add the waiting and service times of the calls to calculate the expected response time of a task.

7.2 Building cross block distributions

Before building phase-type distributions including influences between blocks, these influences are discussed more in detail. This will be explained according to an example as shown in Figure 7.1. In this example block A sends synchronous calls to block B, which is also handling asynchronous calls. Due to the presence of asynchronous call, the synchronous calls may have to wait at block B.

Since the asynchronous calls do not induce blocking, they do not influence the waiting time at other blocks directly. Blocking as induced by synchronous calls is studied in more detail. The flow of the task containing the synchronous calls in the example, is displayed in Figure 7.2. The task starts at block A, then goes to block B. After it finishes at block B, it returns a call to block A, but does not need any more processing at block A. Hence, directly after block B finishes, the think stage of this task starts. After the think stage, the new task arrives at block A.

During the entire time of the task being executed at block A and B, block A cannot perform other activities. Firstly it is processing a call, secondly it is blocked while block B is processing. Therefore the experienced service time of block A consists of the service time at block A, the

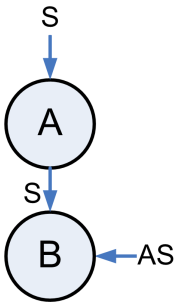


Figure 7.1: Example of two influencing blocks

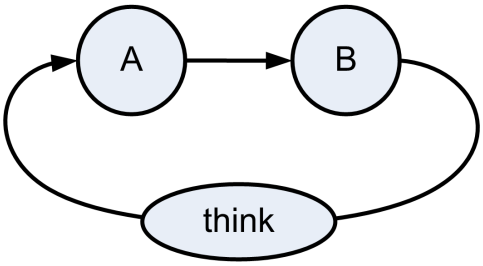
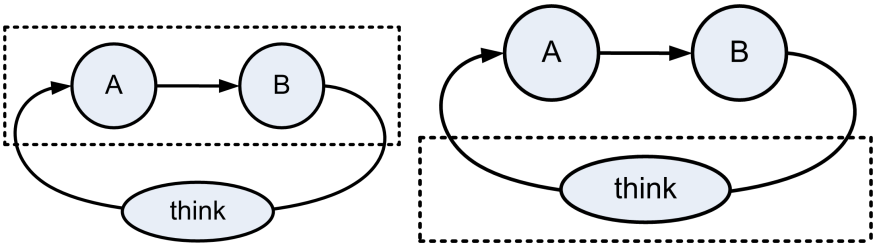


Figure 7.2: Synchronous cycle

waiting time at block B and the service time at block B. Projected on the entire task cycle, the experienced service time is shown by the dashed rectangle in Figure 7.3(a). The other part of the cycle will be the think time as shown in Figure 7.3(b).



(a) Service time block A (b) Think time block A

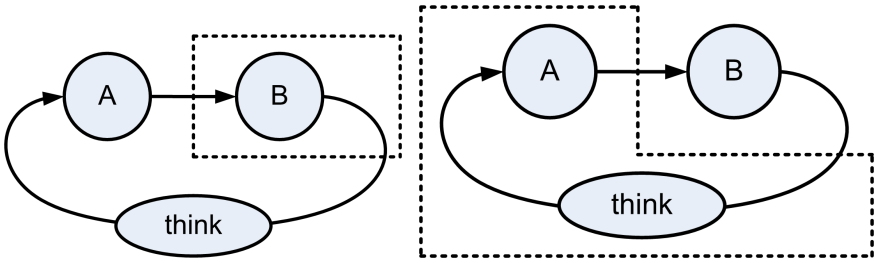
Figure 7.3: Contribution of the parts of the synchronous task to block A

This means that the experienced service time at block A is given by Equation 7.1, which adds the service times for the synchronous calls of both blocks and the waiting time for synchronous calls at block B.

$$E[S^*_{sA}] = E[S_{sA}] + E[W_{sB}] + E[S_{sB}] \tag{7.1}$$

The situation around block A is detailed out, lets look at the situation at block B. Block B has to process both synchronous and asynchronous calls. The asynchronous calls are arriving with a known inter-arrival distribution, which does not depend on any other distribution. However, the synchronous calls only arrive after block A has processed them. After arrival, the synchronous call is processed at block B and then processing has finished. In the perspective of think and service time, the service time at block B is just the processing it does by itself as shown by the dashed rectangle in Figure 7.4(a). The think time at block B includes the think time of the task and the processing at block A, as shown in Figure 7.4(b).

So at block B, the think rate is no longer negative exponential distributed, because it consists of



(a) Service time block B (b) Think time block B

Figure 7.4: Contribution of the parts of the synchronous task to block B

two phases. One phase is the think time of the task, with rate z_{sT}, the second the service time at block A, with rate μ_{sA}.

This means that the experienced value for the think time at block B ($E[Z^*_{sB}]$) is given by Equation 7.2. This equation which accumulates the task think time, expected waiting time for synchronous calls at block A and the service of this task at block A.

$$E[Z^*_{sB}] = E[Z_{sT}] + E[W_{sA}] + E[S_{sA}] \tag{7.2}$$

In general Equations 7.2 and 7.1 cannot be solved directly, because they influence each other by the waiting time. The think rate at block B depends on the waiting time of block A, which on its turn depends on the waiting time of block B, which depends on the think rate at block B. Using iteration a good approximation for the waiting times is computed.

In this case $E[W_sA] = 0$, because at block A only synchronous calls are processed, so iteration is not needed in this situation.

When looking at Figures 7.4 and 7.3, a general modelling rule can be determined for tasks with blocking calls:
All the parts handled while not blocked (except for the waiting time at this block) contribute to the experienced think rate z^*_s. All the parts during the blocked time contribute to the experienced service time S^*_s.

In the previous section the combined behaviour of multiple blocks is analysed. To perform adequate analysis, also the distribution of the influences has to be taken into account. Because all the basic transitions are negative exponentially distributed, combined transitions will become phase type.

The experienced synchronous service of block A (S^*_{sA}) illustrates the application of phase-type distributions. This experienced service time consists of tree parts:
- The actual service time of the synchronous call at block A
- The waiting time for synchronous calls at block B
- The service time at of the synchronous call at block B

When a phase-type distribution for all of these elements is known, the combined distribution of the experienced synchronous service at block A can be constructed using convolution.

First the phase-type distributions for each of the parts mentioned above, are needed. The service times for synchronous calls of both block A and B are assumed to be negative exponentially distributed, resulting in phase-type distributions, as given by Equations 7.3 and 7.4. The matrix **T** has only one element, meaning there is only one phase. Because the value of the matrix **T** represents the rate at which the distribution ends, in this situation $-\mu_{sA}$ or $-\mu_{sB}$. The vector $\vec{a}$ indicates the initial distribution of the phase-type distribution. Because it is 1, it means the it always starts with the phase represented in matrix **T**.

$$\mathbf{T}_{\mu sA} = \left[-\mu_{sA}\right] \text{ and } \vec{a}_{\mu sA} = 1 \tag{7.3}$$

$$\mathbf{T}_{\mu sB} = \left[-\mu_{sB}\right] \text{ and } \vec{a}_{\mu sB} = 1 \tag{7.4}$$

The phase distribution of the waiting time for block B is much harder to derive. This distribution is approximated using the moment matching algorithm and using the first, second and third moments of the waiting time. Recall, that a detailed description of the moment matching algorithm is given Section 3.13.

Using convolution for phase-type distributions (Section 3.12.1) a combined phase-type distribution is acquired. This combined phase-type distribution, represented by $\mathbf{T}^*_{\mu sA}$ and $\vec{a}^*_{\mu sA}$, approximates the experienced service time distribution.

The examples of Sections 6.3 and 6.4 only considered one synchronous task, but in reality, multiple tasks can be using the same block. To derive the experienced think times and service times the following steps have to be carried out for each block:

1. Collect all tasks with synchronous calls at this block.
2. Determine for each call, what part of the task contributes to the think rate and what part to the service time is and derive phase type distributions for them.
3. Combine the distributions of multiple tasks using the mixture operation for phase type transitions, with the expected inter arrival time as measure.

For asynchronous calls, a similar strategy is used. The arrival rates of all tasks are modelled by a single negative exponential distribution with as rate the sum over all arrival rates. The distribution for the asynchronous service time is derived as a mixture of all negative exponential transitions, with the arrival rates as weights.

7.3 Adapting modelling and analysis methods

The analysis methods in Chapter 4 are all based on negative exponentially distributed transitions. These transitions have to be replaced by phase-type transitions and the analysis method has to be adapted, to cope with the analysis of multiple blocks.

So every transition in the CTMC as shown in Figure 4.7 must be replaced by a phase-type transition. As shown in Section 3.12, a phase-type transition is described by the matrix $\mathbf{T}$ and vector $\vec{a}$. Because a phase-type transition itself contains a number of states, when replacing negative exponential transitions by phase type transitions, the state space grows.

Firstly, two simple examples are shown on how to derive the transition matrices based on phase-type distributions. In Figure 7.5 a simple CTMC is shown. When the transitions with rates z_s and μ_s are replaced by phase-type transitions, the corresponding generator matrix $\mathbf{Q}$ is shown in Equation 7.5.

In the left upper section of the matrix the sub-matrix $\mathbf{T}_{\mu s}$ is located, which describes the transitions within the states of the phase-type distribution. The process of going to the terminating state of this phase-type transition is described by vector $\vec{T^0}_{\mu s}$ (upper right part). To initiate the second phase-type transition, this vector is multiplied by the initiating vector $\vec{a}_{zs}$. Now the sub-matrix $\mathbf{T}_{zs}$ (lower right part) describes the movements within this distribution. Leaving of this phase-type distribution is described by vector $\vec{T^0}_{zs}$ (lower left part), which is multiplied by the $\vec{a}_{\mu s}$ to initiate the μ_s phase again. This completes the generator matrix, which is now of size $(n_{zs} + n_{\mu s})$ by $(n_{zs} + n_{\mu s})$.

This generator matrix $\mathbf{Q}$ looks very similar to the convolution rules for phase-type transitions (Section 3.12.1), only in this situation $\sum \vec{a} = 1$ has to hold (no positive probability of having a

duration of 0). If this would not hold, matrix **Q** should be different and would become much more complex.

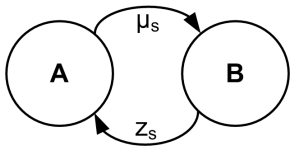


Figure 7.5: CTMC example 1

$$\mathbf{Q} = \begin{bmatrix} \mathbf{T}_{\mu s} & \vec{T^0}_{\mu s}\,\vec{a}_{zs} \\ \vec{T^0}_{zs}\,\vec{a}_{\mu s} & \mathbf{T}_{zs} \end{bmatrix} \tag{7.5}$$

In this first example, from each state only one transition is possible. In the second example there are multiple transitions possible from each state, as shown in Figure 7.6. The corresponding generator matrix **Q** is given by Equation 7.6. In this matrix the ordering of rows and columns corresponds to the state set numbering in the CTMC (A, B, C and D).

The left upper sub-matrix describes the transitions within A. This sub-matrix is build from the **T** matrices of both outgoing transitions. The tensor sum makes that both phases get an independent state dimension, which is needed, because they are independent. More information on tensor sums and products is given in Appendix A. This means that this term becomes a $(n_{\mu s} \cdot n_{\lambda as}) \times (n_{\mu s} \cdot n_{\lambda as})$ matrix, because for each state of one phase, the complete state space of the other phase involved is needed. This independence makes that the state space grows fast when the number of phases increases.

The second sub-matrix in the row models going from A to B. This sub-matrix terminates the phase-type distribution, replacing the transition with rate μ_s (vector $\vec{T^0}_{\mu s}$) and initiates the phase-type distribution replacing transition μ_s (vector $\vec{a}_{zs}$), but does not influence the distribution, replacing the transition with rate λ_{as} (sub-matrix $\mathbf{I}_{\lambda as}$). The $\mathbf{I}_{\lambda as}$ matrix is an identity matrix with the same size as the $\mathbf{T}_{\lambda as}$ matrix. The order of the tensor product should be used consistently throughout the complete matrix, otherwise the phases of different transitions will be mixed up. The third sub-matrix in the row terminates the distribution, replacing the transition with rate λ_{as}, initiates the distribution, replacing the transition with rate μ_{as}, and does not influence the distribution, replacing the transition with rate μ_s. The fourth sub-matrix in the row is zero because there is no possibility of moving from state set A to D. The next rows of the generator matrix are constructed in a similar way. The total matrix width and height is then $(n_{\mu s} + n_{zs}) \cdot (n_{\lambda as} + n_{\mu as})$. Note that, only the diagonal elements in the matrix have to be square matrices, the other elements may be rectangular.

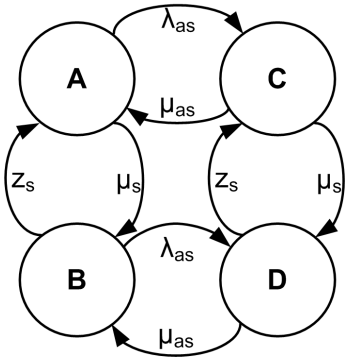


Figure 7.6: CTMC example 2

$$\mathbf{Q} = \begin{bmatrix} \mathbf{T}_{\mu s} \oplus \mathbf{T}_{\lambda as} & \vec{T}^0{}_{\mu s}\, \vec{a}_{zs} \otimes \mathbf{I}_{\lambda as} & \mathbf{I}_{\mu s} \otimes \vec{T}^0{}_{\lambda as}\, \vec{a}_{\mu as} & \mathbf{0} \\ \vec{T}^0{}_{zs}\, \vec{a}_{\mu s} \otimes \mathbf{I}_{\lambda as} & \mathbf{T}_{zs} \oplus \mathbf{T}_{\lambda as} & \mathbf{0} & \mathbf{I}_{zs} \otimes \vec{T}^0{}_{\lambda s}\, \vec{a}_{\mu as} \\ \mathbf{I}_{\mu s} \otimes \vec{T}^0{}_{\mu as}\, \vec{a}_{\lambda as} & \mathbf{0} & \mathbf{T}_{\mu s} \oplus \mathbf{T}_{\mu as} & \vec{T}^0{}_{\mu s}\, \vec{a}_{zs} \otimes \mathbf{I}_{\mu as} \\ \mathbf{0} & \mathbf{I}_{zs} \otimes \vec{T}^0{}_{\mu as}\, \vec{a}_{\lambda as} & \mathbf{0} & \mathbf{T}_{zs} \oplus \mathbf{T}_{\mu as} \end{bmatrix} \tag{7.6}$$

7.3.1 Extending the single block analysis with phase-type transitions

To replace the negative exponential transitions of the single block analysis with phase-type distributions, the matrices $\mathbf{B_{00}}$, $\mathbf{B_{01}}$, $\mathbf{B_{10}}$, $\mathbf{A_0}$, $\mathbf{A_1}$ and $\mathbf{A_2}$ have to be adapted. The new matrices have the same state set ordering as the ones with negative exponential distributions as shown in Chapter 4.

On the diagonal of the $\mathbf{B_{00}}$ and $\mathbf{A_1}$ matrix all the possible transitions in that state set are represented by a $\mathbf{T}$ matrix in the form of a tensor sum, so the operating phases are independent. The states where the transitions go, need to have a $\vec{T}^0$ vector to exit the phase. When a new phase is started in that state set, a $\vec{a}$ vector initiates that phase. Phases that are not affected by the phase ending are represented with an identity matrix $\mathbf{I}$ of the right size. In some cases, a column times row vector multiplication is used, which has the same effect as a tensor product with those vectors.

The new matrices, that describe the quasi birth death process of the ASD block with phase-type distributions, are listed by Equations 7.7 till 7.12.

$$\mathbf{B_{00}} = \begin{bmatrix} \mathbf{T}_{\mu s} & \vec{T}^0{}_{\mu s}\, \vec{a}_{zs} \\ \vec{T}^0{}_{zs}\, \vec{a}_{\mu s} & \mathbf{T}_{zs} \end{bmatrix} \oplus \mathbf{T}_{\lambda as} \tag{7.7}$$

$$\mathbf{B_{01}} = \begin{bmatrix} \mathbf{I}_{\mu s} \otimes \vec{T}^0{}_{\lambda as}\, \vec{a}_{\lambda as} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{zs} \otimes \vec{T}^0{}_{\lambda as}\, \vec{a}_{\lambda as} \otimes \vec{a}_{\mu as} & \mathbf{0} \end{bmatrix} \tag{7.8}$$

$$\mathbf{B_{10}} = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{zs} \otimes \mathbf{I}_{\lambda as} \otimes \vec{T}^0{}_{\mu as} \\ \mathbf{I}_{\lambda as} \otimes \vec{T}^0{}_{\mu as}\, \vec{a}_{\mu s} & \mathbf{0} \end{bmatrix} \tag{7.9}$$

$$\mathbf{A_0} = \begin{bmatrix} \mathbf{I}_{\mu s} \otimes \vec{T}^0{}_{\lambda as}\, \vec{a}_{\lambda as} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{zs} \otimes \vec{T}^0{}_{\lambda as}\, \vec{a}_{\lambda as} \otimes \mathbf{I}_{\mu as} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \vec{T}^0{}_{\lambda as}\, \vec{a}_{\lambda as} \otimes \mathbf{I}_{\mu as} \end{bmatrix} \tag{7.10}$$

$$\mathbf{A_1} = \begin{bmatrix} \mathbf{T}_{\mu s} \oplus \mathbf{T}_{\lambda as} & \vec{T}^0{}_{\mu s}\, \vec{a}_{zs} \otimes \mathbf{I}_{\lambda as} \otimes \vec{a}_{\mu as} & \mathbf{0} \\ \mathbf{0} & (\mathbf{T}_{zs} \oplus \mathbf{T}_{\lambda as}) \oplus \mathbf{T}_{\mu as} & \vec{T}^0{}_{zs} \otimes \mathbf{I}_{\lambda as} \otimes \mathbf{I}_{\mu as} \\ \mathbf{0} & \mathbf{0} & \mathbf{T}_{\lambda as} \oplus \mathbf{T}_{\mu as} \end{bmatrix} \tag{7.11}$$

$$\mathbf{A_2} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{zs} \otimes \mathbf{I}_{\lambda as} \otimes (\vec{T}^0{}_{\mu as}\, \vec{a}_{\mu as}) & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I}_{\lambda as} \otimes (\vec{T}^0{}_{\mu as}\, \vec{a}_{\mu as}) \end{bmatrix} \tag{7.12}$$

For the analysis, the state probabilities of the asynchronous queue upon the arrival of a synchronous call are required. To select these probabilities from the steady-state sub-vectors, a selection matrix is required. These selection matrices are build using a normalize $\vec{T}^0{}_{zs}$ vector and the identity matrices of the other phase-type distributions, to get the situation at arrival, but not influence the other distribution. The selection matrix for $\vec{z}_0$ becomes $\mathbf{V_0}$ as in Equation 7.13, the selection matrix for $\vec{z}_1$ becomes $\mathbf{V_1}$ as in Equation 7.14.

By applying $\vec{z}_0 \cdot \mathbf{V_0}$ (steady-state probability sub-vector times the selection matrix) the steady-state probability sub-vector of the empty asynchronous queue, upon arrival of a synchronous

call, is calculated. The distributions of the replaced transitions λ_{as} and μ_{as} are phase-type, so the resulting steady state vector may consist of multiple elements. By using $\vec{z}_i \cdot \mathbf{V_1}$ the steady-state probability sub-vector when having i calls in the asynchronous queue, upon the arrival of a synchronous call, is calculated.

$$\mathbf{V_0} = \begin{bmatrix} \vec{T^0}_{\mu s} \cdot 0 \\ \frac{\vec{T^0}_{zs}}{\sum \vec{T^0}_{zs}} \end{bmatrix} \otimes (\mathbf{I}_{\lambda as} \cdot \vec{1}) \tag{7.13}$$

$$\mathbf{V_1} = \begin{bmatrix} \vec{T^0}_{\mu s} \cdot 0 \\ \left(\frac{\vec{T^0}_{zs}}{\sum \vec{T^0}_{zs}}\right) \otimes (\mathbf{I}_{\mu as} \cdot \vec{1}) \\ \mathbf{I}_{\mu as} \cdot \vec{0} \end{bmatrix} \otimes (\mathbf{I}_{\lambda as} \cdot \vec{1}) \tag{7.14}$$

7.3.2 Analysis

Although the distributions have become phase-type, using the selection matrices, the analysis as presented in Chapter 4 can still be used.

Using this analysis only systems where tasks have at most one block in common, can be modelled adequately. When the same tasks have multiple blocks in common, they do not only influence their service times, but also their arrivals become dependant. This dependence is not incorporated in this model and not modelling this dependence results in severe errors between analysis and simulation.

7.4 Case 4: Applying Moment matching

To validate the functioning of the analysis algorithm with the moment matching, a new case has been defined as presented in Figure 7.7. In this case a task consisting of two synchronous calls (Task 1) has first a call at block A, next a call at block B. Besides this task, two other tasks are using the blocks. Task 2 consists of a single asynchronous call, that has to be executed at block B, task 3 also consists of a single asynchronous call, which has to be processed on block A. In Table 7.1 the expected service times, think rates and arrival rates for each task are defined.

Task 1, consists of multiple synchronous calls, both blocks the calls have to be processed on, have asynchronous calls arriving, which induces waiting time for the synchronous calls. So in the analysis of a single block, the waiting time at the other block has to be taken into account, therefore moment matching has to be applied in this case, to model the influence of waiting time at other blocks.

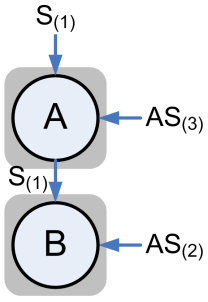


Figure 7.7: Structure case 4

Task number	Arrival / Think Rate	Expected service time
1	$z_s = 0.1$	$E[S_sA] = 3$ $E[S_sB] = 2$
2	$\lambda_{as} = 0.4$	$E[S_{as}B] = 2$
3	$\lambda_{as} = 0.3$	$E[S_{as}A] = 1$

Table 7.1: Case 4 task definition

In Figure 7.8 the results of the analysis using moment matching are displayed. In this graph, the analysis results are plotted with the lines. The confidence window is plotted with error bars and a dot indicating the average value obtained using simulation.

From this graph, it is visible that the analytical results are very close to the simulation results

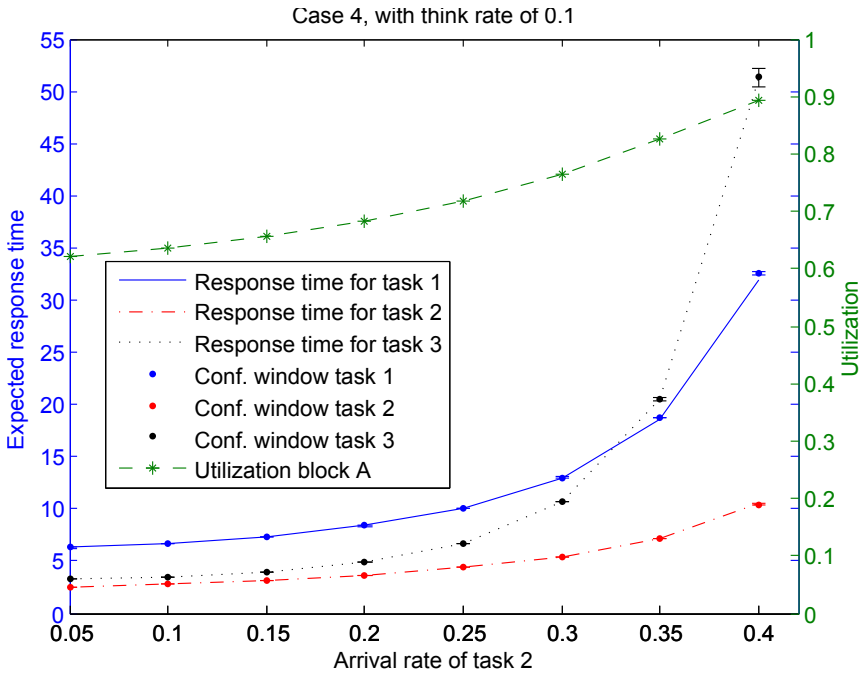


Figure 7.8: Case 4: Analysis versus simulation with $z_{sT1} = 0.1$

or even within the confidence window. In this graph only the expected response time of Task 1 computed using analysis is outside the confidence window of the simulation results, at an arrival rate for Task 2 (λ_{asT2}) of 0.4. Besides the expected response time of the tasks, also the utilization of block A is plotted, to give an indication on the utilization of the system. Figure 7.9 shows a similar graph as displayed in Figure 7.8, but now with a think rate for Task 1 (z_{sT1}) of 1.

To provide a more detailed view, in Table 7.2 both the analytical determined response times and the simulation results are listed for different values of the think rate for Task 1 and the arrival rate for Task 2. The columns "min" and "max" indicate the upper and lower bound of the confidence intervals. In the column "Difference" only percentages are displayed, when the analytical results are out of the confidence window of the simulation. From these results it clear that the analysis and simulation for this case match well, only minor differences occur with high utilizations by asynchronous calls. This is probably caused by the approximation made by applying moment matching instead of an exact distribution.

Parameter	Task numbers	Simulation response time			Analytical response time	Difference (%)
		min	avg	max		
$z_{sT1} = 0.1$	1	6.2747	6.2778	6.2810	6.2769	-
$\lambda_{asT2} = 0.05$	2	2.4921	2.4949	2.4977	2.4953	-
	3	3.2601	3.2621	3.2640	3.2619	-
$z_{sT1} = 0.1$	1	32.4335	32.5341	32.6347	31.8604	-2.0
$\lambda_{asT2} = 0.4$	2	10.4410	10.4764	10.5118	10.4868	-
	3	51.3058	51.7149	52.1240	51.2567	-0.9
$z_{sT1} = 1$	1	29.8648	29.9404	30.0161	30.0660	0.4
$\lambda_{asT2} = 0.4$	2	10.6240	10.6485	10.6731	10.6590	-
	3	55.2111	55.5041	55.7971	55.2507	-

Table 7.2: Case 4 results

As listed in Section 7.1 the generalized approach (waiting time propagation) for analysis of mul-

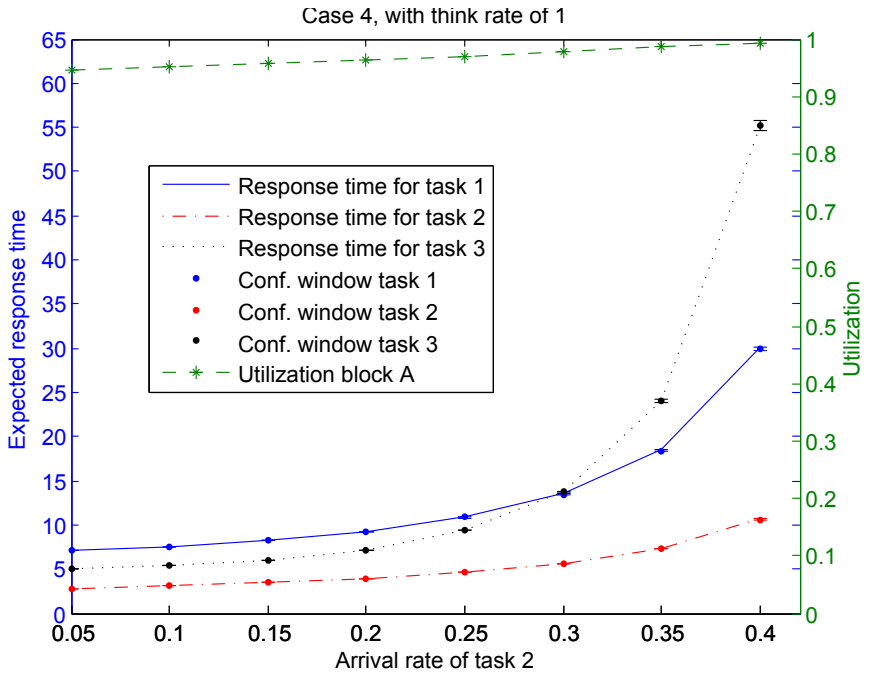


Figure 7.9: Case 4: Analysis versus simulation with $z_{sT1} = 1$

tiple ASD block is based on iteration. Therefore the number of iterations used, influences the results of the analysis. In Figure 7.10 the influence of the number of iterations on the response times of the tasks and the utilization of the blocks is plotted. In this plot a think rate for Task 1 (z_{sT1}) of 0.1 and an arrival rate for Task 2 (λ_{asT2}) of 0.4 are used. The left vertical axis provides the scale for all the response times, the right axis defines the scale for the utilizations.

In this graph it is shown that after two iterations the response times and utilizations are already quite close to the final value and after 6 iterations, the measures of interest do not noticeably change any more, therefore for this case 10 iterations is sufficient.

The plot shows a strong increase in the utilization for Block A with the second iteration. This strong increase is, because in the second iteration the waiting time of Task 1 at Block B is incorporated in the utilization of Block A, because the utilization includes the time the block is blocked. Since this results is an high utilization, the response time for Task 3 grows strongly. The increased utilization also results in an increased response time for Task 1, which increases the inter-arrival time for Task 1, because the response time is part of the inter-arrival time for synchronous tasks. This increased inter-arrival time results in a slight decrease in the utilization for Block B. This decrease is too small to have visible effect on the response time of Task 2.

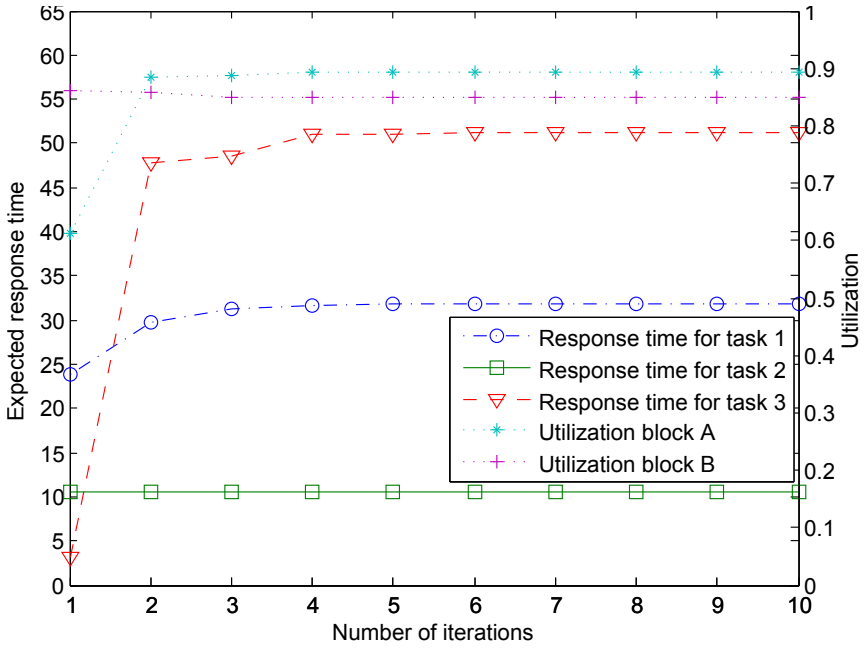


Figure 7.10: Case 4: Effect of number of iterations on response times

7.5 Case 5: Interference of tasks

The situation for this case is sketched in Figure 7.11. In this case one task (Task 1) consists of only synchronous calls going down trough the structure and the other task (Task 2) consists of asynchronous calls going upwards, using the same blocks, taking the opposite path. The corresponding task specifications are given in Table 7.3.

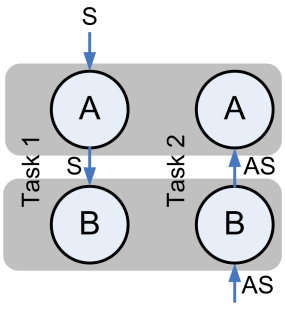


Figure 7.11: Case 5: Interfering tasks

Task number	Arrival / Think Rate	Expected service time
1	$z_s = 0.1$	$E[S_sA] = 3$ $E[S_sB] = 2$
2	$\lambda_{as} = 0.3$	$E[S_{as}B] = 2$ $E[S_{as}A] = 1$

Table 7.3: Case 5: task definition

In Figure 7.12 the analytically derived expected task response times are plotted together with the simulation results. The lines indicate the analysis results, the confidence window obtained using simulation is displayed by the error bars, with a dot indicating the average value. The utilization of block A plotted in this graph is obtained from the analysis and includes the blocking time. From this graph, it is shown that the analysis under estimates the task response times, but comes close to the confidence interval.

To give a more detailed view, in Table 7.4 the with analysis computed expected response times of the tasks are compared with the expected response times acquired by simulation. To provide more insight into the differences between the analytical results and the simulation results the waiting times and the utilization at each block are listed in Table 7.5. As well as in the previous tables the columns indicated with “min” and “max” indicate the upper and lower bounds of the confidence interval.

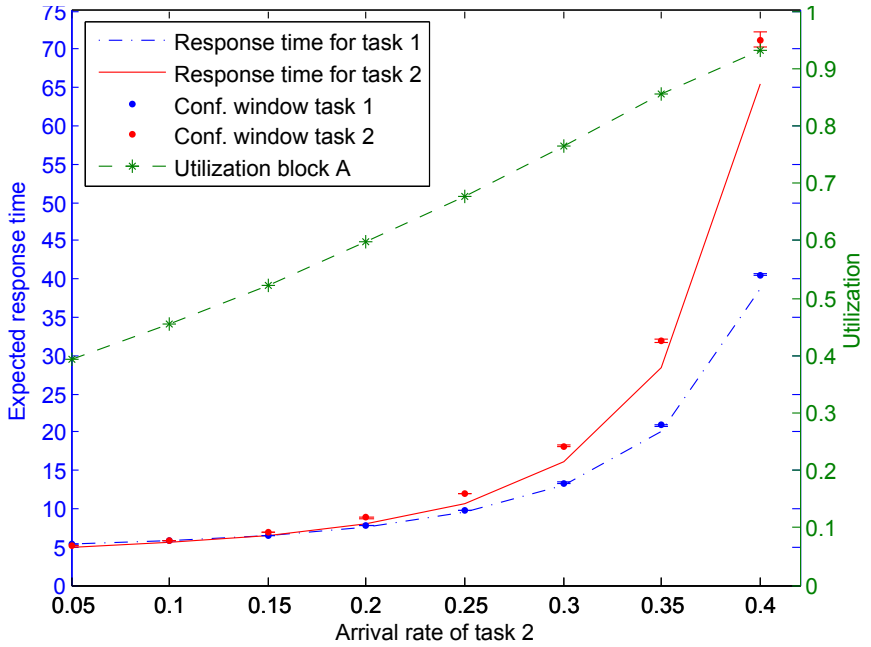


Figure 7.12: Case 5: Task response times

When looking at the task response times, the analytical results are not within the confidence interval. For this case the error in the response times is about 10 %, however when looking at the utilization of the blocks, they are almost the same. Despite this small deviation in utization, the waiting times at block A deviate almost 20 %.

Task number	Task response				Difference
	Simulation results			Analytical results	
	min	average	max		%
1	13.34	13.37	13.39	12.96	-3.1
2	18.17	18.21	18.25	16.06	-11.8

Table 7.4: Case 5: Task responses

Block number	Simulation results			Analytical results	Difference
	min	average	max		%
Expected waiting time for synchronous calls					
A	2.592	2.596	2.601	2.271	-12.5
B	5.753	5.768	5.782	5.689	-1.3
Expected waiting time for asynchronous calls					
A	11.747	11.779	11.812	9.625	-18.3
B	3.420	3.428	3.437	3.455	0.8
Utilization per block					
A	0.761	0.761	0.761	0.766	0.6
B	0.685	0.686	0.686	0.687	0.2

Table 7.5: Case 5: Various measures

The error in the expected response times is probably caused the opposite paths of the tasks (the synchronous task first does Block A then Block B, where Task 2 first does block B, then block A). The analysis for each block assumes negative exponentially distributed inter-arrival times for asynchronous calls. At block B asynchronous calls interfere with synchronous calls, which

have a different inter-arrival distribution. Therefore the inter-release time of asynchronous calls at block B, i.e., the inter-arrival times of the asynchronous calls at block A are not negative exponentially distributed. This effect is not incorporated in the model, introducing errors and further research is required to improve the accuracy in these cases.

Also in this case the results are acquired using iteration. The influence of the number of iteration on the task response time and the utilization is displayed in Figure 7.13. This plot is made with a arrival rate for Task 2 (λ_{as}) of 0.4. As in the previous case, in this case the results are close to their final values after 2 iterations and after 6 iterations no notable change is visible any more, therefore 10 iterations are sufficient for this example.

When looking at the utilization, the utilization of Block A increases significantly after 1 iteration, because in the second iteration, the waiting time Task 1 at Block B is embedded in the utilization of block A (due to the blocking, which is part of the utilization). This results in a longer response time of Task 1, which reduces the arrival rate of Task 1 (because the response time also determines the arrival rate), which reduces the utilization of Block B. Since the utilization at Block A has come close to 1, the waiting time for asynchronous calls at block A is increased strongly, lengthening the response time for Task 2.

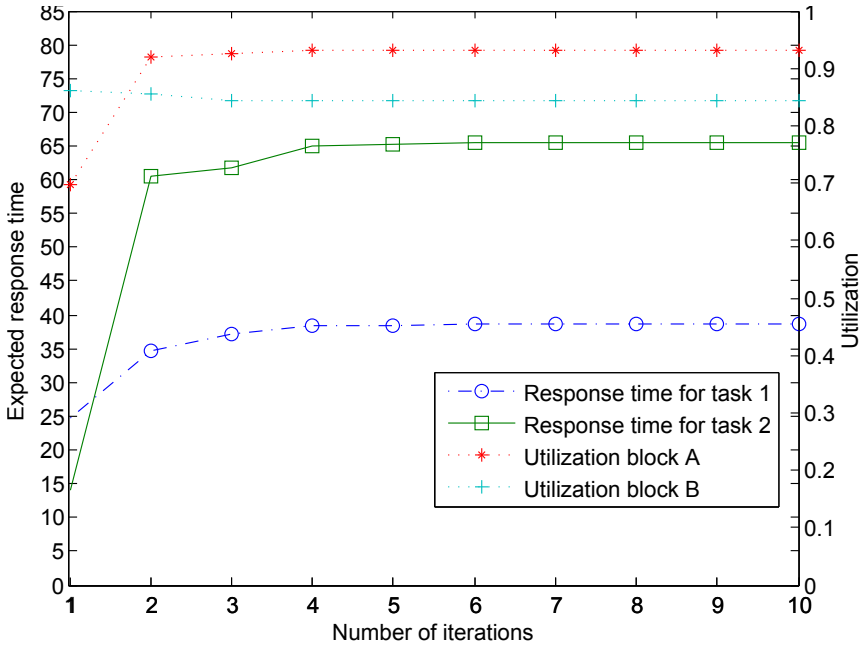


Figure 7.13: Case 5: Influence of iterations on measures of interest

7.6 Required number of iterations

As shown by the graphs of Figure 7.10 and 7.13, the iterative method used for propagating the dependencies between blocks converges for small cases rather quick close to the final solution. For larger cases however more iterations might be required.

A way of estimating the required number of iterations is to calculate the total change in expected waiting times between iterations and do iterations until this estimate is smaller then an allowed error. This condition is mathematically written down in Equation 7.15. For every block i take the absolute value of the current interation's (j) expected waiting time for synchronous calls minus the expected waiting time of the previous iteration ($j-1$) and sum them. Then add the same for the expected waiting time for asynchronous calls. If the result is smaller then the allowed error ϵ_{iter}, enough iterations have been done.

$$\sum_i \left| E[W_s]_i(j-1) - E[W_s]_i(j) \right| + \sum_i \left| E[W_{as}]_i(j-1) - E[W_{as}]_i(j) \right| < \epsilon_{iter} \tag{7.15}$$

7.7 Discussion

In Chapters 6 and 7 several ASD systems have been analysed and the obtained results have been compared with the simulation results. A summary of all results is listed in Table 7.6. In this table, for each type of ASD structure that has been studied, the analysis method is listed and the corresponding accuracy is indicated.

Nr	System	Analysis method	Accuracy	Section
1	Single block	Uses steady state probabilities of the underlying CTMC to compute the expected response times	++	6.1
2	One task, multiple blocks, either only synchronous or only asynchronous calls	Accumulation of single block response times	++	6.2
3	Two tasks (asynchronous and synchronous) sharing at most one block	Replace think and service transitions by phase-type distribution to incorporate dependencies	++	6.3 and 6.4
4	Multiple tasks (one synchronous with multiple asynchronous), where asynchronous tasks only interfere with the synchronous task at their last call	Replace think and service transitions by phase type transitions derived with **moment matching** to incorporate dependencies	+	7.4
5	Multiple tasks (one synchronous with multiple asynchronous), where asynchronous tasks only interfere with the synchronous task at multiple calls	„	+ / -	7.5
6	Multiple tasks, where each task consists either of only synchronous or only asynchronous calls	„	indication	-
7	Multiple tasks, where each task consists of combinations of synchronous and asynchronous calls	No method available yet	N/A	-

Table 7.6: Case comparison table

As can be seen in this table a good start has been made by modelling a simplified version of the ASD structure and deriving the performance analytically. For the first three cases of Table 7.6 the proposed analytical methods produce very accurate results. The methods used in these cases have been generalized to a technique called waiting time propagation that uses moment matching. This implementation of the waiting time propagation algorithm can also deal with first three cases and gives the same results as the simple techniques presented. The waiting time propagation in combination with moment matching algorithm works well for these smaller problems and is a promising approach for future work.

As presented in Section 7.5 (number 5 in the table), the analysis for the opposite paths of tasks only gives an indication of the performance. When a asynchronous task shares a block with synchronous calls, the assumption of negative exponentially distributed inter-arrival times is no longer true for this task at further blocks, because of the non-preemptive priority scheduling discipline of the ASD blocks. To incorporate this effect in the analysis, the inter-arrival time of asynchronous calls could be modelled as a phase type distribution using the moment matching algorithm. Since the current analysis method is based on negative exponentially distributed inter-arrival times for asynchronous calls, this also has to be adapted.

Even though the experienced think time is already modelled using a phase type distribution, a closer look is required on the situation with multiple synchronous tasks, to improve the results with case number 6 in Table 7.6. Since only one synchronous task can be in the system at a time, the response time of a tasks becomes part of the experienced think time of other tasks. This phenomenon is not yet incorporated in the analysis method.

The current modelling and analysis methods are already capable of analysing a lot of different ASD systems well. With these proposed extensions probably even much larger systems can be analysed analytically. Although there is a long way to go before the system is applicable to real ASD systems, the results obtained using the current analytical algorithm are promising.

8 Conclusion

In Section 8.1 the conclusion on this thesis are given. Thereafter Section 8.2 presents some recommendations to improve the models and the corresponding analysis.

8.1 Conclusion

The Analytical Software Design (ASD) suite is one of the products of the company Verum. In this master project, performance models for software generated with the ASD suite have been developed. To keep the problem manageable, only a simplified version of the ASD structure has been modelled. To analyse the ASD structure, a bottom-up approach is applied. Single ASD blocks are modelled as queueing stations and analysed using queueing theory and measures, which has been derived from the underlying Continuous Time Markov Chain (CTMC).

To validate the analysis for the simplified ASD architecture, a discrete event simulator has been implemented. For a given ASD architecture and pre-defined tasks, the simulator can compute the average response times and their confidence intervals from the simulation results.

Using the simulator, the analysis of a single ASD block is validated and the analytically derived expected waiting times lie well within the confidence interval. The simulation results are the basis for extensions made to the single block ASD model to analyse systems with multiple ASD blocks. After manually constructing models for a number of cases with multiple blocks, a generalized method based on moment matching is developed, which adequately models simple systems with multiple ASD blocks. Although this method works well for simple systems with multiple blocks, with more complex systems the analysis is not accurate any more, because some dependencies that are in the ASD generated software are not incorporated in the model. Recommendations are presented to address this problem.

Several assumptions have been made about the ASD generated software, to make modelling and analysis possible. The structure of the ASD generated software as used in this thesis does not contain all constructions allowed by the ASD suite, because at the time of the start of this thesis no more details where available on the exact behaviour of the ASD generated software. Although deviations from the presented structure are allowed by the ASD suite, these software constructions can often not be formally verified by the ASD suite, so keeping close to the simplified structure may be a good practice and is recommended to Philips Healthcare.

Besides the assumption about the structure, also several entities (service times, inter-arrival times etc.) have been assumed to be negative exponentially distributed. This probably does not resemble the distribution in a real system. To closely mimic the behaviour of a real system, these distributions can be replaced by phase type distributions that can be configured to approximate the distribution of the entity in a ASD generated system. Replacing the negative exponentially distribution by phase-type distributions will result in a larger state-space of the underlying problem. However this does not pose any problem for the presented technique.

All together, a good start has been made in this master project by modelling the performance of a simplified version of the ASD structure. Quite complex configurations can be handled using analytical models. Extensions are required before the models can be applied in practice, but the results using the moment matching algorithm look promising.

8.2 Recommendations

Not all aspects of the ASD structure have been covered yet, the following list gives suggestions for new projects and improvements:

Organization The purpose of this master project was to model the performance as part of the Allegio project. Due to ambiguity about the funding of the Allegio project, the project had not started jet, so no case studies were available. Therefore it was a challenge to identify a reasonably simple problem to start the analysis on. The goal of this master project was to take an analytical approach, that is why the simplified ASD structure deviates from the real ASD structure. For future projects a real case study would be very useful.

Arrival rate of asynchronous calls Section 7.7 gives an overview of the quality of the techniques with different systems. The results start to deviate when asynchronous and synchronous tasks cross at multiple blocks. Propagating the effect of interfering synchronous calls on the inter-arrival time of asynchronous calls can probably help to improve the quality of the results.

MPA In this thesis, only the self developed method of waiting time propagation is studied. The Modular Performance Analysis (MPA) could also be suited to model this problem, but applying this method was out of the scope of this project. This method is probably easier to adapt to the real ASD structure and also offers methods to model the processor sharing, which is not implemented in this project. However the MPA has difficulties modelling cyclic dependencies, which occur with the synchronous calls. In some cases MPA will be easier to use, but it probably will not solve all problems.

Simulate more When a real case study is available the modelling could start by implementing modelling decisions in a simulator. Implementing these in an analytical model is usually much more difficult and time consuming than implementing them in a simulator. This way the impact of modelling decisions can be examined in an early stage and no time is waisted by implementing decisions that result in large modelling errors.

A Tensor sum and products

When composing CTMCs out of phase type transitions often tensor products and sums (also called Kronecker products and sums) are used.

The tensor product ($\otimes$) with input matrices **A** and **B** of size n_a, m_a and n_b, m_b results in a matrix of size $n_a \cdot n_b, m_a \cdot m_b$. The product is performed by replacing each element in the first matrix, by the second matrix multiplied by the replaced element. An example with matrices **A** and **B** (Equation A.1) results in Equation A.2. The order of the operation is determining the outcome as shown in the differences between Equations A.2 and A.3. So when actually applying the tensor product to compose a CTMC, the order should be consistent, otherwise a wrong CTMC is obtained.

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \text{ and } \mathbf{B} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \tag{A.1}$$

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} a & 2a & b & 2b \\ 3a & 4a & 3b & 4b \\ 5a & 6a & 5b & 6b \\ c & 2c & d & 2d \\ 3c & 4c & 3d & 4d \\ 5c & 6c & 5d & 6d \end{pmatrix} \tag{A.2}$$

$$\mathbf{B} \otimes \mathbf{A} = \begin{pmatrix} a & b & 2a & 2b \\ c & d & 2c & 2d \\ 3a & 3b & 4a & 4b \\ 3c & 3d & 4c & 4d \\ 5a & 5b & 6a & 6b \\ 5c & 5d & 6c & 6d \end{pmatrix} \tag{A.3}$$

When composing a CTMC with phase type transitions often also the tensor sum operation is needed. Take two phase type distributions, characterized by their square matrices $\mathbf{Q_1}$ and $\mathbf{Q_2}$. In the calculations the identity matrices $\mathbf{I_{q1}}$ (identity matrix, same size as matrix $\mathbf{Q_1}$) and $\mathbf{I_{q2}}$ (identity matrix, same size as matrix $\mathbf{Q_2}$) are used. The tensor sum ($\oplus$) is shown in Equation A.4. The order of the tensor sum is also affecting the outcome because this operation is based on the tensor product.

$$\mathbf{Q_t} = (\mathbf{Q_1} \otimes \mathbf{I_{q2}}) + (\mathbf{Q_2} \otimes \mathbf{I_{q1}}) = \mathbf{Q_1} \oplus \mathbf{Q_2} \tag{A.4}$$

More information on tensor algebra can be found in [12].

B Probability functions and their properties

B.1 Random variable

A variable with an uncertain value is called a random variable (RV). Two types of random variables exist, discrete and continuous. An example of a discrete random variable is the outcome of the throw of a dice. The result can be either 1, 2, 3, 4, 5 or 6. Besides the possible outcomes also the distribution is know. When using a fair dice, the probability of each outcome is equal to $\frac{1}{6}$.

An example of a continuous random variable is the temperature at noon tomorrow. The temperature is a continuous variable, because is can have every value i.e. the number of values is uncountable and infinite. This means the probability of having a value of exact $0.000\ldots^\circ$C is 0. However the probability of having a temperature between -0.05°C and 0.05°C may have a value larger than 0.

B.2 Probability functions of continuous random variable

As seen in the previous section, a random variable may have a known distribution. For a continuous random variable X this distribution is often defined by a *probability density function* (PDF), or in mathematical notation $f_X(x)$. This function describes for each value the contribution to the probability. Using this function $f_X(x)$, a *cumulative density function* (CPF) can be defined as shown in Equation B.1. This function takes into account the contribution to the probability of all the values smaller than x, so the CDF describes the probability of having a value smaller or equal to x ($\Pr(X \leq x) = F_X(x)$).

$$F_X(x) = \int_{-\infty}^{x} f_X(u)\,du \tag{B.1}$$

The probability of having any value is equal to 1 (by definition), so the integral over the contributions by every value (PDF) should equal 1. This is shown in Equation B.2.

$$\Pr(X \leq \infty) = F_X(\infty) = \int_{-\infty}^{\infty} f_X(u)\,du = 1 \tag{B.2}$$

B.3 Moments of a random variable

Not for every application the complete PDF of a random variable is required, but knowledge about the moments of a distribution suffices. The most well known moment is the first moment or *mean* value. The mean value is often determined over a large number of values, but in the context of a random variable a misleading term, because a random variable is only one value. Therefore the term expected value ($E[\cdots]$) is used when talking about random variables.

With discrete RV's the expected value is calculated by summing over all possible outcomes times their probability. This way the expected value of throwing a dice is calculated using Equation B.3

$$E[\text{dice throw}] = 1\cdot\frac{1}{6} + 2\cdot\frac{1}{6} + 3\cdot\frac{1}{6} + 4\cdot\frac{1}{6} + 5\cdot\frac{1}{6} + 6\cdot\frac{1}{6} = 3.5 \tag{B.3}$$

With continuous random variables the expected value has to be calculated using an integral function over all the values times their probability density as shown in Equation B.4.

$$E[X] = \int_{-\infty}^{\infty} x\cdot f_X(x)\;dx \tag{B.4}$$

Besides the expected value, also other moments can be calculated. The calculation of the i-th

moment of a RV X, is done by taking the infinite integral over x^i times the probability density at x. So the i-th moment ($E[X^i]$) can be calculated using Equation B.5.

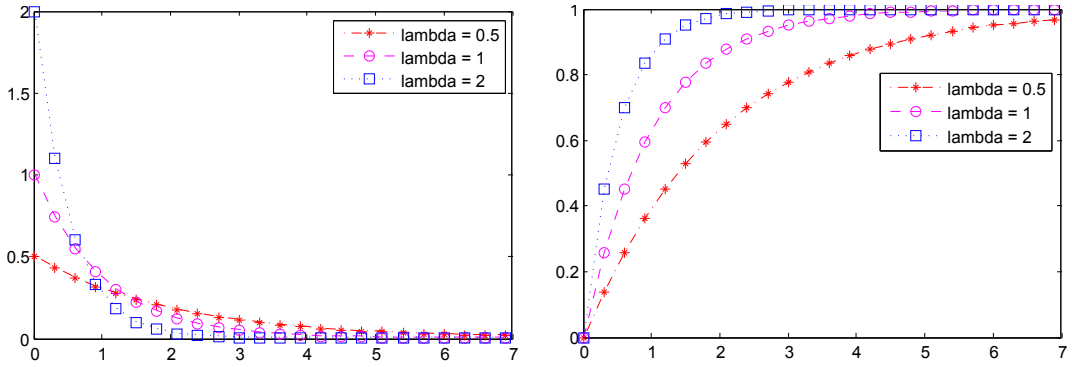
$$E[X^i] = \int_{-\infty}^{\infty} x^i \cdot f_X(x)\, dx \tag{B.5}$$

B.4 Negative exponential distribution

The negative exponential distribution is one of the basic continious distributions. A random variable that is negative exponentially distributed can only have values larger than 0. The PDF of the negative exponential distribution is given by Equation B.6 and its CDF by Equation B.7. These functions have a parameter $\lambda > 0$, which determines the exact shape and are only defined for $x > 0$. As an example, the PDF and CDF are plotted for several values of λ in Figure B.1.

$$f_X(x) = \lambda e^{-\lambda x} \tag{B.6}$$

$$F_X(x) = 1 - e^{-\lambda x} \tag{B.7}$$



(a) PDF (b) CDF

Figure B.1: The PDF and CDF of the negative exponential distribution

This distribution has a special property that it is *memoryless*. This property is discussed in detail in Section 3.7.

Using the definition of the moments, the expected value can be calculated using Equation B.8. Without the intermediate steps the second moment of the negative exponential distribution is given by Equation B.9.

$$\begin{aligned} E[X] &= \int_0^{\infty} x \cdot f_X(x)\, dx = \int_0^{\infty} x \cdot \lambda e^{-\lambda x} dx \\ &= \left[-xe^{-\lambda x}\right]_0^{\infty} - \int_0^{\infty} -e^{-\lambda x} dx = 0 - \left[\frac{1}{\lambda} e^{-\lambda x}\right] = -0 - -\frac{1}{\lambda} = \frac{1}{\lambda} \end{aligned} \tag{B.8}$$

$$E[X^2] = \int_0^{\infty} x^2 \cdot f_X(x)\, dx = \int_0^{\infty} x^2 \cdot \lambda e^{-\lambda x} dx = \frac{2}{\lambda^2} \tag{B.9}$$

B.5 Erlang distribution

Another well known distribution is the Erlang distribution. The Erlang-n distribution is constructed from a series of n independent negative exponential distributions with all the same λ. Since the Erlang distribution is totally composed of negative exponential distributions, the Erlang distribution is a phase type distribution.

The process behind the Erlang distribution is given in Figure B.2. The process starts in state 0 and the time it takes to go to the next state is negative exponentially distributed with parameter λ. The time is takes to reach state n starting from state 0 is Erlang-n distributed.

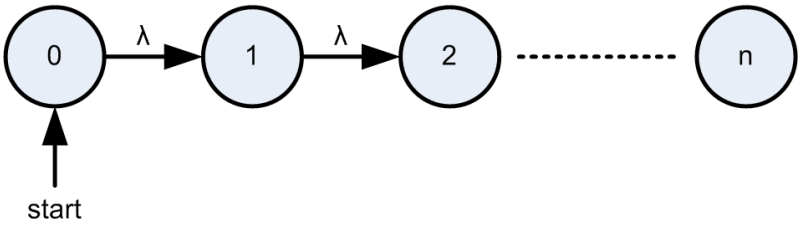
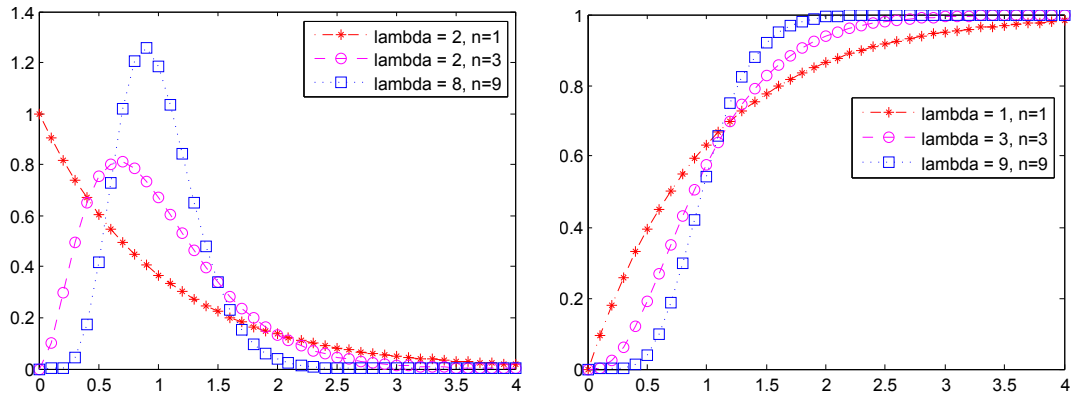


Figure B.2: Composition of the Erlang distribution

The PDF and CDF of the Erlang distribution are given by Equation B.10 and Equation B.11 respectively. These functions only hold for $x > 0$ and $n \in \mathcal{N}^+$. The functions are plotted with three different values for n and λ in Figure B.3. The values for n are chosen and the value for λ is adjusted to keep the expected values equal.

$$f_X(x) = \frac{\lambda(\lambda x)^{n-1}}{(n-1)!} e^{-\lambda x} \tag{B.10}$$

$$F_X(x) = 1 - e^{-\lambda x} \sum_{j=0}^{n-1} \frac{(\lambda x)^j}{j!} \tag{B.11}$$



(a) PDF (b) CDF

Figure B.3: The PDF and CDF of the Erlang distribution

The Erlang-1 distribution is actually a negative exponential distribution. When n becomes larger, the PDF gets a totally different shape and becomes a peak. The higher the n the narrower the peak around the expected value. This peak results in a sudden steep slope in the CDF graph. This effect makes the the Erlang behave like a deterministic distribution. The larger the n, the Erlang distribution resembles a deterministic distribution.

This deterministic behaviour allows the use of Erlang distributions to approximate deterministic variables. Since Erlang distributions are based on negative exponential distributions, analysis methods based on the Markov property (that cannot be used with deterministic transitions) can be used in this situation.

The expected value of the Erlang distribution is $\frac{n}{\lambda}$ and the second moment is $\frac{n(n+1)}{\lambda^2}$.

B.6 Hypo-exponential distribution

The hypo-exponential distribution is the generalized form of the Erlang distribution. Where the Erlang distribution has for each negative exponential distribution it is based on the same λ, the hypo-exponential distribution has for each phase a different λ. For a 2 phase hypo-exponential distribution with values $\lambda_1, \lambda_2 > 0$, the PDF and CDF for $x > 0$ are given by Equations B.12 and B.13.

$$f_X(x) = \frac{\lambda_1\lambda_2}{\lambda_2 - \lambda_1}\left(e^{-\lambda_1 x} - e^{-\lambda_2 x}\right) \tag{B.12}$$

$$F_X(x) = 1 - \frac{\lambda_2}{\lambda_2 - \lambda_1}e^{-\lambda_1 x} + \frac{\lambda_1}{\lambda_2 - \lambda_1}e^{-\lambda_2 x} \tag{B.13}$$

The expected value of a hypo-exponential distribution is $\sum_i \frac{1}{\lambda_i}$, which is de sum of the expected values for each negative exponential distribution it consists of. The second moment is more difficult, because the calculation of the second moment is non-linear, simply adding the second moments of the components is not allowed.

Therefore, the second moment has to be calculated from scratch using Equation B.5. Instead of using the PDF for the two stage hypo-exponential distribution of Equation B.12, the PDF of each stage its negative exponential distribution is used. By combining these two functions, this becomes Equation B.5. In this equation x_1 is the result of the first negative exponential distribution and x_2 of the second. Their sum is the result of the two phase hypo-exponential distribution. Both distributions are independent, so both PDFs can be multiplied.

In Equation B.15 the first and third term are the second moment of the exponential distribution i.e. $(2/\lambda_x^2)$. The second part is two times the multiplication of the mean values of the exponential distributions $(1/\lambda_i)$.

$$E[X_2^2] = \int_0^\infty \int_0^\infty (x_1 + x_2)^2 \lambda_1 e^{-\lambda_1 x_1} \lambda_2 e^{-\lambda_2 x_2} dx_1 dx_2 \tag{B.14}$$

$$= \int_0^\infty \int_0^\infty \left(x_1^2 + 2x_1x_2 + x_2^2\right) \lambda_1\lambda_2 e^{-\lambda_1 x_1 - \lambda_2 x_2} dx_1 dx_2 \tag{B.15}$$

$$= \frac{2}{\lambda_1^2} + 2 \cdot \frac{1}{\lambda_1} \cdot \frac{1}{\lambda_2} + \frac{2}{\lambda_2^2}$$

Based on this result, for prove using induction, suppose that for $E[X_n^2]$ (the second moment of a hypo-exponential distribution with n stages) Equation B.16 holds.

$$E[X_n^2] = \sum_{i=1}^{n}\left(\frac{1}{\lambda_i^2} + \sum_{j=1}^{n}\frac{1}{\lambda_i\lambda_j}\right) \tag{B.16}$$

Using Equation B.17 prove that Equation B.16 also holds for $n+1$.

$$(a+b) = a^2 + 2ab + b^2 \tag{B.17}$$

$$
\begin{aligned}
E[X_{n+1}^2] &= \int_0^\infty \cdots \int_0^\infty (x_1 + \cdots + x_{n+1})^2 \lambda_1 e^{-\lambda_1 x_1} \cdots \lambda_{n+1} e^{-\lambda_{n+1} x_{n+1}} dx_1 \cdots dx_{n+1} \\
&= E[X_n^2] \int_0^\infty \lambda_{n+1} e^{-\lambda_{n+1} x_{n+1}} dx_{n+1} + \\
&\qquad 2E[X_n] \int_0^\infty x_{n+1} \lambda_{n+1} e^{-\lambda_{n+1} x_{n+1}} dx_{n+1} + \\
&\qquad\qquad \int_0^\infty x_{n+1}^2 \lambda_{n+1} e^{-\lambda_{n+1} x_{n+1}} dx_{n+1} \cdot \prod_{i=1}^{n} \int_0^\infty \lambda_i e^{-\lambda_i x_i} dx_i \\
&= E[X_n^2] \cdot 1 + 2E[X_n] \frac{1}{\lambda_{n+1}} + \frac{2}{\lambda_{n+1}^2} \cdot \prod_{i=1}^{n} 1 \\
&= \sum_{i=1}^{n} \left(\frac{1}{\lambda_i^2} + \sum_{j=1}^{n} \frac{1}{\lambda_i \lambda_j} \right) + 2 \cdot \frac{1}{\lambda_{n+1}} \cdot \sum_{i=1}^{n} \frac{1}{\lambda_i} + \frac{1}{\lambda_{n+1}^2} \\
&= \sum_{i=1}^{n+1} \left(\frac{1}{\lambda_i^2} + \sum_{j=1}^{n+1} \frac{1}{\lambda_i \lambda_j} \right)
\end{aligned}
$$

Hereby using induction it is proven that Equation B.16 is true.

Bibliography

[1] CERN. Colt library; url: `http://acs.lbl.gov/software/colt/index.html`, 2010.

[2] Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *In DATE*, 2003.

[3] Alan Cobham. Priority assignment in waiting line problems. *Operations Research*, 2(1):70–76, 1954.

[4] Alan Cobham. Letter to the Editor–Priority Assignment–A Correction. *Operations Research*, 3(4):547, 1955.

[5] B. T. Doshi. Queueing systems with vacations - a survey. *Queueing Systems*, 1:29–66, 1986.

[6] B. Kienhuis et al. An approach for quantitative analysis of application-specific dataflow architectures. In *ASAP '97: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, page 338. IEEE Computer Society, 1997.

[7] F. Balarin et al. *Hardware-software co-design of embedded systems: the POLIS approach*. Kluwer Academic Publishers, 1997.

[8] Donald Gross and Douglas R. Miller. The randomization technique as a modeling tool and solution procedure for transient markov processes. *Operations Research*, 32(2):pp. 343–361, 1984.

[9] Boudewijn R. Haverkort. *Performance of Computer Communication Systems: A Model-Based Approach*. John Wiley & Sons, Inc., 1998.

[10] G. Latouche and V. Ramaswami. *Introduction to Matrix Analytic Methods in Stochastic Modeling*. ASA-SIAM, Philadelphia, 1999.

[11] John D. C. Little. A proof for the queuing formula: L=λw. *Operations Research*, 9(3):383–387, 1961.

[12] W.A. Massey. Open networks of queues - their algebraic structure and estimating their transient-behavior. *Advances in Applied Probability*, 16:pp. 176–201, 1984.

[13] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[14] R. Nelson. Matrix geometric solutions in markov models: A mathematical tutorial. Technical report, IBM Research Report RC 16777, 1991.

[15] Takayuki Osogami. *Analysis of multi-server systems via dimensionality reduction of markov chains*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005. Chair-Harchol-Balter, Mor.

[16] Verum. Url: `http://www.verum.com`, 2010.

[17] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System architecture evaluation using modular performance analysis: a case study. *Int. J. Softw. Tools Technol. Transf.*, 8(6):649–667, 2006.