Imperial College London
Department of Computing

Automated Construction of Petri Net Performance Models from High-Precision Location Tracking Data

Nikolas Anastasiou

March 6, 2013

Supervised by Dr William J. Knottenbelt

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College

Abstract

Stochastic performance models are widely used to analyse the performance and reliability of systems that involve the flow and processing of customers and resources. However, model formulation and parameterisation are traditionally manual and thus expensive, intrusive and error-prone.

This thesis illustrates the feasibility of automated performance model construction from high-precision location tracking data. In particular, we present a methodology based on a four-stage data processing pipeline which automatically constructs *Coloured Generalised Stochastic Petri Net* (CGSPN) performance models from an input dataset consisting of raw location tracking traces. The output performance model can be visualised using PIPE2, the platform independent Petri Net editor. The developed methodology can be applied to customer-processing systems which support multiple customers classes and can capture the initial and inter-routing probability of the customer flow of the underlying system. Furthermore, it detects any presence-based synchronisation conditions that may be inherent in the underlying system and the presence of service cycles. Service time distributions, one for each customer class, of each service area in the system and travelling time distributions between pairs of service areas are also characterised. PEPERCORN, the tool that implements the developed methodology, is also presented.

In addition to the latter, this thesis presents LocTrackJINQS, the extensible, location-aware Queueing Network simulator. LocTrackJINQS was developed to support location-based research. It has the ability to simulate a user-specified Queueing Network and while simulation progresses, it generates and outputs location tracking data – associated with the movement of the customers in the network – in a trace file.

Our methodology is evaluated through six case studies. These case studies use synthetic location tracking data generated by LocTrackJINQS. The obtained results suggest that the methodology can infer the abstract structure of the system – specified in terms of the locations and service radii of the system's service areas (max error 0.320 m and 0.277 m respectively) and customer flow – and approximate its service time delays well. In fact, the maximum relative entropy value that was obtained between the simulated and inferred service time distributions is 0.324 nats. Furthermore, whenever synchronisation between service areas takes place, the simulated synchronisation conditions are successfully inferred.

Acknowledgements

I would like to thank the following people:

- My supervisor, Dr. William Knottenbelt for his constant support, help, guidance and enthusiasm.

- Tzu-Ching Horng for her collaboration on generating synthetic location tracking data.

- Dr. Andrea Marin for his insightful comments which inspired part of this work, as well as for his collaboration.

- My friends for their constant support throughout the course of my PhD.

- The Engineering and Physical Sciences Research Council (EPSRC), as well as the Department of Computing, for providing me with the funding to do my PhD.

- Last but not least my family for their never-ending love, support and encouragement during my studies.

“A truly simple man is the richest of us all.”

This thesis is dedicated to Theodoulos Akathiotis; a truly simple man.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

We live in an information-driven era where the ever-progressing technology provides means of collecting vast amounts of data, especially location-based datasets. Naturally, people strive to extract meaning from this type of data to gain insights into the works, operation and performance of systems. This is a very broad subject and includes, amongst other areas, performance modelling [6, 57], human mobility characterisation [15, 47, 88] and urban planning [16, 28]; this thesis focuses on performance modelling and in particular, the modelling of physical systems that process customers and goods, for example hospitals, airports and car assembly lines.

In such complex systems it is critical to understand the flow of customers and/or resources to ensure that the system can be tuned to meet its Quality of Service (QoS) requirements. Failure to achieve this has led to high profile fiascos such as the opening of Heathrow Terminal 5 [35], the great increase in patient waiting times in hospitals [48], the immigration queues at Heathrow just before the London 2012 Olympics [75] and the delays experienced by people who wished to purchase Olympic football tickets on the spot, resulting in missing the event itself [81]. In an attempt to minimise and ultimately eliminate the occurrence of such incidents, much time and effort has been invested by the research community in performance modelling and analysis.

The traditional performance modelling and analysis pipeline consists of three stages: model construction, model validation and system analysis. The most fundamental stage of this pipeline is model construction as the accuracy of the model is crucial to ensure the validity of subsequent analysis. A model's accuracy, i.e. the degree to which the model reflects its underlying system, depends on its structure and its parameterisation. The establishment of an accurate model facilitates the extraction of useful information regarding customer and resource flow and the identification of bottlenecks via system analysis. Additionally, performance models can be used as predictive tools; a model can be modified to examine the system's performance under hypothetical scenarios such as the addition/removal of resources or increased workload.

The construction of an accurate model usually requires the availability of a large amount of data. Current techniques, such as time and motion studies, involve tedious manual tasks [58], e.g. inspection of video footage, questionnaires and manual collection of timing data, all of which may be error-prone since the human factor is present. This data-gathering process is not only time consuming, but more importantly, it can also disrupt the system's natural flow. This immediately compromises the reliability of the data and, subsequently, the reliability of the model itself.

Usually, the formulation of performance models for a physical customer-processing system requires that the modeller fully comprehends the works, interactions and dependencies between the underlying system's sub-processes. Previous work on modelling patient flow in an Accident and Emergency department by S. Au-Yeung [11] exemplified the difficulties of manual model construction and validation. In particular, Au-Yeung et *al.* [102] structured and parameterised a hierarchical multiclass Markovian Queueing Network. While there was good agreement between *mean* response times emerging from both the simulated model and the data, the *distributions* of response times were not well matched, and there was no straightforward way to identify the causes of discrepancies. In a future attempt [12] the authors modelled patient arrivals using time series, aiming to identify seasonal patterns. While their model provided good insight in walk-in patient arrivals, it failed to characterise ambulance arrivals adequately.

The recent development of real time location systems (RTLSs) enables the automatic and unobtrusive collection of large amounts of high-precision location tracking data in real time. These systems use a variety of technologies, such as RFID (Radio Frequency Identification), UWB (Ultra Wide Band) and Wi-Fi, and have been already deployed with the goal of enhancing performance and system safety, especially in the fields of supply chain management and healthcare. Specific examples include international car makers such as Toyota and Ford Motors which use RFID to monitor and control their automated production line [9]. Tagging systems are now used in hospitals to prevent newborn baby abductions [82] and to monitor the supply of blood and drugs [68, 113]. More example applications in healthcare and system safety can be found in [51, 111] and [67] respectively.

Some research efforts have been made towards the development of methodologies and software packages which enable the automatic construction of performance models. However, such methodologies either focus on software systems, e.g. [36, 27, 90], or are application specific [114, 71]. To the best of our knowledge, no existing methodology, or tool, allows the automatic extraction of performance models for generic physical customer-processing systems. Our work presents a high-level approach, based on four-stage data processing pipeline, which allows the automated extraction of Petri Net performance models (PNPMs) from high-precision location tracking data. Coloured Generalised Stochastic Petri Nets (CGSPNs) are selected as the underlying modelling formalism because of their inherent expression of synchronisation and concurrency. Furthermore, the availability of multiple token types in CGSPNs, instead of one and indistinguishable token type which is the case in Generalised Stochastic Petri Nets (GSPNs), enables the modelling of complex systems in a relatively compact way.

1.2 Modelling Assumptions

The domain of applicability of our methodology is limited to customer-processing systems which satisfy the following assumptions:

1. The various service points of the system from which customers request and receive service are stationary, i.e. their position within the system is fixed.

2. Only one customer can be serviced at each service point of the system during any time period. Furthermore, the next customer to be serviced at a service point is randomly selected; that is, if more than one customers are waiting to receive service from the same service point.

3. Each customer which enters the system is monitored using a separate location tracking tag, i.e. no tag recycling is supported, unless a tag's unique identifier can be reconfigured and then be used to monitor another customer.

4. The customers stop or move at a relatively low speed when are in the physical proximity of a service point, given that the customers wish to be serviced from that particular service point. The area within which this assumption applies is called service area.

5. If synchronisation takes place in the system, it can be detected only if it is defined in terms of the physical presence of customers or resources in service points. Furthermore, when synchronisation is detected, we assume that service only progresses when the synchronisation condition(s) is(are) met.

6. In the scenario where multiple customer classes are processed in a system, the class of each customer must be provided through its associated location updates or by a static mapping of each customer id (unique tag identifier) to the class it belongs to.

1.3 Objectives

The aims and objectives of this thesis are:

- To develop a methodology which, given a spatiotemporal trace of customer and resource flow in a customer-processing system, automatically:
  - infers the location and service radius of service centres, and paths of customer flow,

  - extracts service time and travelling time samples of customers in the system and characterises (or approximates) their underlying service time and travelling time distributions.
  - Hence constructs a CGSPN performance model of the customer/resource flow for a given system. The model should be able to capture and represent:
    * synchronisation dependencies between the system's service centres,
    * multiple customer classes,
    * cyclic services.

- To develop a software package which allows the formulation and simulation of a physical customer-processing system, and which can generate synthetic location traces approximating those of a real RTLS.
- To demonstrate the applicability of the methodology in a variety of case studies.

1.4 Contributions

This thesis presents a methodology which combines existing and new algorithms in order to automatically infer Petri Net performance models from location tracking data, without any prior knowledge regarding the system's structure and operation, given that the system satisfies the stated modelling assumptions.

We show how the location and service radius[1] of each service centre in a physical customer-processing system can be automatically inferred from the traces of the system's customer flow. Using this information we demonstrate that the flow of customers, specified in terms of the initial and inter-routing probabilities, can be easily obtained. Combining the spatial information for each service area, along with the spatiotemporal data which describe customer paths, we extract the service time obtained by each customer at each service centre. Similarly, we extract the travelling time for each customer while traversing between various service areas in the

[1]We assume a circular service area.

system. A conservative scheme to determine whether the processing of customers at each service area is subject to some presence-based synchronisation conditions is also introduced. A Hyper-Erlang distribution (HErD) [106, 42] is fitted to each set of the extracted travelling and service time samples. Upon completion, the model is exported in a non-standardised version of the PNML format [18, 55] that is compatible with PIPE2, an open-source Petri Net editor which also incorporates various analysis modules [20, 31]. Our methodology can be applied for both single and multiple customer class systems. Support for the latter has been enabled via the use of Coloured Generalised Stochastic Petri Nets [74] which we also use to accurately represent the customer flow in the model when cyclic services are present in the underlying system. The results we obtain through several case studies (using synthetically generated location tracking data) suggest that the structure and stochastic features of a large class of physical customer-processing systems can be successfully inferred.

The recent development and widespread adoption of RTLSs, especially in healthcare and industry, has generated new challenges and opportunities for location-based research. However, experiments used to validate such research are, in general, difficult to formulate and require a large amount of time and resources [57]. In this thesis we introduce LocTrackJINQS, an open-source simulation library for constructing simulations with location awareness, which has been originally developed in collaboration with T.-C. Horng to support our research. LocTrackJINQS is an extension of the Queueing Network simulation package JINQS [43] and offers a controlled environment where one can construct and simulate a real life customer-processing system as a Queueing Network. It allows users not only to specify high-level features of the network, i.e. customer flow and time delay distributions, but also low-level ones such as entities geographic locations and their moving speeds and paths. As the simulation progresses, LocTrackJINQS outputs location updates which approximate those of a real RTLS – for each participating entity – in a trace file.

1.5 Thesis Outline

The remainder of this thesis is organised as follows:

Chapter 2 describes the background theory to the work presented in this thesis. A basic introduction to the theory of Markov processes (discrete and continuous time Markov chains) is provided, followed by an overview of two modelling formalisms: Petri Nets, their subclasses, and Queueing Networks. We then present some basic definitions and results from the field of graph theory, as well as a discussion on clustering algorithms. We include a description of various techniques and tools that are used to parameterise stochastic models and an overview of PIPE2, the software we use to visualise our models. Next, we outline the existing technologies currently employed by RTLSs and conclude the chapter with a presentation of related research, giving emphasis on work relating to the automated construction and/or extraction of performance models, and data mining.

Chapter 3 introduces LocTrackJINQS, which we developed and used to generate location tracking data through simulation. Initially, we provide an overview of the simulator and its capabilities, and outline the main features that distinguish it from JINQS. An extensive description of its software architecture follows. Next, we present two case studies. The purpose of the first case study is to demonstrate the construction of location-enabled Queueing Network simulation through LocTrackJINQS's graphical user interface. Standard performance measures such as the mean and standard deviation of the response time for each customer class at each server, as well as for the whole system, are included. The second case study focuses on the evaluation of LocTrackJINQS operation through quantitative results.

Chapter 4 introduces the basic methodology we developed to enable the automated extraction and construction of a hierarchical Generalised Stochastic Petri Net (GSPN) performance model from location tracking data. The methodology is based on a four-stage data processing pipeline for which a detailed description is provided. The chapter concludes with an evaluation of the pipeline through two case studies for which the relevant results are presented.

Chapter 5 realises and overcomes a major deficiency of the original pipeline: the ability to explicitly capture synchronisation. A mechanism which implements the automated detection of synchronisation from location traces is presented and described extensively. In this work, we consider synchronisation to be defined in terms of the number of customers present in various service areas during the processing of customers at another service area. We investigate the impact of the presence of synchronisation on the previously extracted service time samples, and demonstrate how they can be adjusted according to the detected synchronisation conditions. Additionally, we show how these synchronisation conditions are incorporated in our model. A case study which aims to validate the developed mechanism is presented, followed by a discussion on the obtained results.

Chapter 6 presents further enhancements to the original processing pipeline. In particular, using CGSPNs we add support for multiple customer classes and advanced customer routing. The modelling approach taken to support multiple customer classes and its integration with the existing methodology is presented. Then, we demonstrate that systems which contain cyclic services cannot be modelled accurately using GSPNs, and explain how CGSPNs can be used to overcome this issue. We also show how such systems can be automatically identified from the structure of their corresponding Petri Net model, using graph theory. This chapter includes another feature which has been added to the initial methodology, namely the calculation and representation of inter-routing probabilities of the customer flow between service centres. We present the two techniques that can be used to model such probabilities and reason about our selection. The chapter concludes with an evaluation of these three additional features through three case studies.

Chapter 7 describes the design and implementation of the PEtri net PERformance model COnstRuctioN (PEPERCORN) tool. This tool implements the basic methodology presented in Chapter 4, and integrates it with the synchronisation detection mechanism (Chapter 5) and the pipeline extensions (Chapter 6). We note that PEPERCORN has been used to infer the models presented in the conducted case studies.

Chapter 8 concludes this thesis with a summary of the work presented. The chapter also provides a discussion on applications and further work opportunities.

Appendix A contains a simple user manual for LocTrackJINQS and presents the XML node definitions for each element of the Queueing Network.

Appendix B presents the additional results of the three case studies examined in Chapter 6.

1.6 Publications and Statement of Originality

I declare that this thesis was composed by myself, and that the work it presents is my own, unless stated otherwise.

The following publications arose from the work carried out during the course of this PhD:

- **5th International Workshop on Practical Applications of Stochastic Modelling (PASM 2011)** [56] presents LocTrackJINQS, a flexible and extensible spatiotemporal simulation tool for customer-processing systems, which is based on the multiclass Queueing Network simulation library JINQS. LocTrackJINQS allows the specification of realistic, low-level features which are found in the physical world and therefore, its simulations can approximate entities' physical movements in a real life system. During simulation the entities' movements are recorded in such a way that they are similar to data collected from an actual RTLS. The work presented in Chapter 3 is based on this paper. This is a joint work with Tzu-Ching Horng.
- **5th International ICST Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2011)** [6] presents an automated technique which takes as input high-precision location tracking data and constructs a hierarchical Generalised Stochastic Petri Net performance model of the underlying system. The developed methodology is based on four-stage data processing pipeline and aims to provide a high-level description of the customer flow in the system. The material presented in Chapter 4 is based on this paper.

- **8th European Performance Engineering Workshop (EPEW 2011)** [8] presents a mechanism which automatically detects presence-based synchronisation between service centres. We show how this mechanism is incorporated into our original methodology [6] and examine its applicability through a case study. The content of this paper is presented in Chapter 5.

- **4th ACM/SPEC International Conference on Performance Engineering (ICPE 2013)** [7] extends our existing methodology [6, 8] to support the (automated) modelling of systems with multiple customer classes and service cycles. These two features are enabled via the use of Coloured Generalised Stochastic Petri Nets. Furthermore, another feature is introduced: the calculation and representation of the inter-routing probability of the customer flow within the system. The material presented in this paper is included and extensively evaluated in Chapter 6.

- **10th International Conference on Quantitative Evaluation of SysTems (QEST 2013) - (Submitted)** This paper presents the PEtri net PERformance model COnstRuctioN (PEPERCORN) tool. This tool is a Java-based implementation of the methodology presented throughout Chapters 4, 5 and 6. A more detailed description of the material included in this paper, as well as an overview of PEPERCORN's software architecture is presented in Chapter 7.

Chapter 2

Background Theory

This chapter presents the background theory underlying the work in this thesis, as well as work which shares some high-level similarities with ours. We begin by providing a brief introduction of *stochastic processes* and a general overview of *Petri Nets* as a modelling formalism. The basic structure and parameters of *Queueing Networks* is then presented, followed by some key definitions and results from the field of *graph theory.* A discussion on *clustering algorithms* as well as *distribution fitting* techniques and tools is also provided. We then present *PIPE2*, an open-source Petri Net editor, and a high-level overview of existing *location tracking technologies.* This chapter concludes with a section on related research mainly from the fields of performance modelling and data mining.

2.1 Stochastic Processes

A *stochastic process* is a mathematical model used to describe the evolution of an empirical process or system which exhibits some probabilistic behaviour. Such processes can be defined by the set of all possible states that they may reach and the set of probabilities that specify the transitions between possible states. Mathematically, a stochastic process is defined as follows:

Definition 2.1. *A stochastic process is a family of random variables $\{X(t), t \in T\}$ defined over the same probability space and taking values in the set S. The parameter t usually denotes* *time and it is used to index each random variable. S contains the values which can be obtained by the stochastic process, also known as states, and therefore S is called the state space of the process.*

Stochastic processes can be categorised according to their state space and index (time) parameter. If the state space is discrete then the corresponding stochastic process is said to be a *discrete-state* process or *chain*; otherwise, if state space continuous, it is called a *continuous-space* process. Similarly, the parameter set T, i.e. the values of t, can be either discrete or continuous. In the latter case the stochastic process is called a *continuous-time* process while in the former case it is referred to as *discrete-time* process or sometimes as a stochastic *sequence*.

A particular class of stochastic processes, the *Markov* processes, has been of great interest in the research community as it finds numerous applications in many branches of science and engineering amongst other fields. The next subsection introduces the *Markov propery* which characterises such processes. A more formal introduction to the area of Markov processes, as well as proofs for the results we present can be found in [53, 84].

2.1.1 Markov Processes

A Markov process is a stochastic process whose evolution depends only on its current state, without being concerned with how the process arrived there. This is known as the *"memoryless"* or Markov property. Assuming a discrete-state space, a Markov process (Chain) is formally defined as follows:

Definition 2.2. *Let $\{X(t),\ t \in T\}$ be a stochastic process defined on the parameter set T and state space $S = \{x_i = i,\ i \in \mathbb{N}_0{}^1\}$. The process is called a Markov process if it satisfies*

$$\begin{aligned} &P[X(t) = x \mid X(t_n) = x_n, X(t_{n-1}) = x_{n-1}, \ldots, X(t_0) = x_0] \\ &= P[X(t) = x \mid X(t_n) = x_n], \quad t > t_n > t_{n-1} \cdots > t_0 \end{aligned} \tag{2.1}$$

[1] $\mathbb{N}_0$ denotes the set of natural numbers including zero.

If the transitions between the states of a Markov Chain (MC) are independent of the time the chain has spent on its current state, then the chain is referred to as a *time homogeneous* MC, i.e. the MC is invariant to shifts of time. For a time-homogeneous MC the following condition holds:

$$P[X(t) = x \mid X(t_n) = x_n] = P[X(t - t_n) = x \mid X(0) = x_n] \tag{2.2}$$

For the remainder of this section we assume discrete-state space, time homogeneous Markov processes and we present results for *Discrete-time Markov Chains (DTMCs)* and *Continuous-time Markov Chains (CTMCs)* defined over discrete and continuous parameter sets respectively.

2.1.2 Discrete-Time Markov Chains

In the case of DTMCs equation 2.1 can be written as

$$P[X_n = j \mid X_{n-1} = i, X_{n-2} = k, \ldots, X_0 = l] = P[X_n = j \mid X_{n-1} = i] = p_{ij} \tag{2.3}$$

where i, j, k, l denote some state $s \in S$ and p_{ij} is called the *one-step transition probability*. That is the probability that the chain reaches state j given that it is currently at state i. These probabilities are usually displayed as the entries of an $N \times N$ stochastic matrix[2] P, called the *state transition probability matrix*, where N is the total number of states of the Markov chain.

Using a similar notation we define the *n-step transition probability* $p_{ij}(n)$ to be the probability that the MC reaches state j after n transitions given that currently is at state i. It can be easily shown [53] that the n-step transition probability can be expressed as the product of the probability that the process reaches some intermediate state k from state i in r steps, $0 < r < n$, and the probability of reaching state j from state k in $n - r$ steps, i.e.

$$p_{ij}(n) = \sum_k p_{ik}(r) p_{kj}(n - r) \tag{2.4}$$

[2]A stochastic matrix is a matrix where the sum of the entries of each row is equal to one, i.e. $\sum_j p_{ij} = 1$.

or in terms of $P^{(n)}$, the n-step transition matrix,

$$P^{(n)} = P^{(n-r)}P^{(r)} \tag{2.5}$$

Equation 2.4 is known as the *Chapman-Kolmogorov* equation and allows us to recursively calculate the n-step from the one-step transition probabilities.

In performance evaluation we are often interested in the limiting behaviour of the MC, i.e. as $n \to \infty$. This is known as the *steady state* or *stationary* probability distribution Π. This is a vector $\{\pi_j\}$ where each π_j denotes the probability that the MC will be in state j at some arbitrary time in the future; by the law of total probability

$$\sum_j \pi_j = 1 \tag{2.6}$$

The existence and uniqueness of such probabilities is subject to certain conditions that the DTMC must satisfy. In particular the DTMC must be *ergodic*. Informally this means that all states in S must be reachable from every other state in S and that S does not include any periodic states, i.e. occurring in multiples of a fixed number of steps η where $\eta \geq 2$. (for a formal definition we refer to [53]). The stationary probabilities can be calculated by solving the system of linear equations

$$\pi_j = \sum_k \pi_k p_{kj} \tag{2.7}$$

for $j = 1, \ldots, N$, where N is the total number of states, subject to the condition imposed by equation 2.6.

2.1.3 Continuous-Time Markov Chains

In CTMCs the transition probabilities between states are defined as functions of time. This is because a transition into another state may occur at any time point, as opposed to DTMCs were transitions occurred at fixed time points (steps). When the process enters a state it stays there for an amount of time t before moving to another state. t is known as the *sojourn time* or *holding time*. Furthermore, since the CTMC satisfies the memoryless property – defined in equation 2.1 – the sojourn time is exponentially distributed. In fact, the exponential distribution is the only continuous distribution which satisfies this property (proof can be seen in [53]).

As transitions from one state to another can occur at arbitrary time points, we use $q_{ii}(t)$ and $q_{ij}(t)$, defined as

$$q_{ii}(t) = \lim_{\Delta t \to 0} \frac{p_{ii}(t, t + \Delta t) - 1}{\Delta t}$$

$$q_{ij}(t) = \lim_{\Delta t \to 0} \frac{p_{ij}(t, t + \Delta t)}{\Delta t}, \ i \neq j$$

denote the *infinitesimal rates* according to which the CTMC transitions from one state to another; the chain leaves state i at time t with rate $-q_{ii}(t)$ and the chain transitions from state i to state j at time t with rate $q_{ij}(t)$. These are the elements of the *generator* matrix $Q(t)$ which is used to characterise the CTMC.

Again, we are interested in the long term behaviour of the chain. Similar to DTMCs we define the steady state probability distribution Π, $\Pi = \{\pi_j\}$, where π_j is the probability of finding the chain in state j as $t \to \infty$. For a homogeneous CTMC, the values of π_j can be determined from the solution of the system of linear equations

$$-q_{jj}\pi_j + \sum_{i \neq j} q_{ij}\pi_i = 0 \tag{2.8}$$

where $q_{ij} = q_{ij}(t)$, subject to the constraint specified by equation 2.6. As in the case for DTMCs, a unique solution exists if the CTMC is ergodic.

2.2 Petri Nets

Petri Nets (PNs), also known as *Place-Transition Nets (P-T Nets)*, originated from the work of Carl Adam Petri [89], and are now widely used as a modelling formalism across various research fields, such as computer systems, biochemistry, systems biology and physics [78, 91, 54, 73, 1], as well as in industry. Examples of the latter include, amongst others, manufacturing

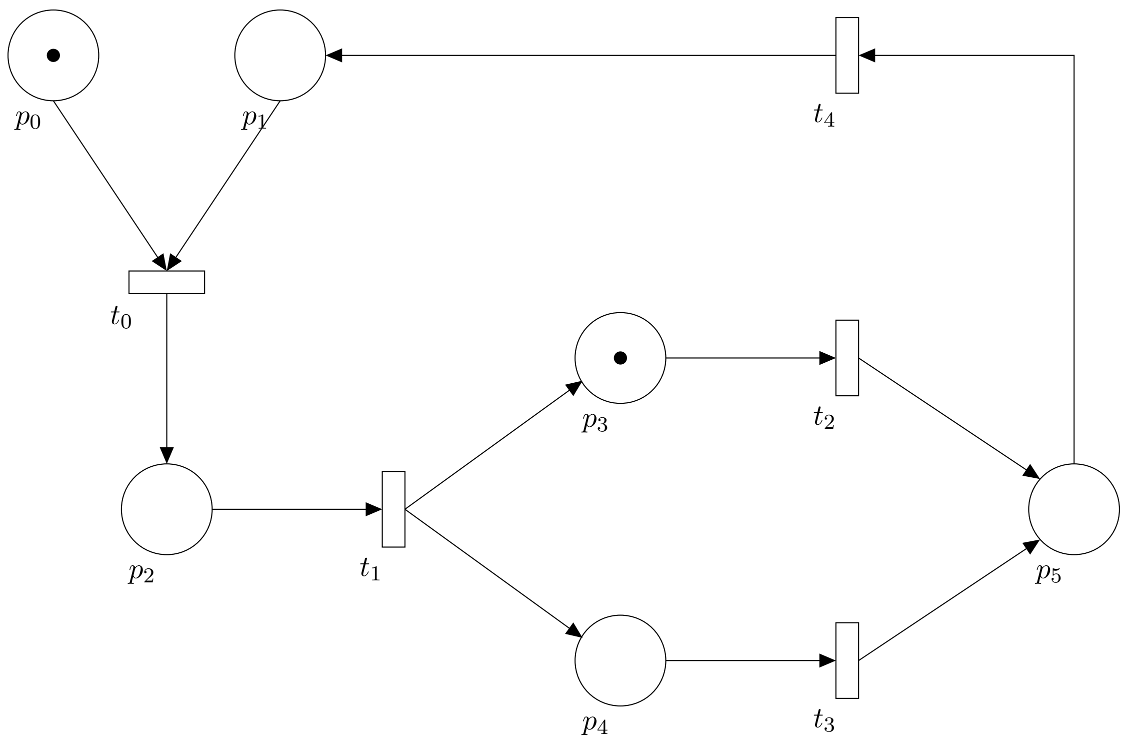


Figure 2.1: A simple Petri Net.

systems [114], workflow management [110] and traffic control [109].

The main advantage of PNs (and their subclasses) over other modelling formalisms, such as *Queueing Networks* [83] – both widely used in performance evaluation – is the inherent expression of synchronisation and concurrency. Furthermore, they provide an intuitive graphical representation of the dynamic behaviour of systems in conjunction with their mathematical nature. A typical Petri Net (see Figure 2.1) consists of:

- **tokens,** drawn by black dots. Tokens are usually used to represent the resources in a system.
- **places,** drawn by circles. Places represent the *resource holders* and may contain tokens.
- **transitions,** drawn by rectangles. Transitions model activities which change the state or availability of resources.
- **arcs,** specifying the interconnection of places and transitions, indicating which objects are changed by a certain activity.

Places may be only connected to transitions and vice versa. If an arc connects a place to a transition then the place is referred to as an input place of the transition and similarly, if an arc connects a transition to a place then it is referred to as an output place of the transition. The activity that changes the state of resources is the firing of an enabled transition. A transition is enabled if each of its input places contains at least the number of tokens specified by the weight of its input arcs. The firing of a transition results in the destruction and construction of a specific number of tokens from its input and output places respectively. The number of tokens to be created and destroyed is indicated by the arc weights. Formally,

Definition 2.3. *A Place-Transition Net is a 5-tuple $PN = (P, T, I^-, I^+, M_0)$, where*

- $P = \{p_1, \ldots, p_n\}$ *is a finite and non-empty set of places,*
- $T = \{t_1, \ldots, t_m\}$ *is a finite and non-empty set of transitions,*
- $P \cap T = \emptyset$
- $I^-, I^+ : P \times T \to \mathbb{N}_0$ *are the backward and forward incidence functions respectively,*
- $M_0 : P \to \mathbb{N}_0$ *is the initial marking.*

The 4-tuple $PN = (P, T, I^-, I^+)$ represents the structure and M_0 the initial token distribution of the model. The state of the Petri Net, identified by the number of tokens in places, is determined by the initial marking and any subsequent firings of the transitions which may occur according to the conditions specified by the backward and forward incidence functions I^- and I^+. Figure 2.2 demonstrates a simple case where one token is destroyed from each input place and another is created to each output place. Another slightly more complicated example is shown in Figure 2.3.

Definition 2.4. *Let $PN = (P, T, I^-, I^+, M_0)$ be a Place-Transition Net.*

- *Input places of transition t:* $\bullet t := \{p \in P \mid I^-(p, t) > 0\}$,
- *Output places of transition t:* $t\bullet := \{p \in P \mid I^+(p, t) > 0\}$,

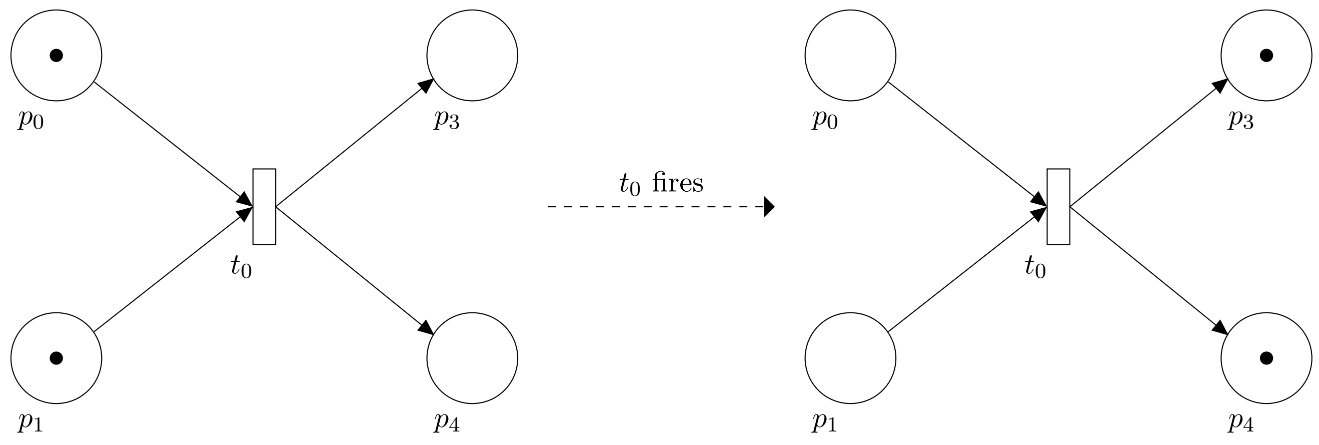


Figure 2.2: Firing of a transition in a Petri Net.

- *Input transitions of place p:* $\bullet p := \{t \in T \mid I^+(p,t) > 0\}$,
- *Output transitions of place p:* $p\bullet := \{p \in P \mid I^-(p,t) > 0\}$.

Initially Petri Nets were used to perform qualitative analysis of systems to detect various properties of the system, e.g. boundness, existence of deadlock etc. Their scope of applicability was limited as they did not incorporate any notion of time, and thus, restricted the use of PNs in performance evaluation of real systems.

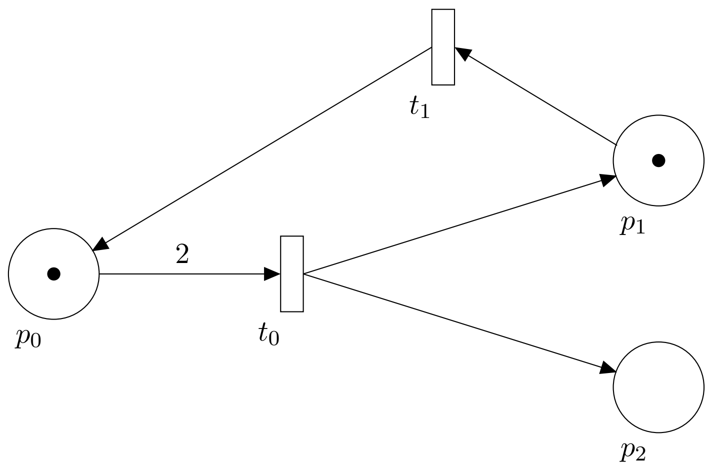


Figure 2.3: Transition t_0 requires two tokens to be contained in p_0 in order to be enabled.

In an attempt to increase the modelling power of Petri Nets many researchers have devised extensions, creating many variations and subclasses such as *Stochastic Petri Nets (SPNs)* [79, 80], *Generalised Stochastic Petri Nets (GSPNs)* [78], PNs with marking dependent arc cardinality [33], *Coloured Petri Nets (CPNs)* [62] and *Queueing Petri Nets (QPNs)* [13]. The most relevant of these variations are described in the following subsections.

2.2.1 Stochastic Petri Nets

Stochastic Petri Nets (SPNs) [79, 80] are similar to Place-Transition Nets, in terms of structure, but the firing of each transition involves an exponentially distributed delay. SPNs have been developed to enable the performance analysis of systems while preserving the intuitive graphical representation of Place-Transition Nets. Given an SPN, its underlying CTMC is isomorphic to the reachability graph of the SPN and is obtained via a depth-first search [14].

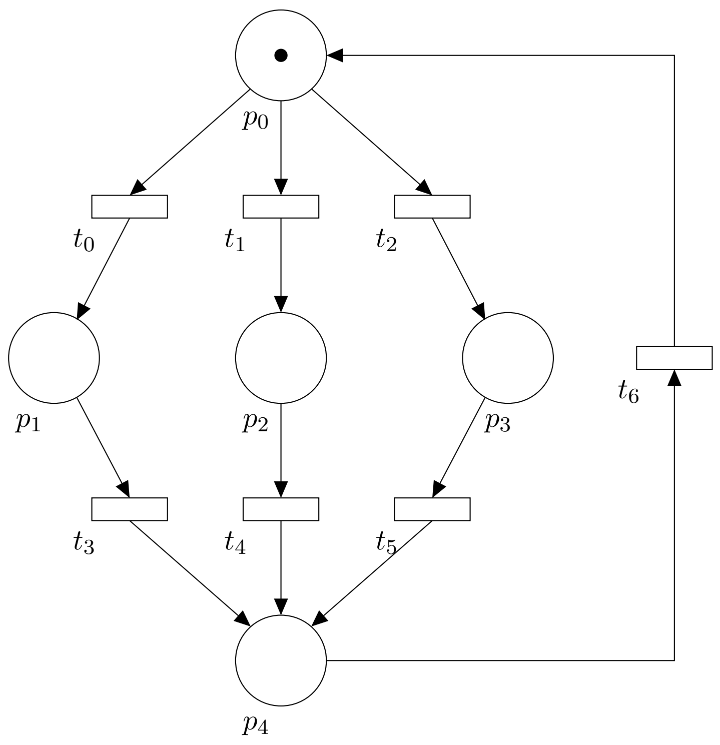
Definition 2.5. *A continuous-time Stochastic Petri Net is formed from a Place-Transition Net with the addition of a set of transition rates $\Lambda = \{\lambda_1, \ldots, \lambda_m\}$ to its definition, i.e. $SPN = (PN, \Lambda)$, where each λ_i, $i = 1 \ldots m$, is the firing rate of the transition t_i. The firing time follows an exponential distribution and the cumulative distribution function of the random variable χ_i of the firing delay of the transition t_i is given by,*

$$F_{\chi_i}(x) = 1 - e^{-\lambda_i x}$$

The sojourn time at a particular state (marking) is also exponentially distributed with the parameter λ equal to the sum of the individual rates of the enabled transitions in that state.

One example of an SPN is depicted in Figure 2.4. The initial marking of the SPN is $M_0 = (1, 0, 0, 0, 0)$ and therefore, transitions t_0, t_1 and t_2 are enabled. In this scenario we have a race condition; one of the enabled transitions will fire first, disabling the others. It is shown in [14] that given a set of n simultaneously enabled transitions at marking M, the probability of a transition t_i, $i = 0, \ldots, n-1$, firing is given by

$$P[t_i \text{ fires at } M] = \frac{\lambda_i}{\sum_{j=0}^{n-1} \lambda_j} \tag{2.9}$$



Transition	Firing Rate
t_0	λ_0
t_1	λ_1
t_2	λ_2
t_3	λ_3
t_4	λ_4
t_5	λ_5
t_6	λ_6

Figure 2.4: A Stochastic Petri Net.

2.2.2 Generalised Stochastic Petri Nets

Generalised Stochastic Petri Nets (GSPNs) inherit and extend the structure and behaviour of SPNs. GSPNs support another type of transitions, namely *immediate* transitions. These are graphically represented by filled rectangles, while timed transitions are represented by empty rectangles (see Figure 2.5).

Definition 2.6. *A Generalised Stochastic Petri Net is a 4-tuple* $GSPN = (PN, T_1, T_2, W)$ *where*

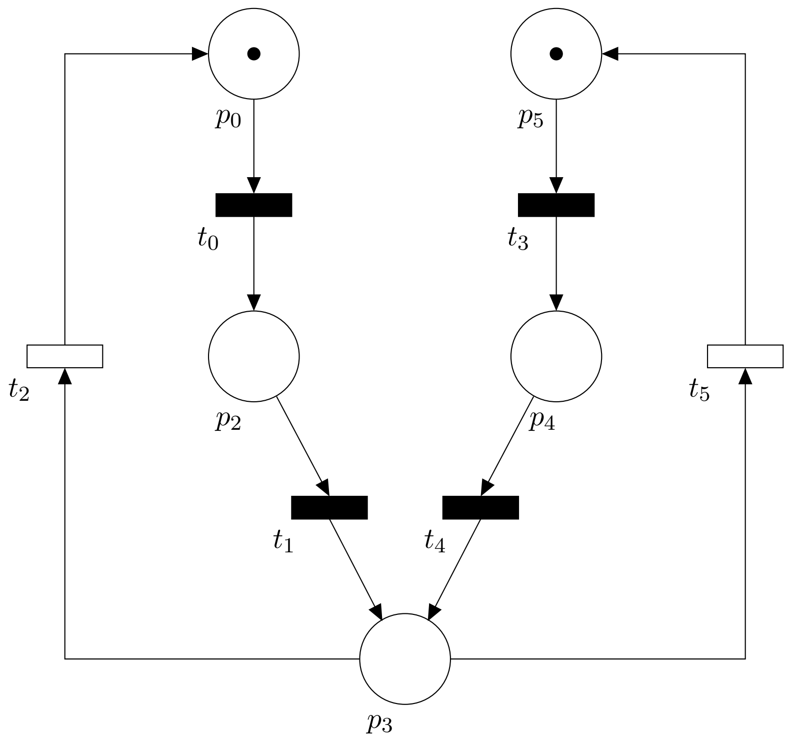
- $PN = (P, T, I^-, I^+, M_0)$ *is the underlying P-T Net,*
- $T_1 \subseteq T$ *is the set of timed transitions,* $T_1 \neq \emptyset$*,*
- $T_2 \subset T$ *is the set of immediate transitions,* $T_1 \cap T_2 = \emptyset$ *and* $T_1 \cup T_2 = T$*,*
- $W = (w_1, \ldots, w_{|T|})$ *is an array where each* $w_i \in \mathbb{R}^+$ [3] *is*

[3] $\mathbb{R}^+$ denotes the set of positive real numbers.

1. *a (possibly marking dependent) rate of an exponential distribution specifying the firing delay, when transition t_i is a timed transition, i.e. $t_i \in T_1$, or*

2. *a (possibly marking dependent) weight, which determines the firing frequency, when transition t_i is an immediate transition, i.e. $t_i \in T_2$.*

The dynamic behaviour of a GSPN follows the same rules as an SPN for timed transitions. Immediate transitions though, fire in zero time, i.e. as soon as they are enabled. Thus, enabled immediate transitions always fire before timed transitions. In the scenario of multiple, simultaneously enabled immediate transitions, the firing weights are taken into account; given a set of simultaneously enabled immediate transitions $\{t_i, i = 1, \ldots, n\}$ with corresponding weights $\{w_i, i = 1, \ldots, n\}$, the probability that t_i fires is given by $w_i / \sum_{j=1}^{n} w_j$.



Transition	Firing rate/Weight
t_0	w_0
t_1	w_1
t_2	λ_2
t_3	w_3
t_4	w_4
t_5	λ_5

Figure 2.5: A Generalised Stochastic Petri Net.

The analysis of GSPNs is also feasible but varies slightly from that of SPNs. The presence of immediate transitions may cause "multiple time discontinuities" between change of states and thus disrupts the correspondence between the reachability graph of the GPSN and a CTMC [78]. Markings that enable immediate transitions are called *vanishing states* as their sojourn time is zero and are never observed. Markings that enable timed transitions are called *tangible* *states* since the sojourn time is exponentially distributed and thus, such markings are not left immediately. Instead, a semi-Markov process can be extracted from the model as the probability of changing states remains independent of the time spent in one marking [14].

2.2.3 Coloured Petri Nets

Coloured Petri Nets (CPNs), introduced by K. Jensen [62], enhance the modelling power of P-T Nets by allowing different, but similar, processes to be described by a shared subnet.

In CPNs various types of tokens – each distinguished by a different colour – are supported. A set of colours is also attached to each place and transition in the model. The transition firing rules are similar to P-T Nets, except that functional dependencies are specified between the colour set of the transition firing and the colours of the tokens involved. Formally,

Definition 2.7. *A Coloured Petri Net is a 6-tuple $CPN = (P, T, C, I^-, I^+, M_0)$, where*

- $P = \{p_1, \ldots, p_n\}$ *is a finite and non-empty set of places,*
- $T = \{t_1, \ldots, t_m\}$ *is a finite and non-empty set of transitions,*
- $P \cap T = \emptyset$
- C *is a colour function defined from* $P \cup T$ *into finite and non-empty sets,*
- I^-, I^+ *are the backward and forward incidence functions defined on* $P \times T$ *such that* $I^-(p,t), I^+(p,t) \in [C(t) \to C(p)_{MS}], \forall (p,t) \in P \times T$,
- M_0 *is a function defined on* P *describing the initial marking such that* $M_0(p) \in C(p)_{MS}$[4], $\forall p \in P$.

This generalisation of P-T Nets reduces the size of the model required for system description and analysis. Figure 2.6 shows the P-T Net and CPN models for a dual processor system where each processor accesses a common bus, but not at the same time [14]. We notice that in

[4]$C(p)_{MS}$ denotes all the finite multisets over $C(p)$.

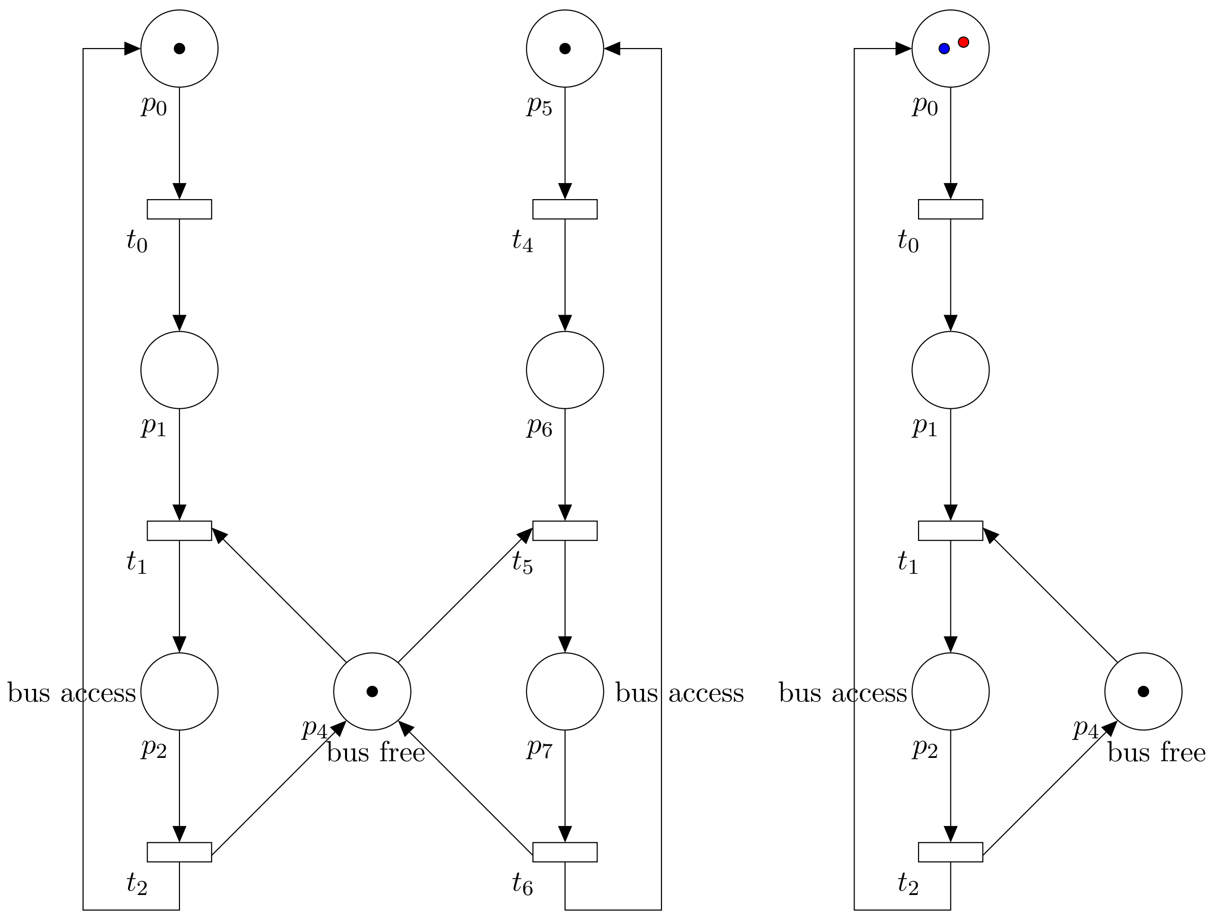


Figure 2.6: A dual processor system modelled using a P-T Net (left) and a CPN (right) [14].

the CPN representation of the system two different colours of tokens are used – one for each processor – resulting in a more compact model.

Furthermore, we note that every Coloured Petri Net can always be unfolded into an ordinary Petri Net in the following way:

1. $\forall p \in P, c \in C(p)$ create a place (p, c) of the Place-Transition.

2. $\forall t \in T, c' \in C(t)$ create a transition (t, c') of the Place-Transition Net.

3. Define the incidence functions of the P-T Net as
   $I^-((p,c)(t,c')) := I^-(p,t)(c')(c)$,
   $I^+((p,c)(t,c')) := I^+(p,t)(c')(c)$.

4. The initial marking of the P-T Net is given by
   $M_0((p,c)) = M_0(p)(c), \forall p \in P, c \in C(p)$.

The unfolded CPN is given by,

$$PN = (\bigcup_{p \in P} \bigcup_{c \in C(p)} (p, c), \bigcup_{t \in T} \bigcup_{c' \in C(t)} (t, c'), I^-, I^+, M_0)$$

CPNs, like Place-Transition Nets, evolved to facilitate the performance evaluation of systems [115, 116].

2.2.4 Coloured Generalised Stochastic Petri Nets

Coloured Generalised Stochastic Petri Nets (CGSPNs) [74] are a combination of CPNs and GSPNs. In this type of PN, transitions – both timed and immediate – can have a variety of firing modes depending on the coloured tokens supported in the model as well as the backward and forward incidence functions defined. CGSPNs are formally defined as follows:

Definition 2.8. *A Coloured GSPN (CGSPN) is a 4-tuple* $CGSPN = (CPN, T_1, T_2, W)$*, where*

- $CPN = (P, T, C, I^-, I^+, M_0)$ *is the underlying Coloured Petri Net,*
- $T_1 \subseteq T$ *is the non-empty set of timed transitions,*
- $T_2 \subset T$ *is the set of immediate transitions,* $T_1 \cap T_2 = \emptyset$, $T_1 \cup T_2 = T$,
- $W = (w_1, \ldots, w_{|T|})$ *is an array whose entry* w_i *is a function of* $[C(t_i) \to \mathbb{R}^+]$ *such that* $\forall c \in C(t_i) : w_i(c) \in \mathbb{R}^+$
  - *is a (possibly marking dependent) rate of a negative exponential distribution specifying the firing delay w.r.t. colour c, if* $t_i \in T_1$ *or*
  - *is a (possibly marking dependent) firing weight w.r.t. colour c, if* $t_i \in T_2$.

2.3 Queueing Networks

In addition to Petri Nets, another formalism which is widely used to model physical and computer systems, and analyse their performance is Queueing Networks. A Queueing Network is a directed graph $G = (V, E)$ where V is a set of nodes and E a set of weighted edges such that $E \subseteq V \times V$ [53].

Nodes represent service points, e.g. bank cashier, CPU, etc., and consist of a queue and one or more parallel servers. In queueing theory nodes are usually referred to as *Queues*. *Customers* flow through the service points of the Queueing Network probabilistically according to the weights of the node interconnections specified by the set E. Customers can either be physical entities, i.e. actual customers, products, and so forth, or software entities such as network messages, processes and so on, that require service or resources. When a customer arrives at a service point, it issues a service request and if the server(s) of that service point is(are) busy, the customer is placed temporarily in the queue; otherwise the customer is immediately serviced. A simple example of a Queueing Network consisting of two single-server queues is depicted in Figure 2.7.

In addition to the number of servers, several other parameters are also required to completely characterise a queue: customer arriving pattern, service pattern, queue capacity and queueing discipline. A unified, compact notation, known as the Kendall notation, uses five variables $A/B/X/Y/Z$ to represent all the aforementioned parameters where [83]:

- A denotes the customer arriving pattern, also known as customer inter-arrival time distribution,
- B denotes the service pattern, also known as service time distribution.
- X denotes the number of parallel servers,
- Y denotes the queue capacity and
- Z denotes the queueing discipline.

Some commonly used distributions for both inter-arrival and service times are M for Markovian (exponentially distributed time between arrivals or exponentially distributed service time), D for deterministic (constant), E_k for Erlang with k stages and G for general distribution [83].

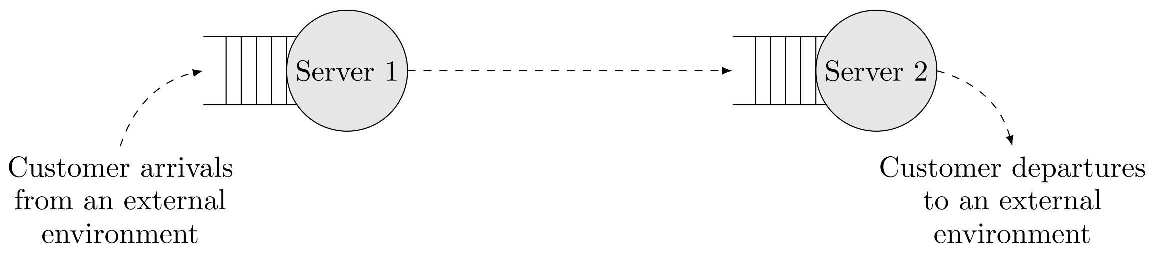

Server 1
Server 2
Customer arrivals from an external environment
Customer departures to an external environment

Figure 2.7: A simple example of a Queueing Network.

In cases where no explicit reference to the last two variables exists, e.g. $M/M/1$ queues, infinite queue capacity and *First-in-first-out (FIFO)* queueing discipline are assumed. Other common queueing disciplines are *Last-in-first-out (LIFO)*, *Priority*, *Random* and *Processor sharing (PS)*. Under FIFO and LIFO queueing disciplines customers have no priority and are removed from the head and the tail of the queue respectively. When the priority discipline is employed (each customer has an assigned priority) the customers with the highest priority are removed first from the queue. In random queueing discipline a customer is randomly chosen to be removed from the queue. The PS discipline is slightly more complex; all customers in the queue receive equal amounts of service. If a customer's service request is completed within the service interval provided by the server, then that customer exits the queue; otherwise the customer is placed at the end of the queue until all other customers which are ahead in the queue receive their share of service.

Furthermore, Queueing Networks can be one of three types: *open*, *closed* or *mixed* [53]. Open Queueing Networks have one or more external sources which inject customers into the network, and one or more sinks which represent the departures of customers from the network. On the other hand, in closed Queueing Networks neither customer arrivals, nor departures occur; the population of customers in the network is conserved and circulates the network's queues. Mixed Queueing Networks are a combination of the latter and former types and exist in the scenario of multiple customer classes. The flow of customers of one class may define an open network while the flow of another class may define a closed network.

2.4 Graph Theory

Graph theory is a branch of mathematics which originated from the work of Euler during his attempt to solve the Königsberg bridge problem. Currently, graph theory is being used in many areas of mathematics, science and technology. Focusing our interest in computer science, graph theory can be used in efficient algorithm construction, scheduling and network design. An overview of more specific application examples can be found in [96]. In particular, graph concepts can be easily applied to construct models of systems, to analyse their properties, and to provide insights into their optimisation. In this section we will present some basic concepts of graph theory and an algorithm for listing the *elementary cycles* of a directed graph [64]. If required, further information and theory regarding this subject can be found in [30].

2.4.1 Basic Concepts

Here, we refer to the terms directed graph and directed edge simply as graph and edge unless stated otherwise. We begin with the basic definition of a graph.

Definition 2.9. *A graph $G = (V, E)$ consists of*

- *a finite set $V = \{v_1, v2, \ldots, v_n\}$, whose elements are called vertices, and*
- *a subset E of the cartesian product $V \times V$, whose elements are called edges.*

Each edge $(v_i, v_j) \in E$ has two endpoints. Vertices v_i and v_j are called the initial and terminal endpoints respectively. In the case where $v_j = v_i$ the edge (v_i, v_i) is called a loop. Next we define the notion of adjacency.

Definition 2.10. *In a graph $G = (V, E)$ two vertices $v_i, v_j \in V$ are said to be adjacent if there exists an edge $e \in E$ whose endpoints are v_i, v_j. Two edges $e_i, e_j \in E$ are said to be adjacent if they have at least one common endpoint.*

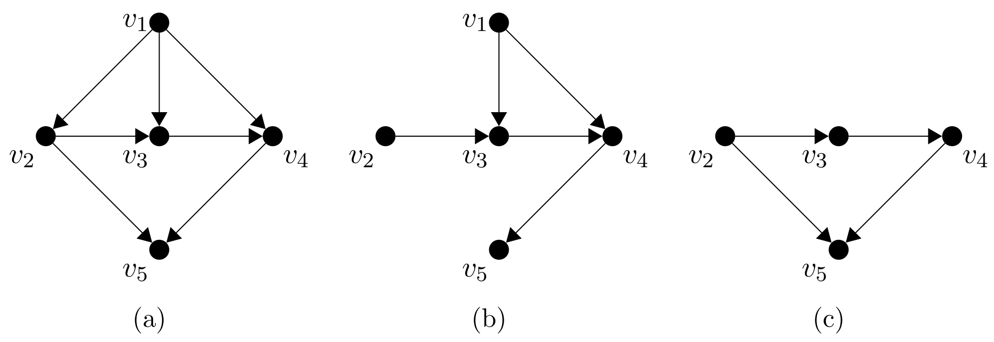


Figure 2.8: A graph G with five vertices and seven edges is shown in 2.8(a). 2.8(b) depicts the partial graph of G when edges (v_1, v_2) and (v_2, v_5) are removed. The subgraph of G, obtained by deleting vertex v_1, is shown in 2.8(c).

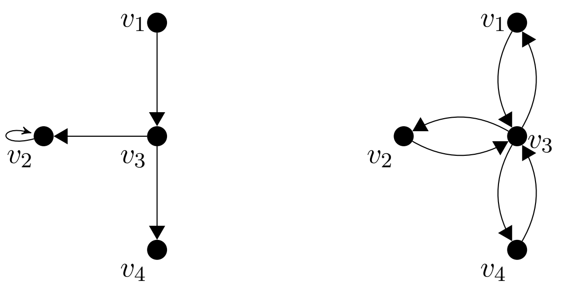
In addition, when two vertices $v_i, v_j \in V$ are adjacent through the edge $(v_i, v_j) \in E$ then v_j is called a *successor* of v_i and v_i is called a *predecessor* of v_j. The sets of all successors and predecessors of a vertex v_i are denoted by $\Gamma^+(v_i)$ and $\Gamma^-(v_i)$ respectively.

If a subset of edges is removed from a graph $G = (V, E)$, we obtain the partial graph of G, denoted by H, where $H = (V, E')$ and $E' \subset E$. If a subset of vertices V' is removed from G, along with all edges which have at least one endpoint $v_i \in V'$, we obtain the subgraph of G. An example of a graph along with its partial graph and subgraph is shown in Figure 2.8.

A finite sequence of edges in which the terminal endpoint of each edge is the initial endpoint of following edge is called a *path.* The initial endpoint of the first edge and the terminal endpoint of the final edge of the sequence are defined as the initial and terminal endpoints of the path respectively. A path which has the same initial and terminal endpoints is called a *cycle* or a *circuit.*

Definition 2.11. *A simple path, is a path which does not traverse any edge more than once. Similarly, an elementary path is defined as a path which does not traverse any vertex more than once. If a path is elementary and the initial and terminal endpoints coincide, it is then called an elementary cycle.*

Now, we revisit the notion of successors and predecessors of a vertex v_i and define the corresponding concepts when the edge (v_i, v_j) is replaced by a path $(v_i, v_k), (v_k, v_{k+1}), \ldots, (v_{k+s}, v_j)$. We say that v_j is a *descendant* of v_i and v_i is an *ascendant* of v_j if there exists a path from v_i to v_j. $\hat{\Gamma}^+(v_i)$ and $\hat{\Gamma}^-(v_i)$ denote the sets of descendants and ascendants of v_i respectively.



(a) G (b) G_S

Figure 2.9: Graph $G = (\{v_1, v_2, v_3, v_4\}, \{(v_1, v_3), (v_3, v_2), (v_2, v_2), (v_3, v_4)\})$ (2.9(a)) and its simplification $G_S = (\{v_1, v_2, v_3, v_4\}, \{(v_1, v_3), (v_3, v_1), (v_3, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_3)\})$ (2.9(b)).

Furthermore, v_j is said to be *accessible* from v_i if it is a descendant of v_i or $v_i = v_j$ and similarly, v_j is said to be *converse-accessible* from v_i if it is an ascendant of v_i or $v_i = v_j$. As before, the sets of vertices which are accessible and converse-accessible from v_i are denoted by $\overset{*}{\Gamma}^+(v_i)$ and $\overset{*}{\Gamma}^-(v_i)$ in respective order.

An important process in graph theory is the decomposition of a graph $G = (V, E)$ into its connected components. This process can be thought as decomposing several objects contained in the same set, into different object classes, where each object in each class is accessible from another object within the same class. Formally,

Definition 2.12. *Consider a binary relation, called the connectivity relation C, which is applied on the set of vertices V of any graph $G = (V, E)$. We write $v_i C v_j$ if $v_j \in \overset{*}{\Gamma}^+(v_i)$ on the simplification G_S of G. If $v_i C v_j$ holds then we say that v_i is connected to v_j.*

The simplification G_S is obtained from G by complementing its edges, i.e. wherever an edge (v_i, v_j) exists we add (v_j, v_i), and by removing its loops (see Figure 2.9). It can be easily shown that the relation C is reflexive, transitive and symmetric, and therefore, an equivalence relation [30]. The subgraphs of G which are generated by the equivalence classes produced by the partition V/C are called the *connected components* of G. A graph is said to be *connected* if it consists of only one connected component. A slightly altered version of Definition 2.12 defines *strong connectivity*:

Definition 2.13. *Consider the binary relation S, called the strong connectivity relation, which is applied on the set of vertices V of any graph $G = (V, E)$. We write $v_i S v_j$ if $v_j \in \overset{*}{\Gamma}^+(v_i)$ and $v_i \in \overset{*}{\Gamma}^+(v_j)$ on G. If $v_i S v_j$ holds then we say that v_i is strongly connected to v_j.*

Applying the same reasoning as above with respect to the relation S, instead of C, we can define *strongly connected components* and *strongly connected* graph.

2.4.2 Determining Elementary Cycles in a Graph

The first relatively efficient algorithm which could list all the elementary cycles in a directed graph was developed by Tiernan [107]. It performs an exhaustive search for cycles from each vertex in the graph using backtracking. Unfortunately, his algorithm is quite slow with a worst-case time complexity exponential in the number of circuits as well as the size of the graph. To this end, Tarjan [104] developed another algorithm to perform the same task whose running time is bounded by a polynomial function, i.e. $O((|V| \cdot |E|)(c+1))$ where c is the number of cycles in the graph $G = (V, E)$. The improvement observed in the running time of this algorithm is mainly due to the way it searches for the next vertex which extends the elementary path; the algorithm uses a depth-first search [103] to traverse the edges of the graph in an efficient manner.

Here, we present the algorithm developed by Johnson [64] which, although it resembles Tarjan's algorithm, realises a worst-case time complexity of $O((|V| + |E|)(c+1))$. The main difference of this algorithm, with respect to the previous ones, is that it considers each edge in the graph at most two times between the output of two consecutive circuits.

Algorithm 1 : `Unblock(int v)`[5]

```
1: blocked[v] ← false
2: for w ∈ B[v] do
3:   B[v].delete(w)
4:   if blocked[w] then
5:     Unblock(w)
6:   end if
7: end for
```

The algorithm requires as input an adjacency list, say A_G, which is used to describe the given graph $G = (V, E)$ and it outputs all elementary cycles that exist in it. It assumes that vertices are represented by distinct and consecutive integer values starting from one. The algorithm

[5]Modularised presentation of Johnson's algorithm as shown in [64].

Algorithm 2 : `Circuit(int v):Boolean`[5]

```
f ← false
stack.push(v)
blocked[v] ← true
for w ∈ $A_K$[v] do
  if w == s then
    {an elementary circuit has been identified}
    store stack followed by s
    f ← true
  else if blocked[w] == false then
    if Circuit(w) then
      f ← true
    end if
  end if
end for
if f then
  Unblock(v)
else
  for w ∈ $A_K$[v] do
    if B[w].contains(v) == false then
      B[w].add(v)
    end if
  end for
end if
stack.pop(v)
return f
```

begins with a root vertex s and eventually explores the set $\overset{*}{\Gamma}^{+}(s)$ of the subgraph $G_K = (V_K, E')$ of G induced by s and vertices "larger than s"[6] by performing recursive calls to a procedure called `Circuit` (see Algorithm 2) [64]. A stack is used to store the vertices of the elementary cycle currently being constructed in the appropriate order. $|V| + 1$ more lists are maintained: one logical array of size $|V|$, used to indicate which vertices are blocked (previously explored), and $|V|$ integer lists, referred to as B-lists, one for each vertex $v \in V$. These are used to store information regarding searches which did not yield an elementary cycle. In particular, a B-list for vertex, say w, $w \in V$, contains the predecessor vertices v of w given that the edge(v, w) has been already traversed, at least once, during the current search. When an unexplored vertex is reached through the current path, it is pushed into the stack and becomes blocked. Then, for each w, where w is a successor of the current vertex, two conditions are checked:

1. If w is equal to s then an elementary circuit is recorded and the procedure `Circuit`

[6]"larger than s" in some user-defined ordering of the vertices.

returns true.

2. If the latter is not the case and w is not blocked, a recursive call is made to the procedure `Circuit` with the starting vertex being w.

The algorithm also uses an auxiliary procedure called `Unblock` – depicted in Algorithm 1 – which takes as input a vertex v, and unblocks it. Also, this procedure removes and unblocks every predecessor vertex of v (found in the B-list of v) in a recursive manner. `Unblock` is called only whenever an elementary circuit is completed. The algorithm's initialisation procedure is shown in Algorithm 3. We note that the particular instantiation of A_K (cf. line 4) is not necessary for the correct operation of the algorithm. It simply guarantees that the algorithm is run only if at least one circuit exists; the algorithm can be directly applied on A_G as well.

Algorithm 3 : `Init`(A_G)[5]

```
1: stack.empty()
2: s ← 1
3: while s < $|V|$ do
4:   $A_K$ ← adjacency list of strong component $K$ with least vertex in subgraph of $G$ induced by
     $V_K = \{s, s+1, \dots, |V|\}$
5:   if !$A_K$.isEmpty() then
6:     s ← least vertex in $V_K$
7:     for $i \in V_K$ do
8:       blocked[i] ← false
9:       B[i].empty()
10:    end for
11:    dummy ← CIRCUIT(s)
12:    s ← s+1
13:  else
14:    s ← $|V|$
15:  end if
16: end while
```

2.5 Clustering Algorithms

During the past few years, in particular the last decade, great technological and scientific advances have been made in the fields, of machine learning, artificial intelligence and bioinformatics. The key to this progress is the analysis of vast flows of data and the identification of patterns that are hidden in them.

For the improvement of automatisation of several computing applications, where data manipulation and analysis is required, some form of unsupervised learning is essential. Several clustering algorithms exist that can be used to perform such tasks and which are chosen according to the type or structure of data at hand. Another factor which determines the choice of the appropriate clustering algorithm is the type of analysis desirable.

Clustering algorithms can be divided into several categories based on the way the data is grouped into clusters according to some, possibly user-defined, similarity measure. The two main categories are hierarchical and partitional clustering (see Figure 2.10).

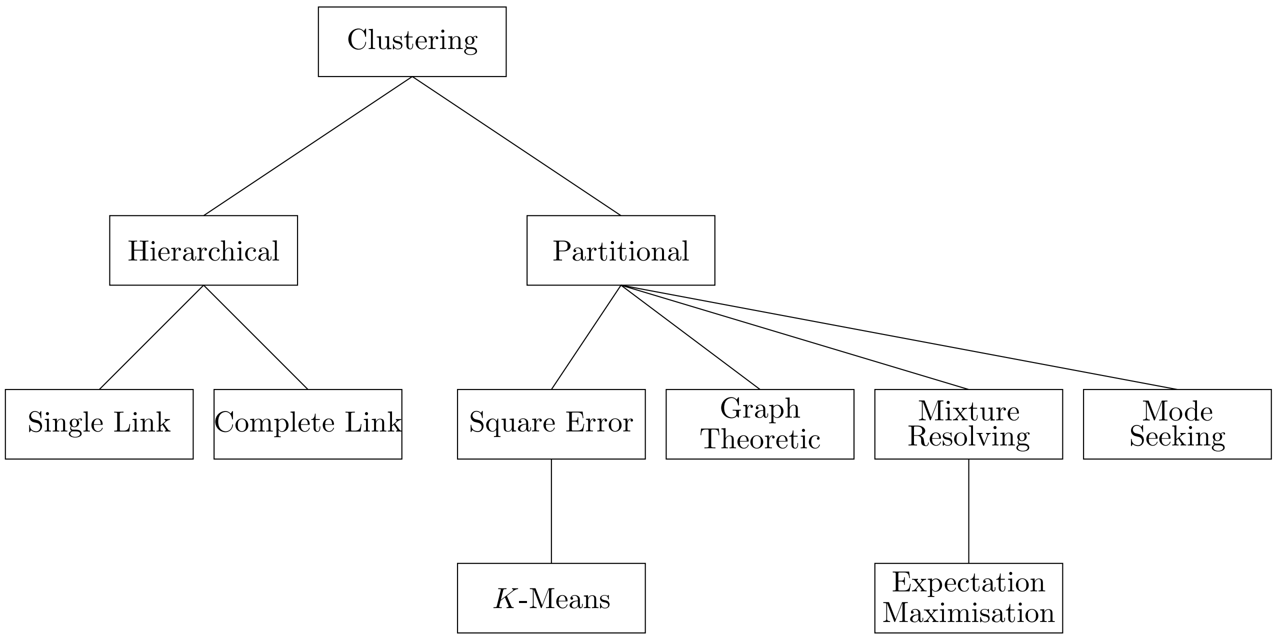


Figure 2.10: A taxonomy of clustering approaches [61].

Hierarchical clustering algorithms operate by decomposing a dataset D into smaller subsets until each subset contains only one element. These subsets merge or split to produce a nested series of partitions, represented by a dendrogram. The dendrogram can be created using two approaches; the *agglomerative* approach and the *divisive* approach. The former creates the dendrogram from the leaves up to the root while the latter from root down to the leaves [41]. Although this class of clustering algorithms does not require the number of partitions to be known beforehand, it requires the definition of a termination criterion to determine if the desired granularity has been achieved. When dealing with large datasets, hierarchical clustering algorithms are usually avoided since the construction of a dendrogram is computationally expensive.

For the remainder of this section we focus on partitional clustering and, in particular, we consider the *k-means* algorithm which is widely used mainly due to its easy implementation. This category of clustering algorithms requires an initial partition of the dataset D and the definition of a similarity criterion which is used to initially place the elements of D into the given partitions. Then, using the same similarity measure the elements are re-assigned to the newly computed partitions. The same process is repeated until the criterion is optimised, i.e. no further re-assignment of elements can be done. The major disadvantage of partitional algorithms is that they require as input the number of partitions, i.e. the number of output clusters. Here, we also present the DBSCAN clustering algorithm which introduces a density-based notion of clusters and lays the foundation of density-based clustering.

2.5.1 K-Means Clustering Algorithm

The k-means algorithm is very simple and works very well when dealing with isolated clusters. Apart from the selection of k – the number of clusters – it also requires an initial selection of centroids, one for each cluster. The centroid of a cluster is computed by taking the average of each cluster's contained elements on each dimension. After the successful completion of the algorithm each element is found in the cluster with the nearest centroid. K-means works with any choice of distance metric which is used to compute the similarity, or distance between an element and a centroid. Given that a similarity measure has been selected, the algorithm consists of four basic steps:

1. Define the number of clusters (value of k).
2. Define the initial centroids (one for each cluster).
3. Assign each element of the dataset to the nearest cluster.
4. Calculate the new centroid for each cluster.

Repeat steps three and four until each cluster's centroid remains constant.

The centroid initialisation (step two) is highly correlated to the clustering result since different choices of centroids usually produce different sets of clusters within the same dataset.

Robinson et *al.* explored through a case study fourteen different techniques to perform the initial centroid selection [93]. Their results indicate that the most efficient method is a synthetic one, namely the *scrambled midpoints*. This method divides the range of a dataset D into k equally spaced partitions. For example, if we are dealing with planar analysis, D consists of two dimensions: x and y coordinates. Assuming that the range of values is $[0.0, 20.0]$ and $[0.0, 15.0]$ for the x and y coordinates respectively, and that the value of k is five, i.e. we require five clusters, we need to divide each dimension's range into five equal sized partitions (see Table 2.1). For each computed partition of each dimension we compute its midpoint. A centroid is then calculated by randomly selecting one midpoint from each dimension.

Partitions for x coordinate	Midpoint	Partitions for y coordinate	Midpoint
$[0.0, 4.0]$	2.0	$[0.0, 3.0]$	1.5
$[4.0, 8.0]$	6.0	$[3.0, 6.0]$	4.5
$[8.0, 12.0]$	10.0	$[6.0, 9.0]$	7.5
$[12.0, 16.0]$	14.0	$[9.0, 12.0]$	10.5
$[16.0, 20.0]$	18.0	$[12.0, 15.0]$	13.5

Table 2.1: The partitions and midpoints for x and y coordinates with $k = 5$.

Another technique to perform the initial centroid selection was developed by Bradley and Fayyad [26]. Their work presents a refinement algorithm which is based on the idea of clustering clusters; multiple subsamples are randomly drawn from D and clustered producing estimates for the starting points. The k-means algorithm is applied on each subsample of D but its completion is determined by an additional condition: if any of the computed clusters has no membership, i.e. it is empty, then its centroid is re-assigned and the subsample is clustered again. The new centroid is chosen to be equal to the data element which differs the most from its cluster's centre. The solutions of each subsample, CM_i are then merged together providing a new dataset CM. Classic k-means is now applied to CM multiple times, each time using CM_i as starting points, producing a new solution FM_i. The FM_i with minimum distortion[7] over CM is chosen to be the the initial centroid selection for the application of k-means on D.

[7]Distortion in this context is interpreted as the sum of squared distances of each data element from its nearest mean.

2.5.2 DBSCAN Clustering Algorithm

The DBSCAN clustering algorithm [41] is a density-based algorithm which was developed for discovering clusters in large spatial databases and which also encapsulates the notion of noise. Clusters are defined as connected regions of high data-density. If the data-density of a region is less than a pre-defined threshold then this data is considered as noise.

If we consider a two-dimensional framework, DBSCAN operates in the following way: given a dataset consisting of n points, DBSCAN will divide them into a number of clusters according to the specified density threshold. This threshold is defined by choosing the minimum number of points, $MinPts$, in a circle of radius *Eps*.

The basic definition of DBSCAN is the *core point*. A point p is a core point if the circle of radius *Eps*, centred at p, contains more points than the value of $MinPts$. Having this in mind, a cluster is then defined to be the set of core points and *boundary points*. Boundary points lie on the boundary of the cluster, i.e. they lie in the circle of a core point but they are not core points themselves. The key function of the algorithm is called `ExpandCluster` (see Algorithm 5); this performs a spatial query which returns all points that lie in the circular region of radius *Eps* around p. It then examines if the number of points contained in this region is less than $MinPts$, and if that is the case, it classifies p as noise, or as core point otherwise. In turn, the algorithm examines all density-connected points, both core and boundary points, and groups them under the same cluster. The two functions of DBSCAN as described in [41] are presented in Algorithms 4 and 5.

Algorithm 4 : `DBSCAN(SetOfPoints,Eps,MinPts)`

```
1: {SetOfPoints is UNCLASSIFIED}
2: ClusterId ← nextId(NOISE)
3: for $i = 1$ to SetOfPoints.size() do
4:    Point ← SetOfPoints.get($i$)
5:    if Point.ClId == UNCLASSIFIED then
6:       if ExpandCluster(SetOfPoints,Point,ClusterId,Eps,MinPts) then
7:          ClusterId ← nextId(ClusterId)
8:       end if
9:    end if
10: end for
```

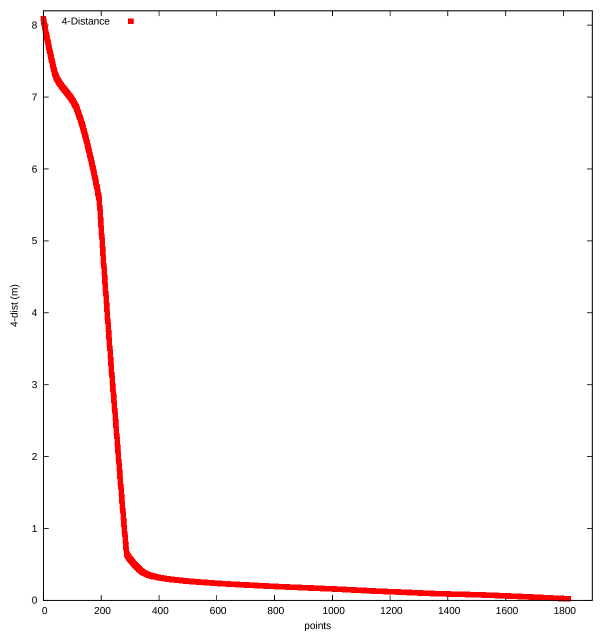
Like k-means, DBSCAN also requires two inputs: the value of *Eps* and the value of $MinPts$.

The authors of this algorithm tried to eliminate the need for determining these values. They defined a function called *k-dist* which maps each point in the dataset D to its kth nearest neighbour. The points were sorted in descending order of their *k-dist* value and plotted producing the *sorted k-dist graph*. Their experiments regarding the determination of optimal selection for k showed that graphs for values of k greater than four, i.e. $k > 4$, were quite similar and required more intense computation. Therefore, they decided to set the value of $MinPts$ equal to four, i.e. $MinPts = 4$, for all cases dealing with two-dimensional data, thus eliminating the need to specify this parameter. An example of a sorted *4-dist* graph is depicted in Figure 2.11(a).

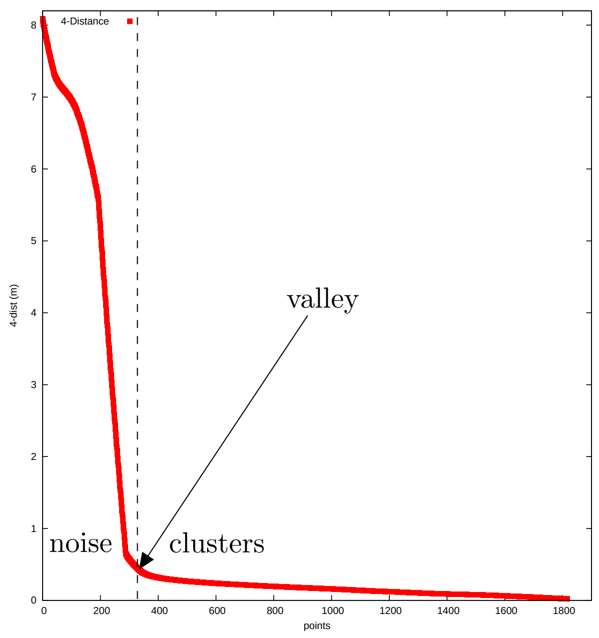
Algorithm 5 : `ExpandCluster(SetOfPoints,Point,ClId,Eps,MinPts):Boolean`

```
Seeds ← SetOfPoints.regionQuery(Point,Eps)
if seeds.size() < MinPts then
  {not a core point}
  SetOfPoints.changeClId(Point,NOISE)
  return false
else
  {all points in seeds are density-reachable from Point}
  SetOfPoints.changeClIds(seeds,ClId)
  seeds.delete(Point)
  while seeds.size() != 0 do
    currentPoint ← seeds.first()
    result ← SetOfPoints.regionQuery(currentPoint,Eps)
    if result.size() >= MinPts then
      for $i = 1$ to result.size() do
        resultPoint ← result.get($i$)
        if resultPoint.ClId == {UNCLASSIFIED or NOISE} then
          if resultPoint.ClId == UNCLASSIFIED then
            seeds.append(resultPoint)
          end if
          SetOfPoints.changeClId(resultPoint,ClId)
        end if
      end for
    end if
    seeds.delete(currentPoint)
  end while
  return true
end if
```

The value of the second parameter *Eps* depends on each particular dataset D and it can be easily determined by looking at the sorted *4-dist* graph of D. If a random point p is chosen and we set *Eps* to be the *4-dist*(p), then all points with *4-dist* less or equal to *4-dist*(p) will be core points (see Figure 2.11(b)). Thus, we need to identify a threshold point which has two



(a)



(b)

Figure 2.11: The sorted *4-dist* graph of a sample dataset D (left) and *Eps* selection (right).

properties: it has a maximal *4-dist* value and it is located in the "thinnest" cluster of D. The value of *Eps* is hence set to be the value of the first point in the first "valley" of the sorted *4-dist* graph [41]. Although this selection is fairly simple for a user when the graphical representation of this graph is available, it is very difficult to be determined automatically.

As an alternative, the authors proposed an interactive approach where the system computes and displays the *4-dist* graph to the user, and asks him to provide an estimate of the noise percentage. Then, the system proposes a value for *Eps* based on the input estimate. If the user is satisfied with the proposed value, the algorithm initiates, otherwise this selection process is repeated.

2.5.3 Clustering Algorithm Selection

Both k-means and DBSCAN are currently widely used in many data mining applications. As described in Section 2.5.1 k-means is very sensitive to the starting point selection. Using the refinement algorithm presented in [26] we managed to overcome the latter issue but we were not able to discard the need to enter the value of k – the number of clusters – manually. We have employed two different techniques: the application of G-means clustering algorithm which is a slight variation of the classical k-means, and brute force.

G-means [50] executes the k-means algorithm several times, while increasing the value of k on each run. On each execution each cluster is tested for member normality, i.e. whether the data points assigned to a cluster are normally distributed with respect to its centre. If a cluster fails this test (the test also requires a confidence interval to be set), its centre is replaced by two others. The second technique we tried runs k-means several times. We set k equal to one, i.e. $k = 1$, as a starting point and subsequently increased it by one on each execution. At the end of each execution an associated error was calculated, provided by an error function based on the intra-cluster distance, i.e. the distance of each member of a cluster from its centroid. This process was repeated until the total error of a particular partition was below a pre-specified threshold, or a pre-specified maximum value of k was reached. Both aforementioned techniques showed moderate success when tested in the scope of our application using synthetic location tracking data generated by the location-aware Queueing Network simulator LocTrackJINQS [56].

DBSCAN on the other hand, given that we fix the value of $MinPts$ to four, requires minimal or no domain knowledge to determine the remaining input parameter *Eps*. Since our work aims to keep user input at a minimum and we have managed to approximate the value of *Eps* by applying a simple interpercentile distance technique on the *4-dist*, we chose DBSCAN as our clustering algorithm.

We note that other density-based algorithms such as OPTICS and WaveCluster [10, 95] also exist. The main advantage of OPTICS against DBSCAN[8] is that it allows multiple distance parameter settings to be processed simultaneously. WaveCluster is a grid-based algorithm and it requires several input parameters such as the grid size on each dimension, the wavelet to use and the number of applications of the wavelet transform. However, DBSCAN performed very well in our framework and thus we did not investigate further into the other algorithms.

[8]DBSCAN and OPTICS operate using the same principle of density-connected points.

2.6 Estimating Model Parameters

An important part of performance modelling is the extraction of service and response time distribution(s), customer arrival distribution(s) and other statistical information so that the model accurately reflects the stochastic features of the underlying system. This process, in most cases, requires some distribution to be fitted to the available data. The most common technique used to perform this task is the *Expectation Maximisation (EM)* algorithm. It is essentially an iterative method, a form of parametric clustering, based on the assumption that the data to be clustered are drawn from one distribution. Its goal is to identify the parameters of the distribution by assigning a score – provided by the likelihood function – to each iteration while changing the parameters. The final parameters chosen are the ones that produced the maximum likelihood, given that the likelihood has converged. The limitation of this algorithm is that it needs to be tailored for a specific distribution, i.e. certain assumptions must be made regarding the distribution that the sample data follow. In this section we present the *Maximum Likelihood Estimation (MLE)* method and EM algorithm. A short description of the *hyper-Erlang* distribution along with an overview of the *G-FIT* tool [106] is also provided. We conclude this section with a presentation of the *Akaike Information Criterion (AIC)* and the concept of *Relative Entropy*.

2.6.1 Maximum Likelihood Estimation

MLE [4] is widely used to fit a probability density function to a dataset. Suppose that a probabilistic model depends on the set of parameters Θ and the dataset consists of N observations. We denote the dataset as $X = \{x_i\}_{i=1}^{N}$ and we assume that each x_i is independently drawn from the same probability density function $p(x \mid \Theta)$. The aim of this technique is to determine the values of Θ so they maximise the value (probability) of the likelihood function

$$P(X \mid \Theta) = \prod_{i=1}^{N} p(x_i \mid \Theta) \tag{2.10}$$

Computing the right hand side of equation 2.10 can be hard. Taking the logarithms of both sides we obtain,

$$\log(P(X \mid \Theta)) = \sum_{i=1}^{N} \log(p(x_i \mid \Theta)) \tag{2.11}$$

yielding the log-likelihood function, denoted by $L(\Theta \mid X)$. The solution

$$\Theta^* = \arg\max_{\Theta} L(\Theta \mid X) \tag{2.12}$$

can be obtained by solving the equations of the first derivatives of the log-likelihood function, with respect to Θ, set equal to zero.

2.6.2 EM Algorithm

The Expectation Maximisation (EM) algorithm [37] is a statistical tool widely used to estimate the parameters of an underlying statistical model Θ from a given, possibly incomplete, set of observations X. This is achieved through the maximisation of the log-likelihood function $L(\Theta \mid X)$ (defined in Equation 2.11). EM works iteratively and at each iteration i the updated value of $\Theta^{(i)}$ is calculated so that the difference

$$\log(P(X \mid \Theta^{(i)})) - \log(P(X \mid \Theta^{(i-1)})) \tag{2.13}$$

where $\Theta^{(i-1)}$ is the previously computed estimate for Θ, is maximised. This process terminates when the log-likelihood converges to some local maximum. The convergence of the algorithm is guaranteed [19] and the convergence rate depends on the set of initial parameters $\Theta^{(0)}$ which can be initialised via several strategies [17].

The EM algorithm is also used in cases where the maximisation of the log-likelihood function is analytically intractable. In this case, or in the case of an incomplete dataset, it can be assumed that a complete dataset Z exists which contains both the observed and unobserved data, i.e. $Z = \{X, Y\}$, and thus the joint density function $P(Z \mid \Theta) = P(X, Y \mid \Theta)$ can be specified. This density function is known as the complete-data log-likelihood $L(\Theta \mid Z)$.

Using Equation 2.13, $L(\Theta \mid Z)$ and Jensen's inequality, one can derive [21] that the updated value for Θ, $\Theta^{(i)}$, is given by the equation

$$\Theta^{(i)} = \arg\max_{\Theta}\{E_{Z \mid X, \Theta^{(i-1)}}\{\log(P(X, Y \mid \Theta)\}\} \tag{2.14}$$

The computation of the right hand side of Equation 2.14 is performed in two steps:

1. **The Expectation Step:** The evaluation of the conditional expectation $E_{Z \mid X, \Theta^{(i-1)}}\{\log(P(X, Y \mid \Theta)\}$ of the complete-data log-likelihood function, given the observed data X and the previously computed estimated parameter value $\Theta^{(i-1)}$. We note that this is evaluated as a function of Θ.

2. **The Maximisation Step:** The calculation of the current parameter value, $\Theta^{(i)}$. This calculation involves choosing an appropriate value for Θ so that the conditional expectation function is maximised.

Another version of this algorithm, known as the Generalised EM (GEM) algorithm [21, 19], performs the maximisation step in a slightly different way; instead of finding the value Θ which maximises the conditional expectation expression, it selects $\Theta^{(i)}$ so that

$$E_{Z \mid X, \Theta^{(i-1)}}\{\log(P(X, Y \mid \Theta^{(i)})\} > E_{Z \mid X, \Theta^{(i-1)}}\{\log(P(X, Y \mid \Theta)\} \tag{2.15}$$

2.6.3 The Hyper-Erlang Distribution and G-FIT

As stated earlier, the EM algorithm needs to be tailored to fit a particular distribution to the observed data. In the case where no prior knowledge exists for the data distribution, the latter can be a problem. To this end the work of Thümmler, Buchholz and Telek [106] provides a solution. They have developed a new tool, G-FIT, which fits a hyper-Erlang distribution (HErD), a kind of phase-type distribution, to the sample data using the EM algorithm. Their work relies on the theory that any general distribution of non-negative random variables can be approximated arbitrarily closely by a HErD with the correct choice of parameters.

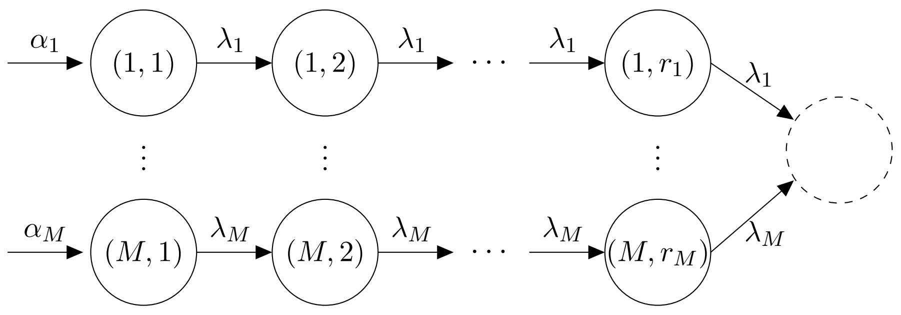


Figure 2.12: State transition graph of a hyper-Erlang distribution [106].

A HErD is defined as a weighted mixture of Erlang distributions and it belongs to the class of acyclic phase-type distributions. Formally,

Definition 2.14. *The probability density function of a HErD is*

$$f_X(x; M, \mathbf{r}, \boldsymbol{\alpha}, \boldsymbol{\lambda}) = \sum_{m=1}^{M} \alpha_m \frac{(\lambda_m x)^{r_m - 1}}{(r_m - 1)!} \lambda_m e^{-\lambda_m x} \tag{2.16}$$

where M is the number of Erlang branches, $\mathbf{r} = (r_1, r_2, ..., r_M) \in \mathbb{N}^M$ is the vector specifying the number of phases of each Erlang branch, $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, ..., \alpha_M) \in \mathbb{R}_+^M$ is the vector specifying the weight (contribution) of each Erlang branch, with $\sum_{m=1}^{M} \alpha_m = 1$, and $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, ..., \lambda_M) \in \mathbb{R}_+^M$ is the vector of scaling parameters (rates) (see Figure 2.12).

A common standardised measure of dispersion of a probability distribution is the coefficient of variation (CoV) which is a dimensionless number defined as,

$$CoV = \frac{\sigma}{\mu} \tag{2.17}$$

where σ and μ are the standard deviation and mean of the distribution. The CoV of a HErD in terms of the first and second moment is,

$$CoV = \sqrt{\frac{E[X^2]}{E[X]^2} - 1} \tag{2.18}$$

where the ith moment is given by,

$$E[X^i] = \sum_{m=1}^{M} \frac{\alpha_m (r_m - 1 + i)!}{(r_m - 1)! \lambda_m^i} \tag{2.19}$$

If we define the set

$$H = \{f_X(x; M, \mathbf{r}, \boldsymbol{\alpha}, \boldsymbol{\lambda})\}$$

i.e. the set containing all hyper-Erlang distribution models, then H has the following properties [42]:

1. H is a convex set. Therefore, $\forall f_X^1(x), f_X^2(x) \in H$ and $f_X^3(x) = c_1 f_X^1(x) + c2 f_X^2(x)$, $f_X^3(x) \in H$ with $c_1 + c_2 = 1$ and $c_1, c_2 \geq 0$. That is, any convex combination of hyper-Erlang distribution models is also in this set. This also holds for a combination of k HErD models.

2. Let F be the set containing all probability density functions of non-negative random variables. Then H is dense in F, i.e. any probability density function, $g_X^i(x) \in F$, can be approximated by a hyper-Erlang distribution model.

3. Hyper-Erlang distributions can have CoV less than, greater than and equal to one depending on the choice of parameters. Their CoV is naturally bounded below by zero but is unbounded above, i.e. it can be as large as desired.

Property 1 above can be easily verified by performing basic calculus. On the other hand, properties 2 and 3 require more complex calculations and theoretical background. Their proofs can be found in [65] and [42] respectively.

G-FIT operates using the EM algorithm tailored to the parameter estimation of a HErD. At each iteration the algorithm increases the log-likelihood function until either the maximal difference between consecutive values of the parameter vectors, or the relative difference of the log-likelihood function between successive iterations, falls below a pre-defined threshold. The computational complexity of both E- and M-step is $O(M \cdot K)$ where K is the sample size. Therefore, the overall computational complexity for each iteration is $O(M \cdot K)$. Moreover, G-FIT provides an option to find the "best" N-state HErD to be fitted where N is the total number of states. The algorithm enumerates all possible settings of M and $r_1, ..., r_M$ and fits a HErD for each case. Unfortunately, this is only possible for a small number of states ($N \leq 10$)

and samples ($K \leq 10^6$) as the number of possible settings grows exponentially for larger values of N. The authors of G-FIT [106] recommend two different strategies to be employed in such situations depending on the properties of the sample data distribution, i.e. CoV and heavy-tail, and one general strategy (progressive pre-selection).

A series of case studies fitting HErDs to six benchmark traces and two real traffic traces was performed [106]. The results indicate that G-FIT matches the first, second and third moments of the sample distribution with 0.0%, 0.8% − 69.4% and 1.2% − 94.3% error respectively, depending on the type of the sample distribution. Although it seems that the approximation is not always good – matching third moment with a maximum error of 94.3% – it is in most cases better than approximations obtained using the same sample data with PH-FIT, a phase-type distribution fitting tool [59]. In general, G-FIT is able to provide relatively fast and good results without much configuration and, most importantly, it is easy to automate. Its ability to be run as an automated module makes it a very attractive option for our research.

Another distribution fitting tool which was recently developed by Reinecke et *al.* [92], called the Hyper-* tool, showed to perform better than PH-FIT and G-FIT on three selected data sets. It follows a two-step process: firstly the samples are clustered using the k-means algorithm (see Section 2.5.1) and secondly an Erlang distribution is fitted to each computed cluster. Clusters are then refined using two possible criteria: highest density or probabilistic assignment. This process is repeated until convergence or until a maximum number of iterations is performed. The final result is a hyper-Erlang distribution where each branch is the Erlang distribution fitted to each of the final clusters and its weight is the relative size of the corresponding cluster. Unfortunately, Hyper-* requires user input as it employs the k-means algorithm and therefore is currently not suitable for our purpose[9].

2.6.4 The Akaike Information Criterion

Akaike's entropic information criterion known as AIC [3] is widely used to provide a measure of how good a statistical model is relative to the number of parameters that it employs. The

[9]With an alternative suitable clustering algorithm Hyper-* could possibly be completely automated.

problem that arises when multiple models are available for a given dataset, is which model to choose since no knowledge of the “true” model is available.

Given a general statistical model $f(\cdot|\Theta)$ and several competing models, each with a different set of parameters, we can represent each competing model as

$$\text{Model}(k) : f(\cdot|\Theta_k),\ \Theta_k = (\theta_1, \theta_2, \ldots, \theta_k) \tag{2.20}$$

where k is the number of free parameters of Model(k) and it is often called the dimension or order of the model.

The AIC(k) of each Model(k) is defined to be

$$AIC(k) = 2k - 2L(\Theta_k) \tag{2.21}$$

where $L(\Theta_k)$ denotes the log-likelihood function. A geometrically-based proof is provided in [23]. The model that minimises the value of AIC over the set of competing models is selected as the best approximation of the true model, i.e.

$$\{f(\cdot|\Theta_k)|\arg\min_{\Theta_k} AIC(k)\} \tag{2.22}$$

As it can be clearly seen from Equation 2.21, AIC rewards the goodness of a fit while adding a penalty proportional to the number of parameters of the model.

When dealing with finite and especially small samples of size, say n, AIC can become biased when the dimension k of a competing model increases with respect to n. Hurvich and Tsai [60] presented a bias-corrected version of AIC, the AICc which takes into account the sample size. AICc adds an extra penalty as k increases which is inversely proportional to n, thus eliminating the latter problem. That is,

$$AICc = AIC + \frac{2k(k+1)}{n-k-1} \tag{2.23}$$

From Equation 2.23 we notice that as the sample size becomes larger, AICc converges to AIC since

$$\lim_{n\to\infty} \frac{2k(k+1)}{n-k-1} = 0$$

2.6.5 Relative Entropy

Entropy, in the context of information theory, was first formally introduced by C. E. Shannon in [94]. In his work he formulated and extensively studied the problem of message passing through a channel, and set the foundation of communication theory. In an attempt to quantify "choice" or uncertainty, Shannon defined the entropy of a discrete random variable X with N possible events x_i each with probability p_i, $i = 1, \ldots, N$, as

$$H(X) = \sum_{i=1}^{N} p_i \log \frac{1}{p_i} \tag{2.24}$$

Relative entropy, also known as the Kullback-Liebler divergence [72], is a generalisation of Shannon's entropy. It provides a measure of the difference between two probability distributions say P and Q, given that they are defined over the same domain. This measure is not symmetric, though it is often intuitively thought of as a distance metric since it is always greater than zero[10], or equal to zero if the two distributions are the same. It can be used as distance metric in distribution fitting algorithms where the objective is to find the set of model parameters Θ that minimise the value of the relative entropy between the empirical distribution P and its approximation model Q_Θ.

Formally, the relative entropy between two discrete distributions P and Q, denoted by R_{PQ} is defined as

$$R_{PQ} = \sum_i p_i \log \frac{p_i}{q_i} \tag{2.25}$$

where p_i, q_i are the probabilities of event i in P and Q, and for continuous distributions as,

$$R_{PQ} = \int_\Omega f(x) \log \frac{f(x)}{g(x)} dx \tag{2.26}$$

[10]It must be noted that the value of relative entropy increases as the distributions diverge.

where $f(x)$, $g(x)$ are the probability density functions for P and Q respectively, and Ω is their domain.

Since $R_{PQ} \neq R_{QP}$, care must be exercised when ordering the two distributions. Usually, as in our work, the first distribution, say P, represents the theoretical distribution and the second distribution, say Q, a model or an approximation of P. The units of relative entropy (same as entropy) depend on the base of the logarithm, e.g. for base 2 we have bits, base e, nat and for base 10 the units are known as both ban or hartley.

2.7 PIPE2

PIPE2 [20, 40] is a Java-based open-source tool for GSPN-based system modelling and analysis. It was developed as a platform-independent Petri Net editor, but was later enhanced by a number of analysis modules and evolved into a distributed performance evaluation environment.

PIPE2 provides a friendly graphical user interface (see Figure 2.13) which allows the user to create, load, save and edit GSPN models that conform to a format similar to the Petri Net Mark-up Language (PNML) [18, 55]. While this unfortunately limits the inter-change of the constructed models between PIPE2 and other tools, PNML support was prohibited since the current PNML standard does not provide support for stochastic variations of Petri Nets such as GSPNs.

In PIPE2, Petri Net models can be drawn on the canvas using available features from the drawing toolbar. The drawing toolbar supports all the components required to construct a GSPN model of arbitrary complexity, namely places, timed and immediate transitions, arcs and tokens.

The model design interface also provides additional visual features, like zoom, export, tabbed editing and animation. The animation mode has two available options. The user can either manually fire enabled transitions or observe the behaviour of its model under random firing of enabled transitions, by selecting the number of firings and the time delay between them.

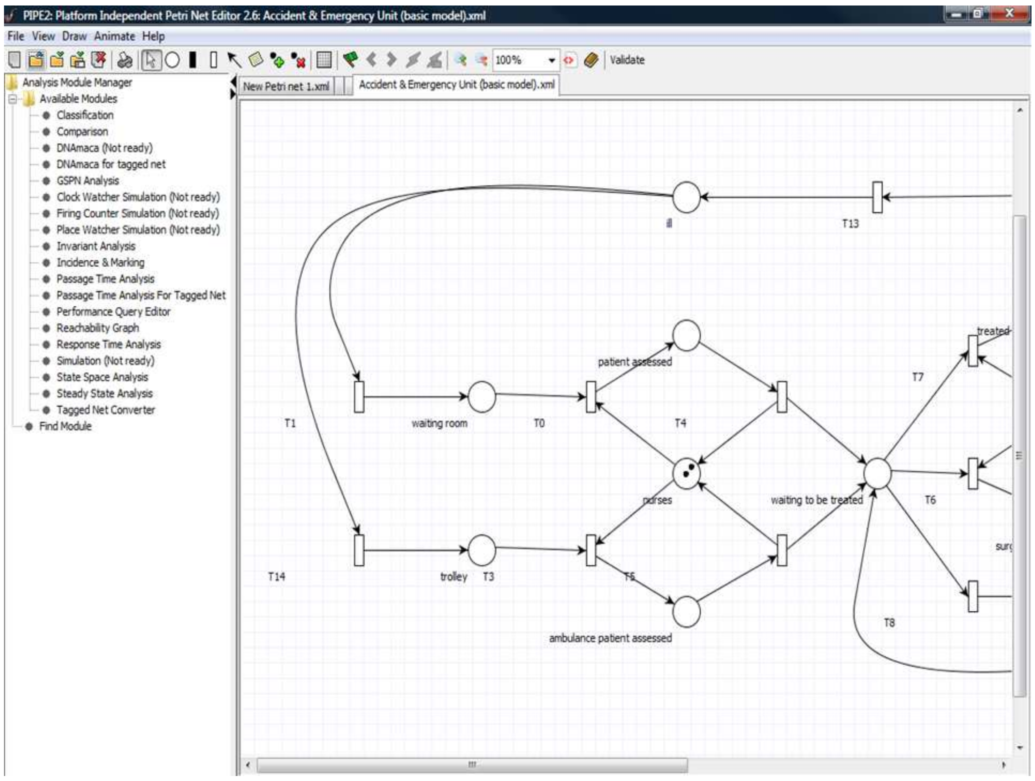


Figure 2.13: PIPE2: Petri Net Model Design Interface.

PIPE2 has been extended to support visualization of Coloured Generalised Stochastic Petri nets (CGSPNs) [31]. PIPE2 also provides a CGSPN unfolding feature [14] that enables the transformation of a CGSPN to an equivalent GSPN model and then uses the existing analysis modules to perform model evaluation.

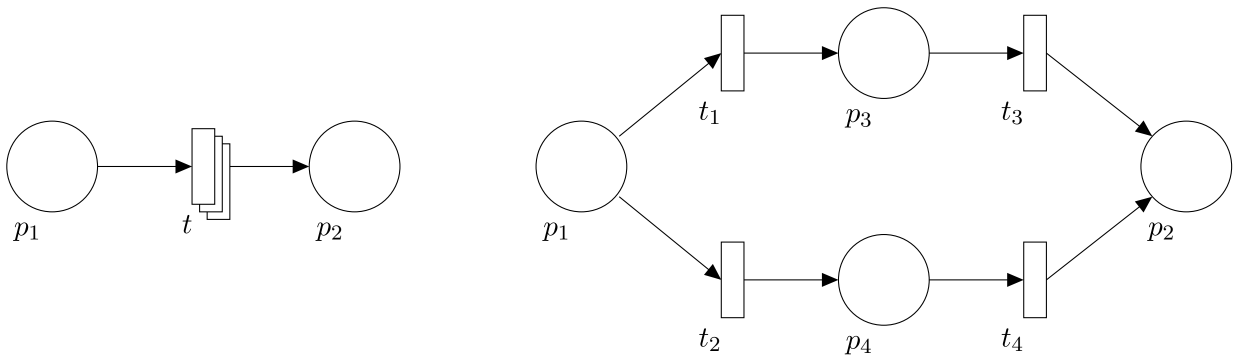


Figure 2.14: An example of a GSPN subnet shown in compact transition representation (left) and in detailed representation (right) in PIPE2.

Another extension has been made to PIPE2 to enable the compact visualisation of large models; specifically, a special form of transition has been introduced which essentially abstracts a Petri Net subnet from a visualisation point of view, i.e. the dynamic behaviour of the model, as well as the delays introduced by the subnet's transitions, do not change. An example of this type of transition is illustrated in Figure 2.14.

2.8 Real Time Location Systems

In this section we provide a brief overview of the established location tracking technologies available. *Real Time Location Systems (RTLSs)* facilitate data collection in real time. This process is done automatically without requiring any human input thus increasing the quantity and accuracy of data.

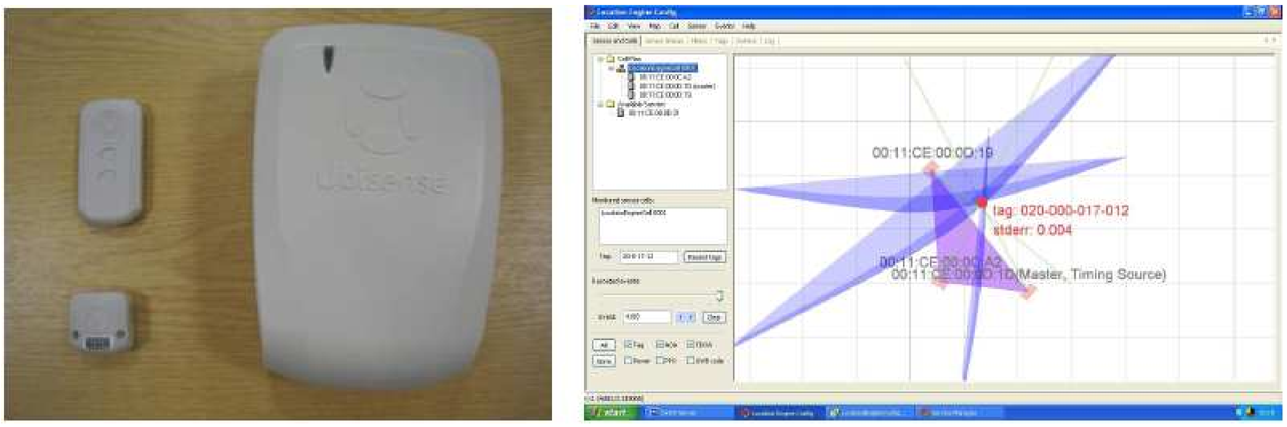


Figure 2.15: UWB-based sensors and tags (left) use a combination of angle-of-arrival and timed-difference-of-arrival triangulation schemes to enable high accuracy tag location tracking (right).

RTLSs have been deployed mostly in the fields of supply chain management and healthcare [9, 101], but their domain of applicability is increasing with time, especially with the realisation of the vision of the Internet of Things [5]. If we assume a generic customer-processing system, the use of RTLSs can provide insights into resource and customer flow. This information can be then used to improve performance efficiency, asset management and system safety.

A variety of location tracking systems is available where each utilises a different technology: Infrared (IR), WiFi, Ultra Wide Band (UWB) and Radio Frequency Identification (RFID). Each technology has merits over the others but the choice of technology to be used depends on several factors such as the cost, the environment it is applied in and the degree of accuracy required.

Infrared technology provides continuous tracking in 2D. Compared to the other technologies it provides the worst level of accuracy, typically seven to twelve metres, and its signal can be blocked by objects placed in the line of sight between the detection devices (tags) and the spotter. Currently, IR is not used as a stand-alone location tracking technology but in combination with RFID [85].

WiFi RTLS has the advantage that it can be deployed in existing WiFi infrastructure since all WiFi devices operate under the IEEE 802.11 standard. Similar to IR, it can also provide continuous monitoring in 2D but its accuracy can reach higher levels depending on the number of access points that are available. Vendors suggest that its maximum accuracy can reach one to two metres but it is in the range of five to ten metres in practice.

Ultra Wide Band technology can provide continuous tracking in 3D with accuracy up to fifteen centimetres. The high accuracy of UWB makes it stand out from the other RTLS technologies. UWB emits short-duration high-bandwidth radio pulses at a low power. This limits the interference of UWB with other signals, thus making it suitable for radio sensitive environments. However, the low pulses limit its range. The high frequency on which UWB operates (6-8.5 GHz), allows the monitoring of multiple tags in an area. UWB-based RTLSs (an example can be seen in Figure 2.15) can be the most expensive systems to deploy, depending on two factors: scalability and accuracy required. Until recently, their domain of applicability was limited to mission critical applications and/or personnel tracking in dangerous environments [86, 44]. Currently, they are also employed – amongst other areas – in automotive and aerospace manufacturing [108].

The most widely applied technology is RFID due to its low cost. RFID tracking consists of three types: passive, battery-assisted passive and active. Passive RFID requires no battery and is the cheapest and most basic option; it provides the tag's location when it is in range from an RFID reader with an accuracy of five to ten metres. Active and battery-assisted passive RFID tags have onboard batteries thus emitting a signal to the RFID reader without the need to be as close as the passive tags. RFID-based location tracking systems are also[11] used in the fields

[11] Additional application fields of RFID-based RTLSs are listed in Section 1.1.

of logistics and asset management. Some examples can be found in [87, 112].

2.9 Related Work

The problems associated with manual construction of performance models of both software and physical systems have been acknowledged during the past two decades by the research community. It is an expensive and time-consuming process that often results in models which fail to accurately capture the relevant time delays associated with the system's underlying processes and thus, fail to emulate correctly the system's behaviour. In this section we present research related, but not limited, to the automated construction of performance models and data mining.

Research conducted in the fields of workflow induction and process mining share some high-level similarities with ours. For example, Agrawal et *al.* present an approach which automatically constructs process models from execution logs [2]. Their algorithm produces a directed acyclic graph (DAG), known also as execution graph, which represents the workflow structure of the underlying business process, by identifying dependencies between activities. In the model graph, these dependencies are represented by directed edges. Gonzalez et *al.* propose a method to construct compressed probabilistic workflow models from RFID-based location tracking data [46]. Their research, which focuses on supply chain management applications, yields an efficient method for extracting and storing item flow information, such as the path of items, their transition probabilities to different locations and duration distribution at each location. The authors manage to achieve a high-level of data compression by grouping individually tagged items into categories and by ignoring or merging deviations of individual items within the same category. Item flows were stored in a data cube, referred to as flowcube, whose measure is a flowgraph[12] [45]. These flowgraphs differ from traditional flowgraphs; their nodes are annotated with a distribution of possible durations at the particular node and they maintain a set of exceptions to the transition probabilities and duration distribution at each

[12] A traditional flowgraph is a tree where each node represents a location and each edge represents a transition between two nodes.

node. These exceptions – recorded only when sufficient evidence to support them exists – are essentially deviations of paths or duration, which significantly affect the transition probabilities and duration distribution respectively. In our work we employ techniques which share some similarities with [45] to extract high-level information from raw location tracking data, i.e. service duration of a customer at a service area, and to record or merge path deviations of customers of a particular class from the general path of the same class.

Techniques, similar to those of process mining, are also applied in software architecture-level performance model extraction of component-based systems [27]. Event records such as BEGIN and END are used to signal the entry or exit from a software building block and during the software run-time they are stored in a list. This list can be processed to yield the components and component connections (control flow) of the system. We employ a similar approach to infer the high-level customer flow – in terms of service area locations – from past customer location traces. Since our research is based on location tracking data we use the low-level location of customers, specified by x and y coordinates, and the computed boundary of each service area to determine high-level events, such as a customer's arrival or departure from a service area.

A previous research endeavour in the field of automatic construction of GSPN models has been performed by Xue et *al.* [114]. They developed a methodology that automatically constructs GSPN models for Flexible Manufacturing Systems (FMSs)[13], which can respond to changes in their environment in real-time. The authors' work includes a software package, FMSPet, where an input language called FMSDL is used to describe the physical system being modelled. The output of FMSPet is compatible with the Stochastic Petri Net Package (SPNP) [34] which is used to analyse GSPNs. However, their methodology does not extract the model from data; the FMS must be explicitly defined in the input file, and it is application-specific. Thus, it cannot be easily adapted for other scenarios.

Another piece of work related to the automatic construction of performance models, but in a different context, was performed by Kounev et *al.* [71]. They propose an approach to auto-

[13]An FMS consists of two components: the actual manufacturing system, i.e. an assembly line, and a controller which allows the system to react to various changes.

matically extract the performance model of an Enterprise Data Fabric (EDF)[14]. The model is realised using Queueing Petri Nets (QPNs) and its parameterisation is based on readily available monitoring data which are provided by the EDF. Three submodels, the Client, Server and Network submodel, are combined to produce the EDF's performance model for any number of servers and clients (multiple instances of the Client and Server submodels may be used). The authors' approach is implemented as part of the simulation-based tool Jewel whose purpose is the automated performance prediction and capacity planning for EDFs. The applicability of their modelling approach, performance prediction and capacity planning are demonstrated through a case study using a representative EDF.

Our work is most closely related to the work of Horng et *al.* which examined the aspects of automatic Queueing Network model construction of physical systems from location tracking data [57]. In this work the authors propose a multiple-stage processing pipeline in order to infer an accurate performance model. The model encapsulates both the structure of the network through routing probabilities and the service time of its service centres. The inter-arrival time distribution of customers at service centres is also inferred. In addition to the use of different modelling formalisms, there is another major difference with our work: the authors tagged the servers in the system, i.e. their location is known and they assume that each server has a user-defined service area (circular). We make the same assumption regarding the shape of the service area, but we are able to automatically infer both the servers' location and service radius from the location tracking data. Furthermore, unlike our work, the existence of presence-based synchronisation between service centres and the presence of service cycles is not inferred.

[14]An EDF is a distributed enterprise middleware, located between the application and the host network, and it is used to allocate and manage data and resources across multiple, physically separate, hardware nodes.

Chapter 3

Generating Synthetic Location Tracking Data

Simulation has been widely used in both academic and business applications since it provides the user with the ability to rapidly perform experiments so as to investigate the system behaviour under different scenarios. Furthermore, this process is quite inexpensive, not only in terms of money but resources too, compared with the actual cost for implementation, or modification, of a real life system. Applications where simulation is of particular interest include performance modelling and evaluation of systems, workflow analysis, process improvement, asset/personnel management, etc. The availability of simulation software tools, such as MOMOSE [22], Asset-Manager NT and PT [29], WITNESS [76] and Simul8 [97], assist organisations to identify their system's bottlenecks and gain better insights into the implications of various resource and work-load modifications on the overall system performance. However, some simulation tools have limited application domains (e.g. [22, 29]) and many of them also fail to capture the stochastic nature of the system's inherent time delays.

In this chapter we present LocTrackJINQS [56], originally developed to support location-based research [6, 57], an open-source simulation library used to construct location-aware simulations. LocTrackJINQS is an extension of JINQS, a Java simulation library for multi-class Queueing Networks [43]. JINQS provides a suite of primitives which enable developers to build simulations for a wide range of stochastic Queueing Network models in a time efficient manner. Furthermore, it offers simplicity for simulation construction and flexibility for application-specific functionalities through the use of inheritance [43].

Unfortunately, JINQS only allows the creation of high-level simulations of abstract Queueing Networks and fails to provide support for realistic low-level features found in the physical world. This makes JINQS unsuitable for constructing simulations that can approximate entities' physical movements in a real life system, since entities' travelling time might significantly influence the overall system response time. In order to overcome the latter limitation, LocTrackJINQS provides low-level models of entity movement, while retaining the abstract high-level model specification power of JINQS. LocTrackJINQS also provides primitives for generating synthetic location tracking data similar to that collected from actual real time location tracking systems. This eliminates the need for heavy upfront investment and long-running observation periods that an RTLS installation requires, and thus benefits research in data mining from location tracking data and industry by providing a test bed for the development of location-based applications.

The remainder of this chapter presents LocTrackJINQS in more detail. We first give an overview for the simulator and its capabilities. We then present its basic software architecture and explain how the location-awareness related features are implemented. The chapter concludes with a presentation of two case studies which demonstrate the construction of a simulation for a real life system and evaluate LocTrackJINQS's operation.

3.1 LocTrackJINQS

LocTrackJINQS is a simulation library that offers functionalities to setup and execute simulations of real life customer-processing systems as Queueing Networks with low-level location information (see Figure 3.1). It provides the user with the ability to specify the high-level features of the network, i.e. the customer flow structure and time delay distributions (e.g. service time and inter-arrival time), and the low-level ones, i.e. the entities' geographic locations, their

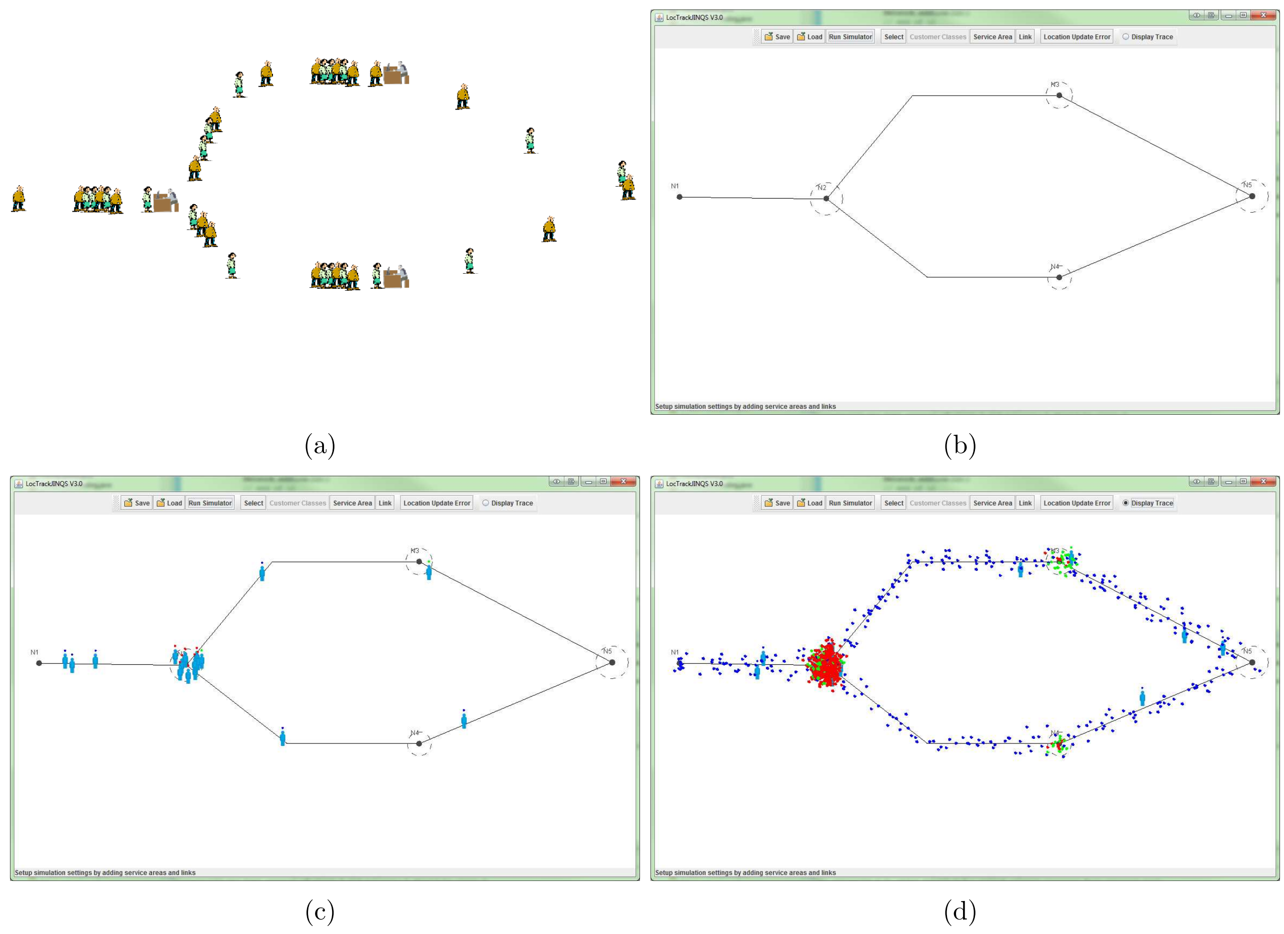


Figure 3.1: An example of simulating a real life system using LocTrackJINQS: Figures 3.1(a) and 3.1(b) demonstrate how a customer processing system is represented as a high-level Queueing Network with low-level location information; Figures 3.1(c) and 3.1(d) show a screen shot of the simulation in progress and the generated location traces respectively.

moving speeds and paths.

A simulation can be specified and executed through LocTrackJINQS's GUI or programmatically. Simulations can also be saved and loaded for future execution or modification, e.g. addition of new service areas, customer paths, etc. Figure 3.2 presents the main features supported by LocTrackJINQS, grouped into categories. Multiple N-Servers[1] with common queue can be used to simulate customer-processing systems which employ multiple service points – each with its own service area – in order to service a shared queue of customers, e.g. the cashiers in a bank or post office counters.

Service preemption allows a progressing service, provided by a service area, to be paused when certain conditions are met, i.e. a customer, who has higher priority than the current customer

[1] N can be set equal to one, thus enabling the use of multiple single-servers which share a common queue.

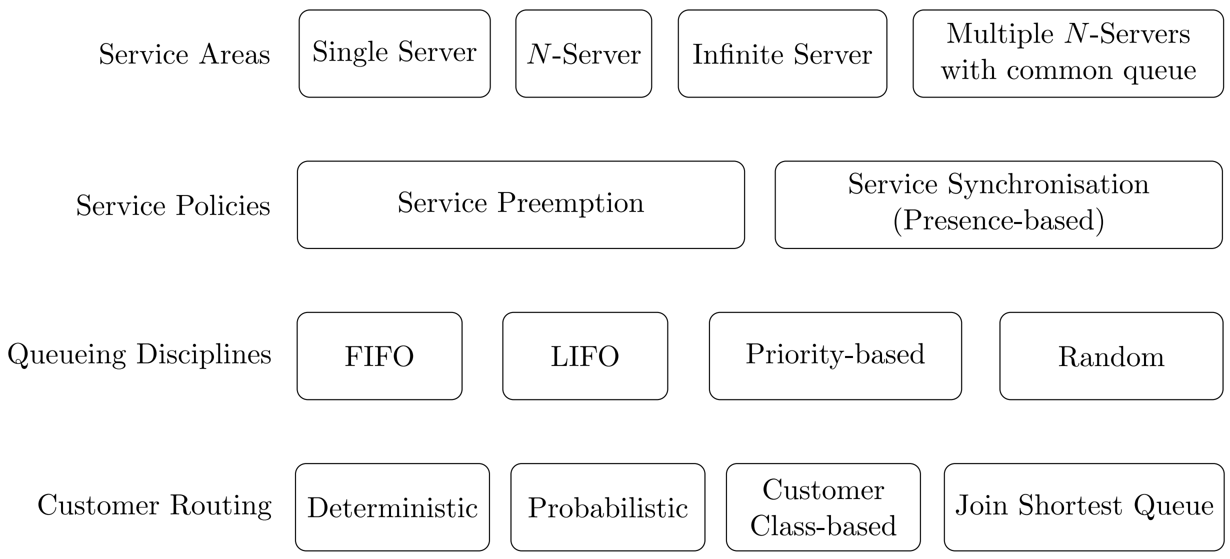


Figure 3.2: Main features supported by LocTrackJINQS.

being served, arrives at the service area and requests service, or simply whenever a new customer arrival occurs. Whenever the service of a customer is paused, it is resumed as soon as the service which caused the interruption has been completed. The service synchronisation feature is application-specific and it was developed to make the validation of the presence-based synchronisation detection mechanism possible (see Chapter 5). This feature allows the user to specify one or more conditions which associate the service status of customers at a particular service area with the presence of customers in other service areas[2]. Service, provided to customers by the server of the synchronised service area, initiates and progresses only if all the specified synchronisation conditions are met. We note that in service synchronisation, when service is paused, the server of the synchronised service area remains unavailable to all customers in the queue until the paused service is resumed and completed.

Figure 3.3 shows all currently supported distributions which can be used to specify the location update error, customer speed, service times and customer arrivals. Other distributions can be easily implemented and incorporated in the simulation via the use of inheritance.

In addition to the simulation specification features presented above, LocTrackJINQS provides customer and system-oriented performance measures for each service area defined in

[2]A synchronisation condition is defined with respect to the number of customers, of each class, that must be present in the synchronising service area so that the service of a customer in the synchronised service area can initiate and progress.

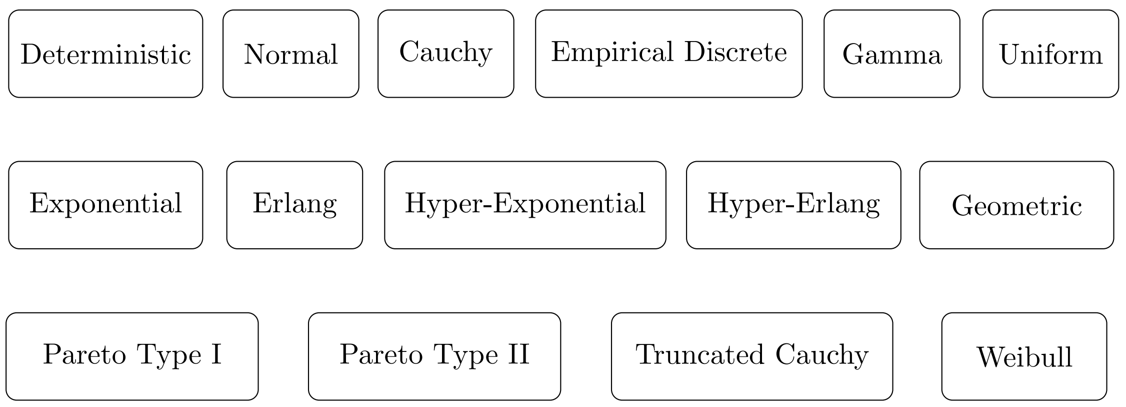


Figure 3.3: Distributions supported by LocTrackJINQS.

the simulation [43]. Examples of such performance metrics include the mean response time and its variance (customer-oriented), mean queue length, server utilisation and mean population (system-oriented). The computation of such metrics is enabled by the use of two types of measurement variables contained within each node defined in the network; their computed values can be displayed at the end of the simulation.

Before we proceed to examine the software architecture of LocTrackJINQS, we outline the three main features which distinguish it from JINQS:

1. Location-aware simulation support. As mentioned earlier, JINQS only provides the user with the ability to specify high-level simulations, where service areas are defined as servers with an associated queue and no geographical location. Furthermore, as no low-level information is incorporated, customer location and flow are defined only with respect to which server the customer is currently located, and which one the customer is going to visit when its current service request is completed. Transitions of customers between servers are assumed to occur instantaneously. In LocTrackJINQS, each entity in the Queueing Network is assigned a geographical location[3] (see Section 3.2.1) and its movements occur along user-defined paths at a speed sampled from a user-specified distribution.

2. Location update generation. The original motivation and purpose of LocTrackJINQS is to support location-based research, e.g. mining agent flow patterns and performance models from location tracking data. It can generate synthetic, yet reasonably realistic,

[3]The geographical location is defined under a 2D Cartesian coordinate system.

location traces on the fly, as the simulation progresses (see Sections 3.2.3 and 3.2.4). Furthermore, since the user is able to specify the location update error, LocTrackJINQS can produce location tracking data that virtually approximate any type of RTLS.

3. Graphical User Interface. LocTrackJINQS provides a graphical user interface that enables the setup of the simulation environment, along with all required parameters[4], in an intuitive and user-friendly manner. Also, it allows the user to monitor the progressing simulation visually. Further details regarding the GUI's design can be seen Section 3.2.5. An introductory user's manual is also included in Appendix A.1

3.2 Software Architecture

LocTrackJINQS has been designed in a way to be either used as a stand-alone simulation tool, or as a Java library, thus enabling users to incorporate it in their own applications; for this purpose and also, to add new features to LocTrackJINQS, understanding its software architecture is important. The two main packages which enable the construction and execution of simulations programmatically are **network** and **tools** [43]. The classes which correspond to entities used to formulate the structure of the Queueing Network model are located within **network** while **tools** contains various utility classes. These are used to define various properties of the network's entities (such as their service time distributions), facilitate the simulation's event scheduling and calculate performance metrics. A third package, namely **gui**, contains classes which deploy the graphical user interface of LocTrackJINQS and allows it to be used as a stand-alone simulation tool.

The design of LocTrackJINQS, when considered as a stand-alone tool, follows the Model View Controller design pattern (see Figure 3.4) [100]. Its components are defined by the following classes:

- Model: It represents the overall state and data of the application domain. In LocTrack-JINQS, the `Network` class in the **network** package represents the model and updates

[4]Simulation setup through the GUI supports only the specification of non-application-specific features.

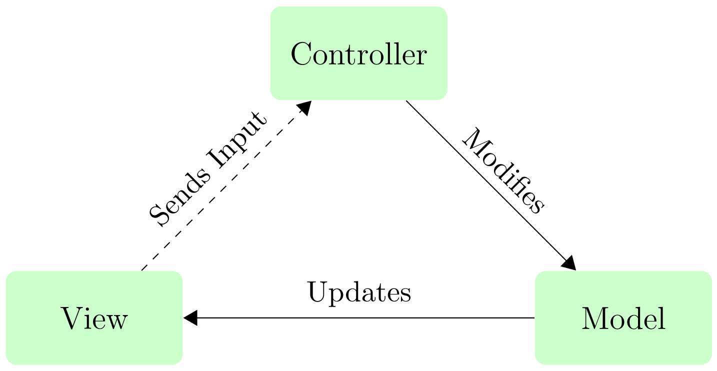


Figure 3.4: Model View Controller design pattern for LocTrackJINQS.

the view when changes occur in its state. View updates are issued either directly from the `Network` class, e.g. the addition of a service area, or indirectly from its components, e.g. the movement of customers.

- View: It displays the model and handles communication with the user. The model is rendered so that users can visualise it and provides them with the ability to interact with the application. In LocTrackJINQS, classes `ControlPanel` and `DrawingPanel`, contained within the **gui** package, act as views. In particular, the `ControlPanel` class allows the user to interact with the application, i.e. setup the simulation environment and location update error. The `DrawingPanel` class renders the model and enables users to visualise the simulation while it progresses. Furthermore, before the simulation is initiated, it allows low-level customisation of the paths which define the flow of customer in the model.

- Controller: It receives user input from view and responds by invoking the appropriate methods on the model. The `ControlPanel` class also acts as the controller in LocTrack-JINQS. It contains several `swing`[5] components which are used to accept user input and depending on the type of input obtained, it issues calls to the corresponding methods of the `Network` class.

[5]`swing` is a core Java package which contains important classes for adding a GUI to an application.

3.2.1 Architectural Modifications and Additions of Queueing Network Components

During the process of extending JINQS to support location-awareness, several structural modifications were required to be made, especially in terms of the way a Queueing Network is defined. In general, a Queueing Network is defined as a collection of nodes (service centres) with an associated queue and service policy (see Section 2.3). Furthermore, when multiple nodes exist in a Queueing Network, an associated set of routing rules, which specifies the flow of customers, must be provided.

JINQS uses three classes, along with their specialisations, to represent a Queueing Network: `Node`, `Link` and `Customer`. `Link`s are used to connect `Node`s and enable the (instantaneous) transportation of `Customer`s. `Node` class specialisations, i.e. `Source` and `Sink`, are used to inject and retract `Customer`s in the network respectively. `Customer`s move along `Link`s to various `Node`s within the network and request service or resources. The UML representation of the Queueing Network structure is shown in Figure 3.5.

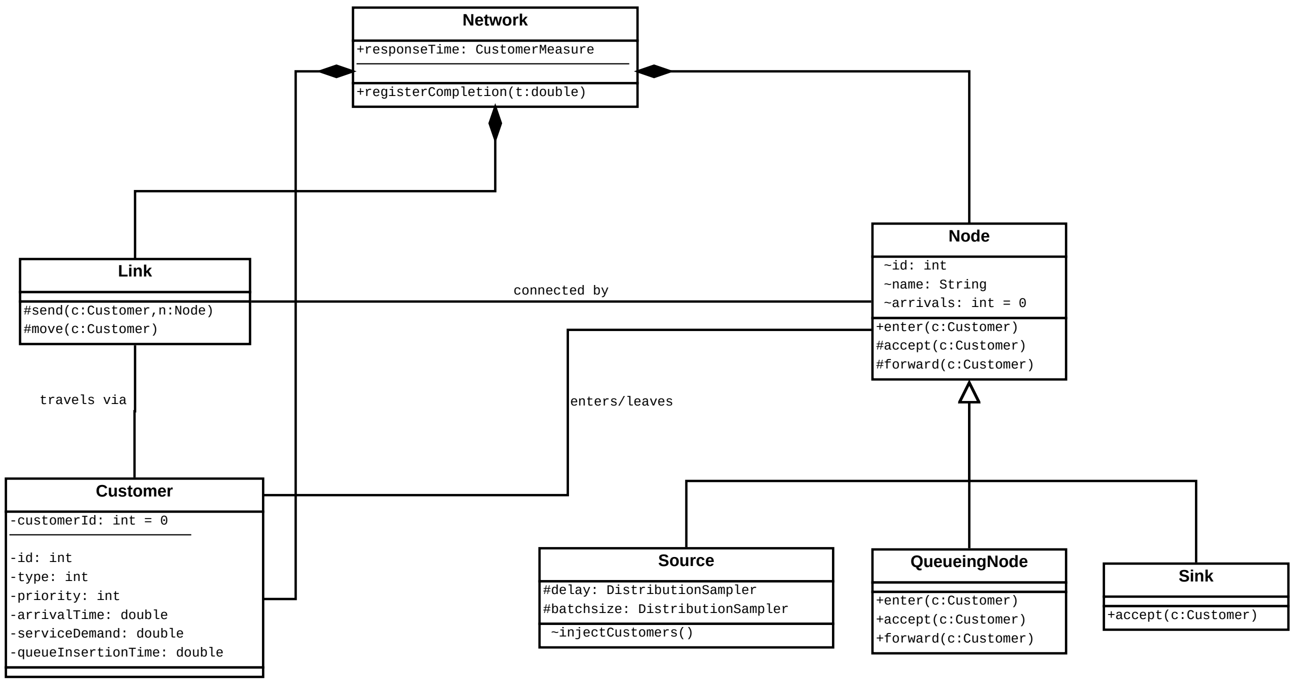


Figure 3.5: The main classes in JINQS in UML representation.

The main challenge we encountered during the design phase of LocTrackJINQS, was to incorporate low-level, spatial information[6] regarding customer movement and node location, while maintaining a class hierarchy that can be easily extendable at the same time. To achieve a suitable level of abstraction, similar to that of JINQS, we define the `INode` interface. This interface extends the notion of `Node` to include any (planar) region a `Customer` instance may enter, and remain for a period of time[7] before departing for its next destination [56]. `INode` is implemented by the classes `Server`, `MultiQueueingServers` and `QueueingArea`. An instance of the `Server` class does not provide "service" to customers; after a `Customer` entity is accepted to the `Server`'s service area, it is immediately forwarded to its next destination which is defined by the outgoing link(s). The `MultiQueueingServers` class implements a cluster of `Server`s (or subclasses of `Server`) which share a common queue (an instance of the `QueueingArea` class); this class can also be used to model multiple N-Servers.

The classes `Source` and `Sink` are subclasses of the `Server` class and provide the same functionality as they do in JINQS: a `Source` instance injects `Customer`s in the network and a `Sink` instance retracts them. Two more basic types of service points are provided by LocTrackJINQS which are subclasses of `Server`, namely the `InfiniteServer` and `QueueingServer` classes. Instances of the `InfiniteServer` class provide immediate service, i.e. no waiting time, to accepted customers as they have an infinite number of resources (servers). On the contrary, an instance of `QueueingServer` class has a limited, user-specified, number of servers and it can be used to model an N-Server. If no servers are idle when a `Customer` entity is accepted, the `Customer` is queued and waits until a server becomes available. The service time for each serviced `Customer` is sampled from a user-specified distribution. All types of service points that are currently supported by LocTrackJINQS have a fixed location and a circular service area. The radius of each service area is user-specified and is used – in combination with the `Customer` entity's location – as the criterion to determine whether a customer is accepted into a service point's service area and thus, request service from it.

Furthermore, application-specific types of service points can be easily implemented using inheritance. Examples of such types of service points include the `SynchronisedQueueingServer`,

[6]Specified as 2D Cartesian coordinates.

[7]In a CTMC analogy, this period of time corresponds to the sojourn time of the customer at a certain node (state).

`PreemtiveGenericGenericServer` and `SynchronisingQueueingServer` classes, all of which are subclasses of the `QueueingServer` class and are included in the current version of LocTrackJINQS. Figure 3.6 depicts the basic Queueing Network structure, as well as all readily available types of service points along with their class hierarchy, in UML representation.

Using the `INode` interface we have successfully managed to embed low-level information, such as location and service radius, in the various types of service points supported in LocTrackJINQS. A similar approach is employed to enable the simulation of the low-level movement of `Customer` entities between different locations in the system. Instead of using the `Link` class as the superclass to define the various types of links, which is the case in JINQS, LocTrackJINQS uses an abstract class, namely the `AbstractLink`.

The introduction of the `AbstractLink` class enables a well defined link hierarchy: types of links can be differentiated according to the customer routing policy they employ (see Figure 3.7). The `Link` class, which is a subclass of `AbsrtactLink`, maintains the same functionality as the `Link` class in JINQS, i.e. forwards `Customer` entities to their destination in zero time. However, in LocTrackJINQS, two subclasses of `Link`, namely `PhysicalLink` and `TransportLink`, are used to represent the physical path – defined as line segments connected together using break points – that a `Customer` entity follows when moving between two `INode` implementations. Using the method `moveCustomers`, a `TransportLink` updates the location of the `Customer`s traversing the link by taking into consideration their individual speed and direction of travel. The `MultiLinks` class (also a subclass of `Link`) is composed of one or more `TransportLink` instances and is mainly used within the `MultiQueueingServers` class. Its main application is to transport `Customer` entities from the shared `Queueing Area` to the individual service points which consist a `MultiQueueingServers` instance. It also uses the `findAvailablePath` function which selects one of its `TransportLink`s to forward a `Customer`; this selection depends on service point availability, i.e. which service point is not engaged in serving another `Customer`.

The `RoutingLink` class (also a subclass of `AbstractLink`) can be used as the parent class for implementing various customer routing policies. Examples of such policies which are currently included in LocTrackJINQS are:

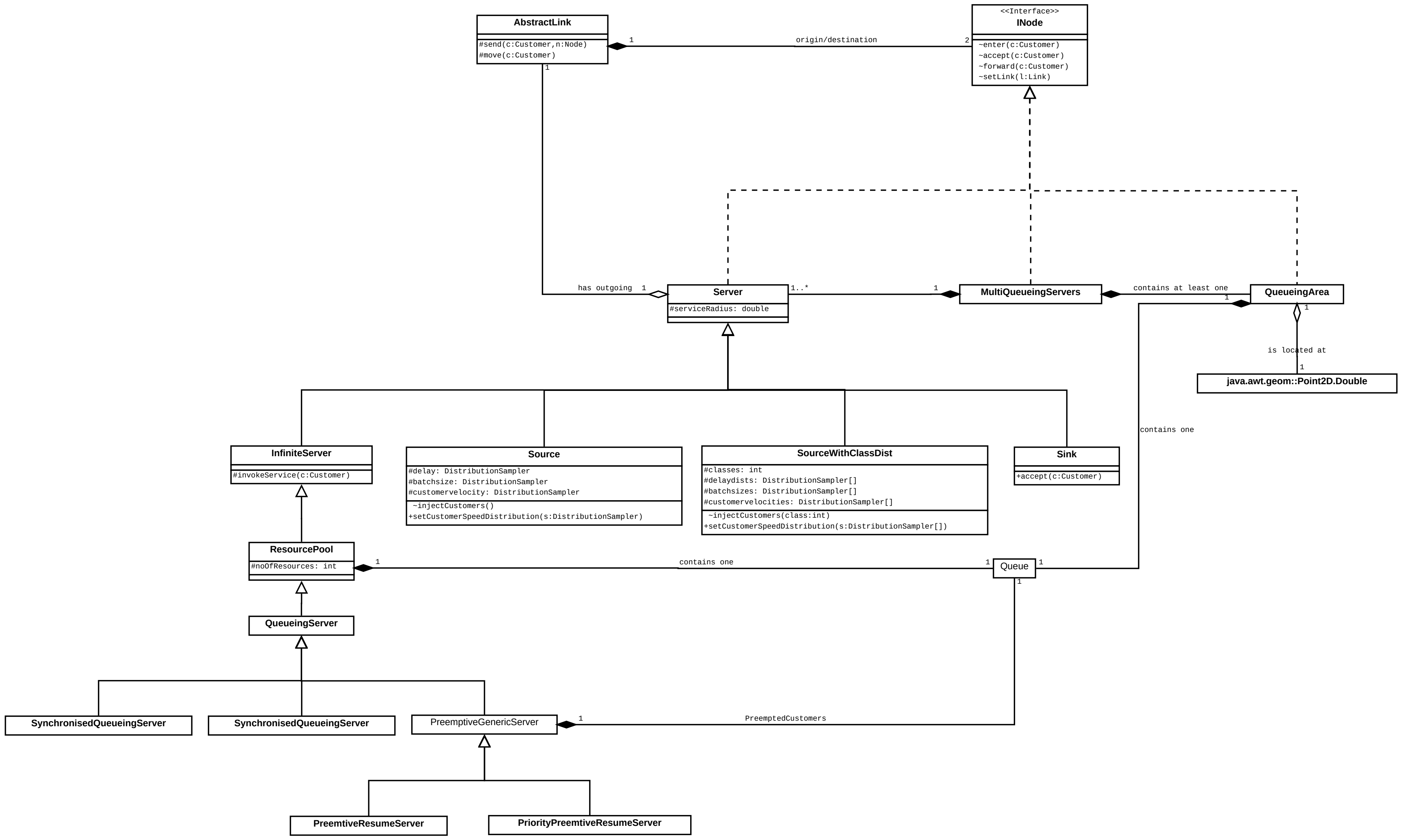


Figure 3.6: The basic Queueing Network structure and supported types of service points of LocTrackJINQS in UML representation.

- Customer class-based routing. This routing policy allows `Customer` entities to be routed from one `INode` to another, where the destination `INode` is selected from a user-defined set. The destination selection is performed by applying a user-defined rule, which maps the class of a `Customer` entity to a member of the latter set. This policy is implemented by the `ClassBasedRouting` class.

- Probabilistic routing. This routing policy does not consider any individual properties of a `Customer` entity. As in class-based routing, a set of destination `INode`s must be defined; however, the destination selection is purely probabilistic. The probability associated with each destination must also be provided by the user. This routing policy is implemented by the `ProbabilisticRouting` class.

- Join shortest queue routing. This routing policy allows `Customer` entities, given a set of destination `INode`s, to be routed to the destination which, at that time instance, has the shortest queue. This is particularly useful when modelling a real life system, where customers wish to obtain some service that is provided by several service points in the system, each with its own queue, e.g. passport control checkpoints at an airport. This policy is implemented by the `JoinShortestQueueRouting` class.

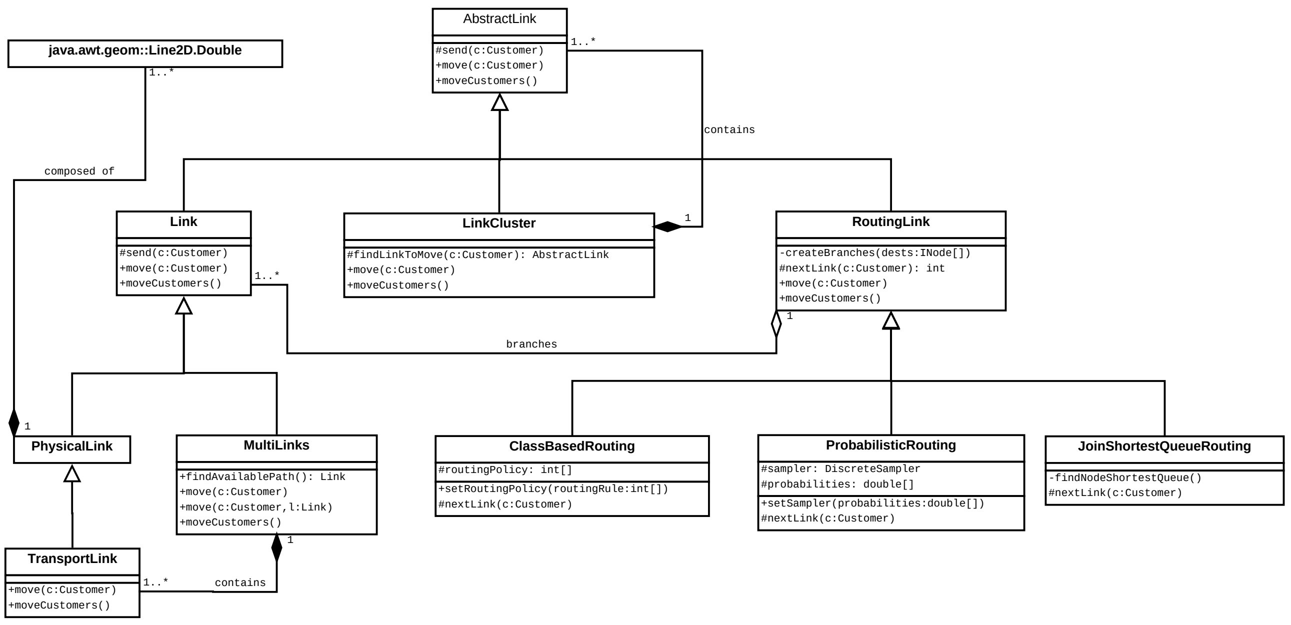
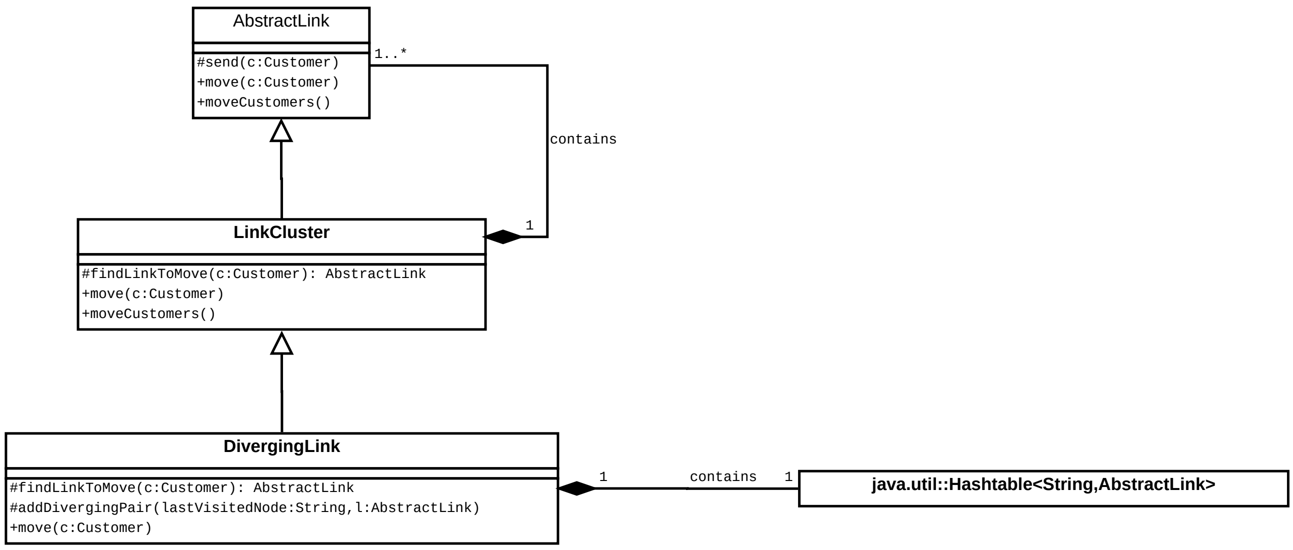


Figure 3.7: Hierarchy of classes used to implement different types of links in LocTrackJINQS (UML representation).

The `LinkCluster` and `MultiLinks` classes may appear similar in the sense that they consist of multiple links; yet they have significant differences in terms of structure, operation and functionality. In `Multilinks`, links, in particular `TransportLink` instances, are constructed internally when a new `Server` node is added to the corresponding `MultiQueueingServers` instance. These internal links are not considered as independent components of the Queueing Network. On the contrary, `LinkCluster` provides an abstract structure used to group links which are explicitly defined within the network. `LinkCluster` uses a set of user-defined rules to manage `Customer` forwarding via its member links. Subclasses of `LinkCluster`,



AbstractLink
#send(c:Customer)
+move(c:Customer)
+moveCustomers()
1..*
contains
1
LinkCluster
#findLinkToMove(c:Customer): AbstractLink
+move(c:Customer)
+moveCustomers()
DivergingLink
#findLinkToMove(c:Customer): AbstractLink
#addDivergingPair(lastVisitedNode:String,l:AbstractLink)
+move(c:Customer)
1
contains
1
java.util::Hashtable<String,AbstractLink>

Figure 3.8: UML representation of the `DivergingLink` class in LocTrackJINQS.

e.g. `DivergingLink` (see Figure 3.8), can be used in conjunction with certain implementations of the `INode` interface, e.g. `ConvergenceNode` (see Figure 3.9), to enable the connection of multiple links to and from a node. The `ConvergenceNode` acts as a virtual layer on top of a `Server` instance in order to allow multiple connections with links. Furthermore, using the `DivergingLink` class, `Customer` entities accepted by a `ConvergenceNode` instance can be forwarded – by links which are members of the group – to specific destinations. The destination is determined by considering the node where the `Customer` was located prior to its arrival at the `ConvergenceNode`.

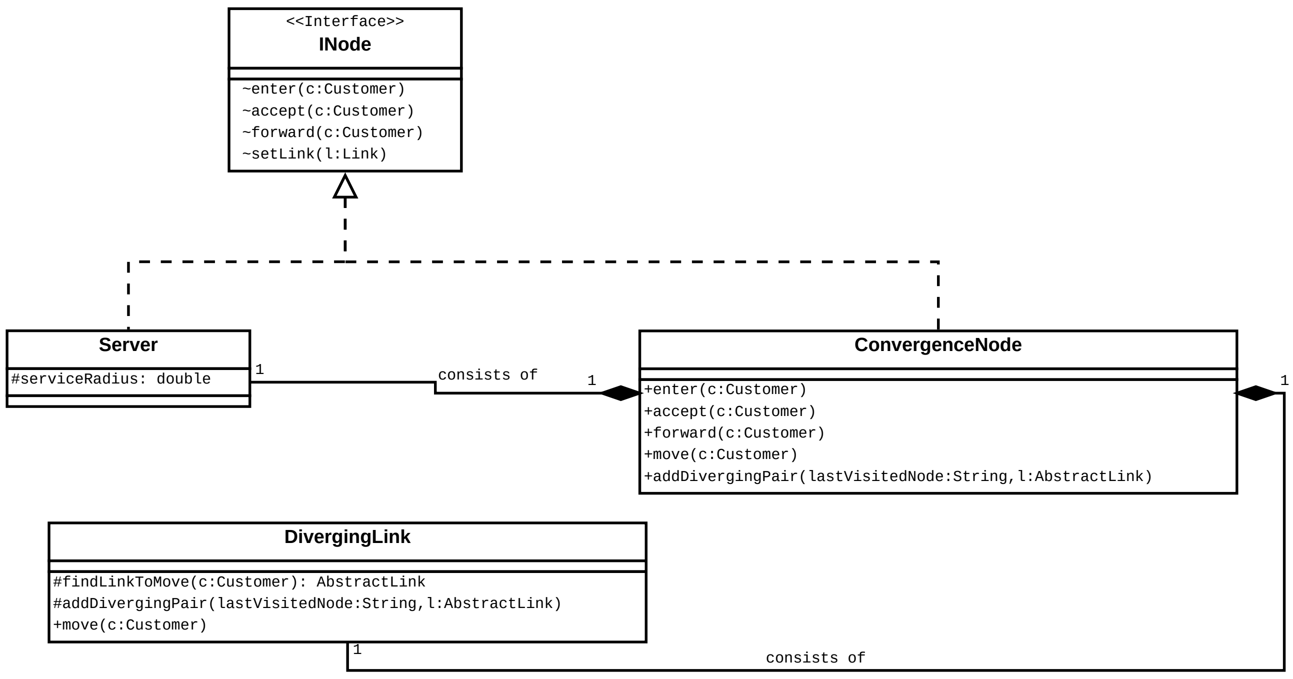


Figure 3.9: UML representation of the `ConvergenceNode` class in LocTrackJINQS.

3.2.2 Saving and Loading Queueing Networks

Another feature introduced by LocTrackJINQS is the ability to save and load Queueing Networks. The need for this feature was induced by the introduction of the GUI. When the simulation environment is defined programmatically, it is naturally stored within a file placed on the machine's hard drive. On the other hand, when the user specifies the Queueing Network through the GUI, the simulation model, including all its associated parameters, is temporarily stored on the machine's physical memory. Thus, when the application is terminated, all effort put by the user to perform this, sometimes tedious, task is lost. In order to overcome this, and allow users to modify and/or execute previously specified simulations, we include a save/load feature into LocTrackJINQS.

Before developing this feature, we searched the existing literature for a unified Queueing Network model specification format, similar to PNML for Petri Nets. The work of Smith and Williams [99] proposes the Performance Model Interchange Format (PMIF) as a standard for performance model interchange between performance modelling tools. The authors perform a survey of representative Queueing Network modelling tools and define a prototype PMIF meta-model[8] based on the EIA/CDIF (Electronic Industries Association/CASE Data Interchange Format). This meta-model is defined by taking into consideration the data required by the tools included in the authors' survey. The latest version of PMIF, i.e. PMIF 2, developed by Smith et. *al.* [98], implements – amongst other features – a revised meta-model and XML schema which provides additional syntactic validations, as well as a set of validations that confirm that the models are semantically correct. Unfortunately, PMIF 2 currently does not incorporate any information regarding the graphical representation of the Queueing Network model, e.g. node coordinates and low-level link customisation.

The main contribution of LocTrackJINQS is the ability to generate location tracking data and therefore, low-level location information is required to be included in the Queueing Network's meta-model definition. Thus, PMIF 2 is not yet[9] suitable for our purpose; instead, we define the LTJ file format, using the Extensible Markup Language (XML), and specify how each network component, along with its parameters, must be represented. Each (abstract) superclass or interface which is used to define a family of Queueing Network components, i.e. `AbstractLink`, `INode` and `Queue`, contains an export function. This function is overridden by the subclasses or implementations of the latter and when called, it returns the XML node representation of that particular instance. A similar hierarchical approach is employed to enable the representation of all distributions supported by LocTrackJINQS as XML nodes. Figure 3.10 shows an example of the representation of an instance of the `QueueingServer` and `TransportLink` classes as XML nodes, as defined for the LTJ file format. The XML node representation for all elements of the Queueing Network can be seen in Appendix A.2.

Saving

The `NetworkSaver` class, located within the **network** package, provides the `saveNetwork` method. This method iterates through all `Network` components, i.e. nodes and links and for each component, it invokes the component's export method. The returned XML node of each instance is stored in the LTJ file – specified as the argument of the `saveNetwork` method. In addition to the network's components, the number of customer classes and the specified

[8]The meta-model is the model of the information required to construct a model.

[9]Location information regarding the model's graphical representation are being considered as future work in PMIF 2.

error/noise distribution (its type and parameters) are also stored within this file.

Loading

The method `loadNetwork` is a member of the `NetworkLoader` class (also contained in the **network** package) and it takes as argument a `File` object that conforms to the LTJ file format described above. This method is responsible for setting up the simulation environment according to the specifications and components contained in the input file. It first initialises the `Network` and `NetworkMonitor` classes, and then parses the input file to create a `Document` object. Several auxiliary methods – one for each type of XML node – read the constructed document and add the corresponding elements into the `Network`; these elements are subsequently displayed onto the `DrawingPanel`.

```
<node id = "N3">
<tagType>Server</tagType>
<tagged>true</tagged>
<updateRate>6</updateRate>
<location>
<Point2D id = "pt1">
<x>5.0</x>
<y>5.0</y>
</Point2D>
</location>
<type>Queueing Server</type>
<serviceRadius>0.5</serviceRadius>
<numberOfServers>1</numberOfServers>
<serviceTimeDistribution>
<distribution>
<type>Exponential</type>
<parameter name = "Rate">0.5</parameter>
</distribution>
</serviceTimeDistribution>
<interarrivalTimeDistribution>
</interarrivalTimeDistribution>
<velocityDistribution>
</velocityDistribution>
<queueingDiscipline>
<type>FIFO</type>
<capacity>2147483647</capacity>
<supportedCustomerClasses><
    /supportedCustomerClasses>
<subQueueDiscipline></subQueueDiscipline>
</queueingDiscipline>
<priorityPolicy></PriorityPolicy>
</node>
```

```
<link id ="N4 to N2">
<type>Single Link</type>
<owner>N4</owner>
<target id = "1">N2</target>
</probabilities>
</customerClassDestinations>
<pathsAndPoints>
<path id ="N4 to N2">
<Point2D id = "pt0">
<x>15.0</x>
<y>5.0</y>
</Point2D>
<Point2D id = "pt1">
<x>20.0</x>
<y>10.0</y>
</Point2D
></path>
</pathAndPoints>
</link>
```

Figure 3.10: XML node representation of an instance of the `QueueingServer` class (left) and `TransportLink` class (right).

3.2.3 Simulating the Monitoring Behaviour of an RTLS

Real life customer-processing systems are usually monitored by an RTLS in order to help system administrators gain insights regarding the system's customer and/or resource flow. An RTLS collects low-level data which can then be processed to provide high-level information used to assess the performance of the underlying system. In particular, an RTLS monitors tags which are attached to, or in some cases embedded in, entities which either provide or request service. Each tag is assigned a QoS (Quality of Service) value which determines the tag's update rate requirement, i.e. how often the RTLS should harvest a location update from that tag. At each location update time slot, the RTLS chooses a tag and retrieves its location. The tag selection is performed in such a way so that each tag's QoS requirement is approximately satisfied, while preventing starvation, i.e. a tag's location is not updated for a long period of time.

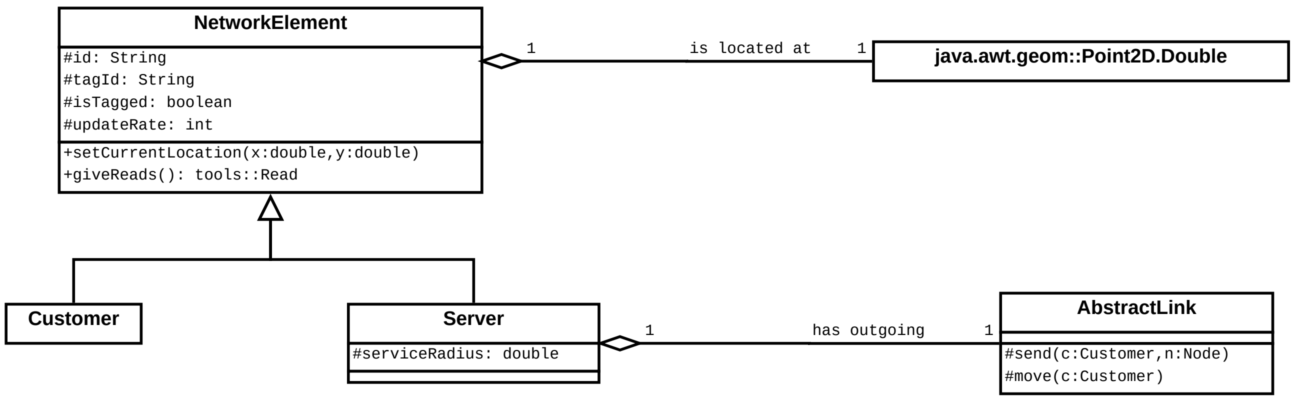


Figure 3.11: UML class diagram of the `NetworkElememt` and its two subclasses `Server` and `Customer`.

LocTrackJINQS introduces the `NetworkElement` class to represent a "tagged" entity in the system. `NetworkElement` equips its subclasses, i.e. `Server` and `Customer`, with attributes and methods which facilitate the simulation of the monitoring behaviour of the RTLS: the `updateRate` attribute defines the QoS requirement of the entity and the `giveReads` method provides the entity's (current) location update.

The monitoring behaviour of the RTLS is implemented by the `NetworkMonitor` class. This class keeps track of all entities being monitored, and at regular time intervals it selects a tag and outputs the tag's location update. The tag selection algorithm (implemented by T.-C. Horng) tries to emulate the one of an actual RTLS; it probabilistically selects a tag from a list of candidates. The list of candidate tags to be updated is constructed by taking into consideration the QoS requirement of each tag, as well as the time which has passed from the tag's last location update (thus preventing starvation for a particular tag).

RTLSs employ different technologies, e.g. Ultra Wide Band, RFID and Wi-Fi, each with a different level of noise associated with its location measurements (see Section 2.8). Additionally, the quality of the location readings obtained by an RTLS also depends on several other factors which relate to the environment the RTLS is being deployed in: its nature, e.g. radio sensitive environments may contain devices which cause signal interference, its topology, e.g. existence of objects that may obstruct the sensors' signal, and the density of the sensors in range. It is thus very difficult, and also impractical, to assume any noise distributions. Therefore, LocTrack-JINQS allows the user to specify an appropriate error distribution; this enables the generation of location tracking traces tailored to approximate those produced by any RTLS technology.

The generated location updates are essentially tuples of the form `(tagName, type, time, x, y, stderr)`. The `tagName` field denotes the unique identifier for each entity in the system. `type` contains the category of the entity, e.g. it specifies whether the entity represents a customer or a tagged service point. In the case of multiple customer classes, `type` can also be used to specify the customer class a customer entity belongs to. `time` denotes the timestamp of the location update, i.e. the time when the location update was recorded by the RTLS, and `x`, `y` specify the location of the tag at that particular instance. `stderr` is the expected deviation between the tag's recorded location and actual location.

3.2.4 Introducing New `Event` Classes

JINQS performs high-level Queueing Network simulations using discrete-event simulation (DES). DES maintains a time ordered list of various simulation events and the simulation time progresses instantaneously to the next event in the list when the processing of the current event is completed. Invoking an event may result to other events being added to the list. LocTrack-JINQS performs simulations in the same way.

Both JINQS and LocTrackJINQS use the `Event` class, along with its subclasses, to define events such as customer arrival events and service termination events. We note that these types of events, shared by both JINQS and LocTrackJINQS, do not contain any low-level information regarding customer location. To support the location-based features introduced earlier, two new `Event` subclasses are implemented:

- `TransportCustomersEvent` class. When such an event is triggered, it performs a call to the `moveCustomers` function which is contained in every instance of the `TransportLink` class. This event is scheduled to occur at frequent, regular time intervals, e.g. every few milliseconds, thus enabling the simulation of customer movement at fine resolution.
- `TagReadEvent` class. The execution of this event invokes the `updateTagReads` function of the `NetworkMonitor` class. By taking into account the two criteria mentioned in the previous section, i.e. QoS requirement and starvation avoidance, this function reads the locations[10] of the tags and outputs them to the trace file.

3.2.5 Graphical User Interface

LocTrackJINQS's graphical user interface is implemented by nineteen classes which are contained in the **gui** package. The main classes which consist LocTrackJINQS's interface, including their relationships, are depicted in Figure 3.12.

Through this interface, users are able to easily lay out the topology of the Queueing Network, specify the parameters of each added service point and its corresponding service area, and visualise the simulation as it progresses. Furthermore, customer paths between service points, i.e. links, can be explicitly seen (drawn as a straight line segment) and customised with a few mouse clicks; the user can click on a path and enter a new break point which can be dragged to the desired position.

The `MainUserFrame` class contains the components of LocTrackJINQS's GUI: an instance of the `ControlPanel`, `DrawingPanel`, `StatusBar` and `GUI_Sim` class. The `StatusBar` class is

[10]The read location is the tag's true location adjusted according to the user-defined error/noise distribution.

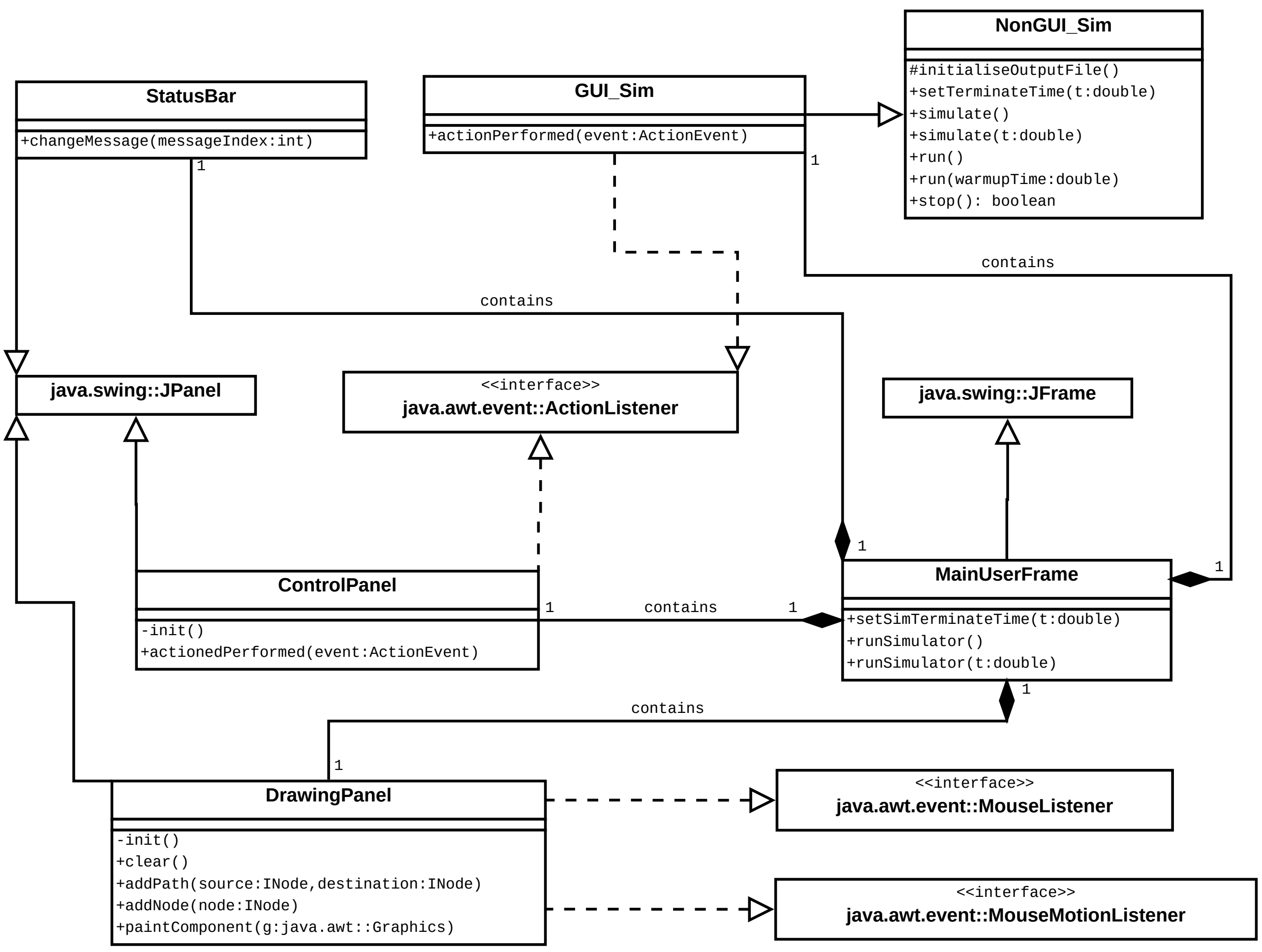

NonGUI_Sim
#initialiseOutputFile()
+setTerminateTime(t:double)
+simulate()
+simulate(t:double)
+run()
+run(warmupTime:double)
+stop(): boolean
StatusBar
+changeMessage(messageIndex:int)
GUI_Sim
+actionPerformed(event:ActionEvent)
contains
java.swing::JPanel
<<interface>>
java.awt.event::ActionListener
java.swing::JFrame
ControlPanel
-init()
+actionedPerformed(event:ActionEvent)
MainUserFrame
+setSimTerminateTime(t:double)
+runSimulator()
+runSimulator(t:double)
DrawingPanel
-init()
+clear()
+addPath(source:INode,destination:INode)
+addNode(node:INode)
+paintComponent(g:java.awt::Graphics)
<<interface>>
java.awt.event::MouseListener
<<interface>>
java.awt.event::MouseMotionListener
1

Figure 3.12: UML class diagram of the main classes in **gui** package.

used to display various messages which serve as guidelines on how to setup the simulation environment through the GUI.

The `ControlPanel` class acknowledges various events generated by pressing on certain `swing` components (`JButtons`). For each triggered event, a corresponding action is executed which usually requires some user input. This input is inserted through dialogs which also validate it before accepting it; when unacceptable values or data types are inserted, relevant messages are displayed to the user. User interaction via the `DrawingPanel` class only allows low-level path customisation or deletion, and the specification of the location of a new service point.

`Gui_Sim` class holds the simulation parameters, i.e. duration and warmup time, and it is responsible for removing and invoking scheduled `Events`. It maintains an instance of the `Timer` class, provided by the Java `swing` package, which enables the fine-grained animation of the customer movement; it produces `ActionEvent` instances every ten milliseconds which trigger the execution of various events, e.g. `TransportCustomersEvent`, and update the state of the simulation accordingly.

3.3 Evaluation

In this section we present two case studies: the first one simulates the security process in a small airport, and the second one simulates a simple $M/M/1$ queue. The purpose of the first case study is to demonstrate the functionality of some of the new features implemented in LocTrackJINQS and illustrate its GUI. Results associated with the response time (mean and standard deviation) and mean queue length of each customer class at each service point in the system are also presented. The second case study focuses on the validation of LocTrackJINQS's operation. The $M/M/1$ queue is simulated ten times and the averages of the server utilisation, mean queue length and mean response time are compared against their corresponding analytically obtained values.

3.3.1 Case Studies

The first case study simulates the security section of a small airport which consists of two passport control and two passenger screening checkpoints, each with its own queue. Figure 3.13 provides a description of the high-level topology of the airport's security section. Customers entering this system are processed in a way similar to a standard airport security process: before entering the airport's departure lounge, a passenger is subject to one passport control check and one security screening.

Considering an EU airport, we differentiate customers into three distinct classes:

- Pilots and flight attendants (class 0). This class of customers has high priority.
- EU passengers (class 1). This customer class has low priority.
- Non-EU passengers (class 2). This class of customers also has low priority.

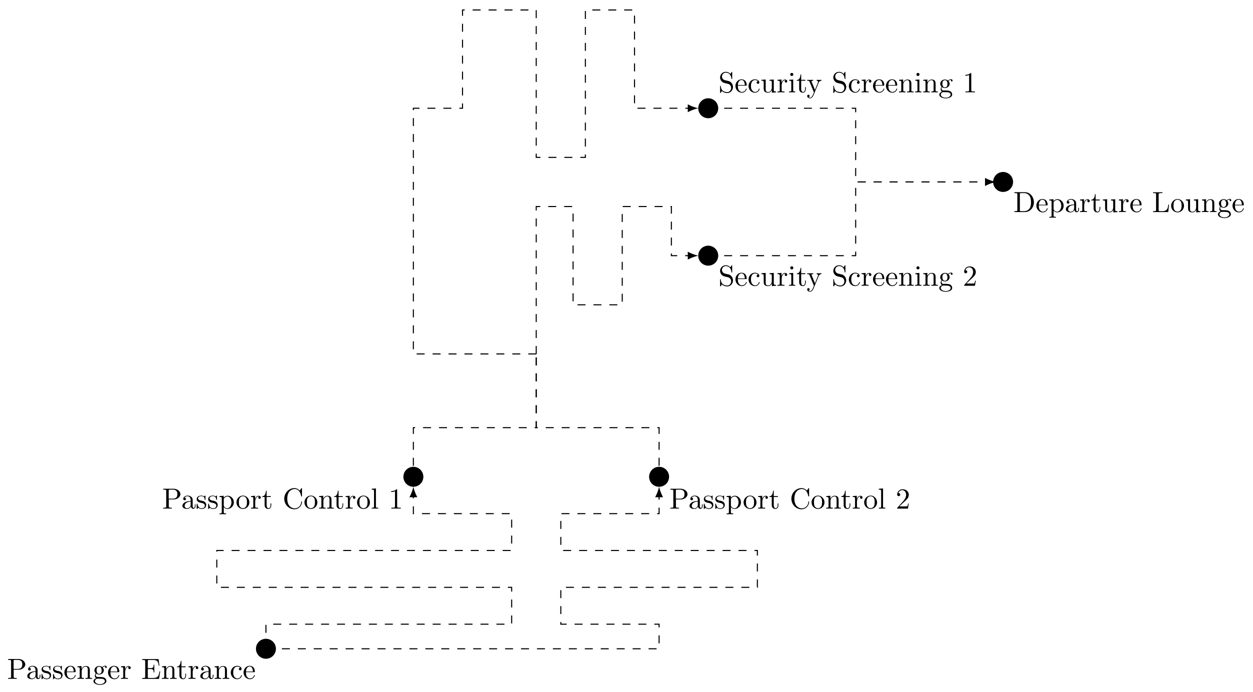


Figure 3.13: Case study one: high-level topology of the airport's security section. Dashed arrows are used to represent the flow of customers in the system.

The model for this system, constructed via LocTrackJINQS's GUI, is depicted in Figure 3.14. Node N1 represents the passengers entry point and is implemented by the `SourceWithClassDist` class. The passport control checkpoints, represented by nodes N2 and N3, are implemented by two identical instances of the `QueueingServer` class. Their service radius is set to be one metre and their corresponding service area is indicated by the dashed circles. An additional node (N4) – not explicitly specified in Figure 3.13 – is used to merge the flow of customers departing from N2 and N3, and route customers to nodes N5 and N6 which represent the passenger screening checkpoints. Since N4 provides no "real" service, it is implemented as an infinite server with zero service time. The implementation of N5 and N6 is similar to that of N2 and N3, but with a service radius of 1.5 m. The two branchings of the customer flow, i.e. from N1 to N2 and N3, and from N4 to N5 and N6, are implemented as instances of the `JoinShortestQueueRouting` class since they are best suited to emulate the customers' natural behaviour in such type of systems. N7, an instance of the `Sink` class, is the model's exit point; a customer arrival at N7 denotes the completion of that customer's security check and its entry in the airport's departure lounge. All service points in this system employ a priority-based queueing discipline and customers that belong in the same priority class queue in a FIFO fashion.

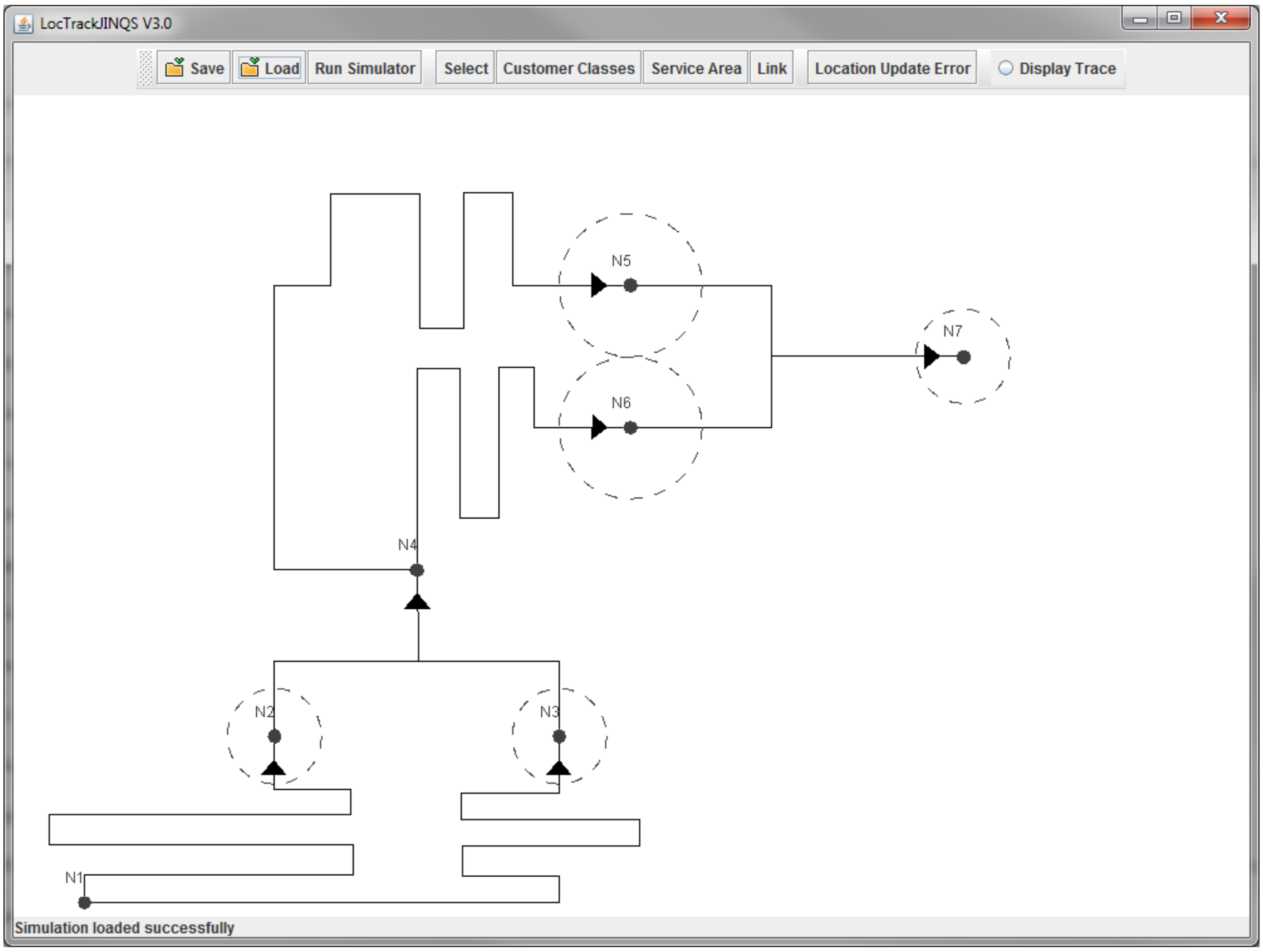


Figure 3.14: Case study 1: the airport simulation model as created using the GUI of LocTrack-JINQS.

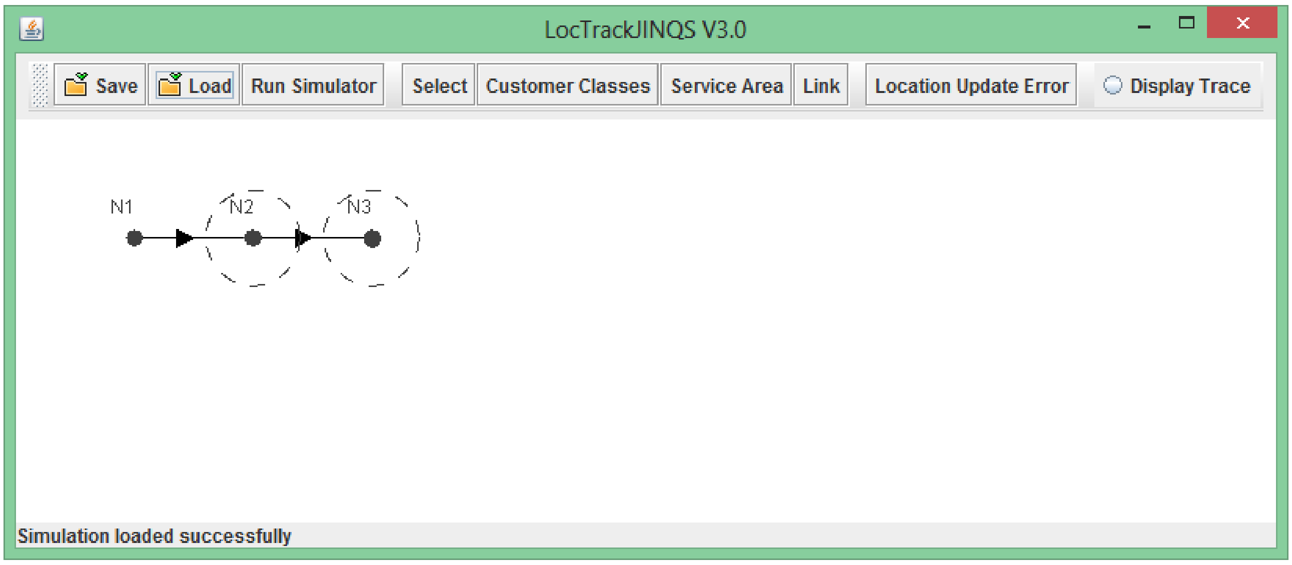
The type and parameters of the inter-arrival and service time distribution for each customer class, at each service point are depicted in Table 3.1.

	Pilots & Flight Attendants	EU Passengers	Non-EU Passengers
Passenger Entrance (Inter-arrival time)	Exp(0.02)	Exp(0.33)	Exp(0.2)
Passport Control (Service time)	Normal(2, 0.5)	Exp(0.25)	Erlang(2, 0.33)
Passenger Screening (Service time)	Exp(0.2)	Erlang(4, 0.1)	Erlang(4, 0.1)

Table 3.1: Case study 1: the specified inter-arrival/service time distributions for each type of service point and customer class of the airport simulation model.

The speed of all customers in this simulation environment is assumed to follow a Normal distribution. Pilots and flight attendants' speed distribution has a mean of 0.45 m/s and a standard deviation of 0.1 m/s. Passengers (EU and non-EU) share the same speed distribution with a mean and standard deviation of 0.3 m/s and 0.1 m/s respectively. The location update error is also normally distributed with mean 0.15 m and standard deviation 0.2 m.

The second case study simulates an $M/M/1$ queue ten times. Customer arrivals follow an exponential inter-arrival time distribution with rate 0.03 customers/s. The service time is also exponentially distributed with rate 0.05 customers/s. The queue (implemented as an instance of the `QueueingServer` class) consists of one server and employs a FIFO queueing discipline. Infinite queue capacity is also assumed. Figure 3.15 illustrates this system as modelled in LocTrackJINQS. As we discussed in the introduction of this section, the purpose of this case



Simulation loaded successfully

Figure 3.15: Case study 2: the $M/M/1$ queue simulation model as created using the GUI of LocTrackJINQS. N1 and N3 represent the system's entry and exit points respectively. The queue is represented by N2.

study is to assess the operation of LocTrackJINQS using quantitative results. We compute the average of standard performance measures, such as server utilisation, mean response time and mean queue length, obtained through the simulation of this queue and compare them against the corresponding analytical results. In order to simulate this queue without accounting for travelling delays of customers, these delays can cause inadmissible discrepancies between the simulation and steady-state analytical results, customers must travel very fast, in fact instantaneously. For this reason, we set the speed of all customers equal to 100 m/s (a Deterministic distribution is employed). We also note that a warm-up period of 100 s is employed in each simulation to obtain a better approximation of the steady state analytical results.

3.3.2 Results

Figures 3.16 and 3.17 show the (on-going) simulation of the airport model, at different time instances. Light blue, red and green coloured person figures represent pilots and flight attendants, EU and non-EU passengers respectively. The colour of the small circle located above each person figure indicates the customer's status: blue indicates movement, red queueing and light green rendered service.

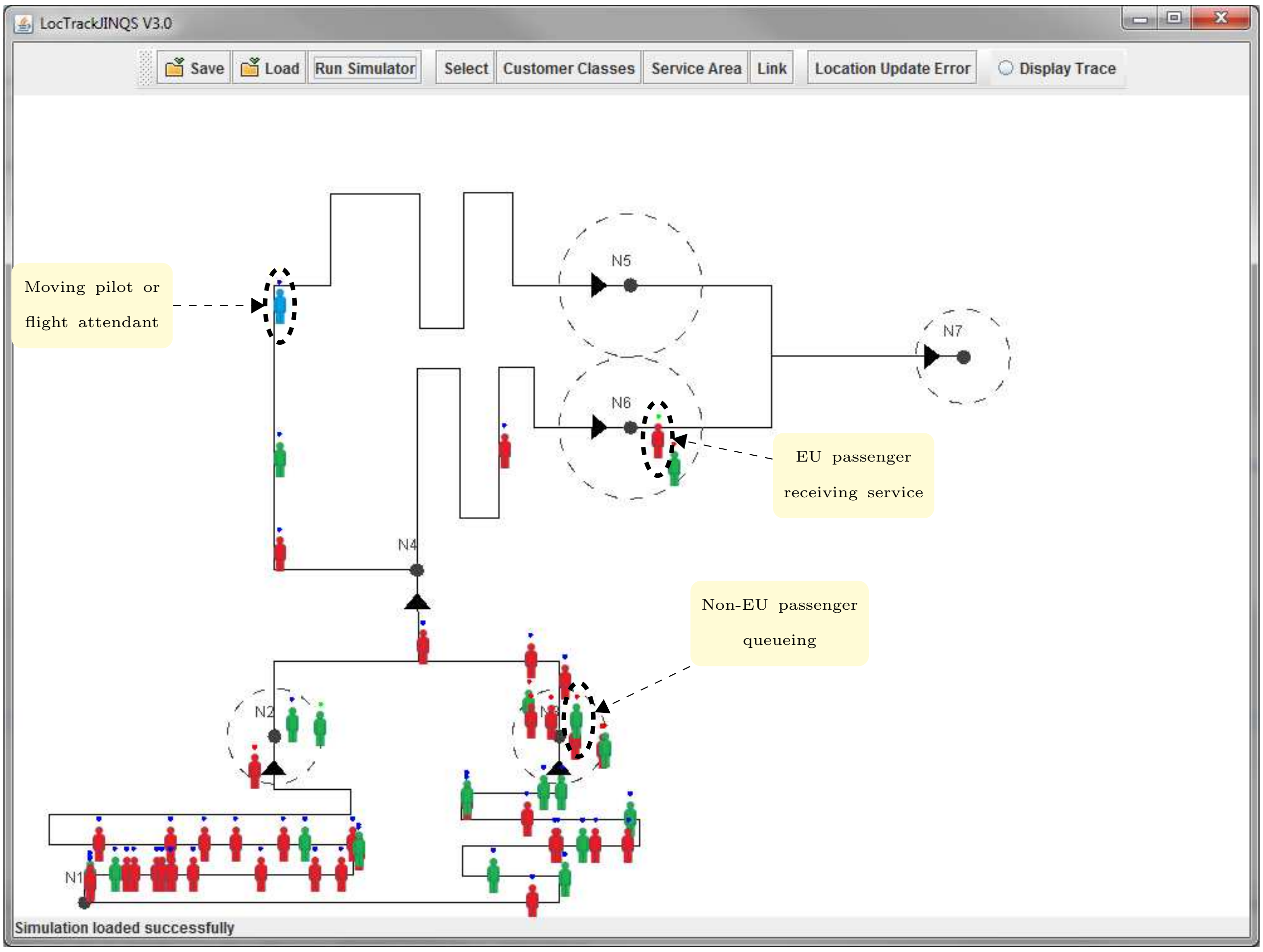
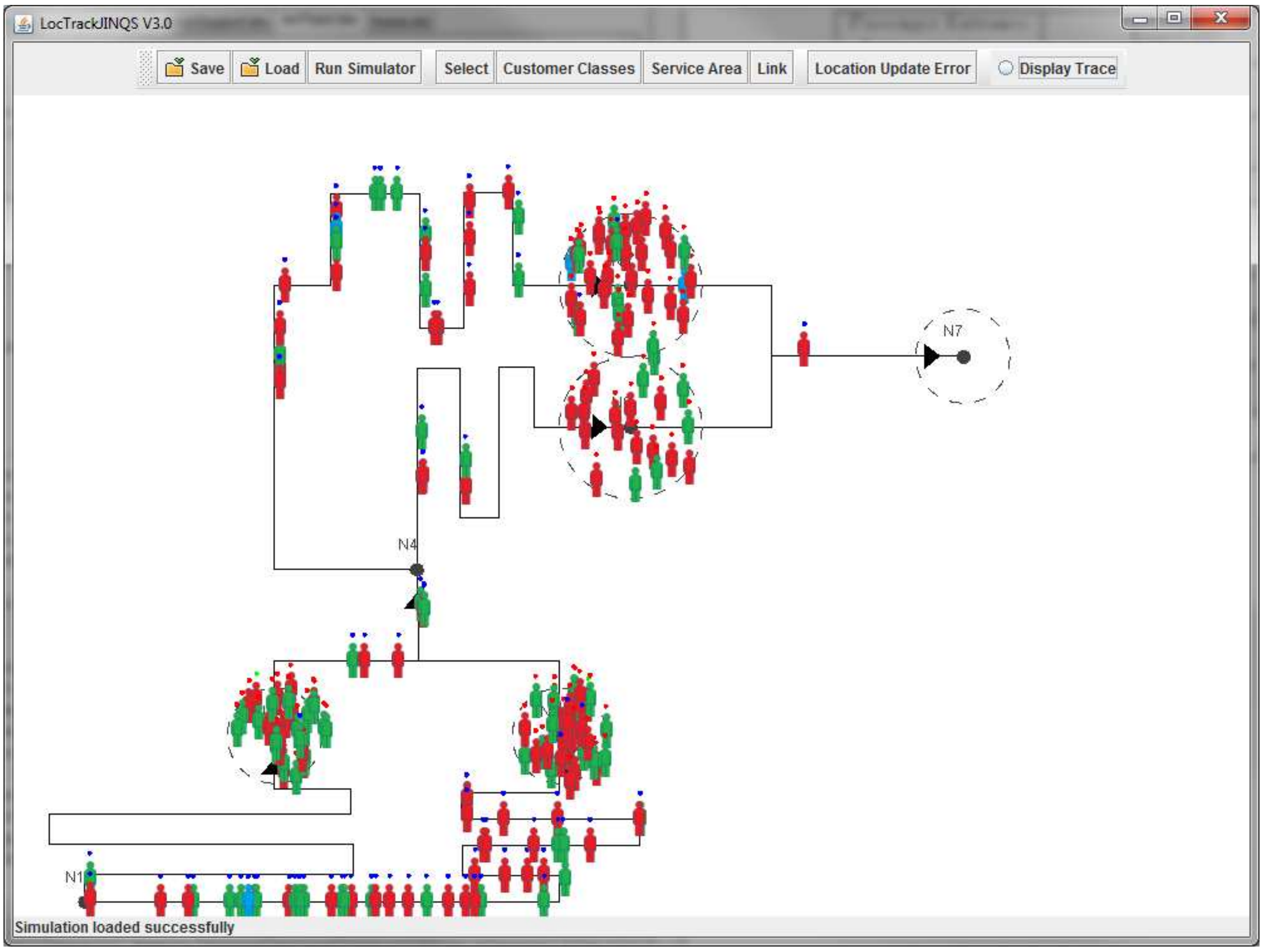


Figure 3.16: Case study 1: the on-going simulation of the airport model at an early stage.

Table 3.2 displays the mean and standard deviation of the response time for each customer class at each service point, as well as the mean queue length at each service point, for case study one. It can be seen from these results that the quality of service received by customers with high priority at each node, i.e. pilots and flight attendants, is much better than that of the remaining classes. In fact, the mean response time experienced by these customers – at every

		Passport Control 1 (N2)	Passport Control 2 (N3)	Passenger Screening 1 (N5)	Passenger Screening 2 (N6)
Class 0	μ_{rt}	5.462	6.936	26.652	23.051
	σ_{rt}	2.337	3.423	19.931	18.525
Class 1	μ_{rt}	226.279	250.316	573.261	485.416
	σ_{rt}	117.048	139.623	295.540	287.235
Class 2	μ_{rt}	247.300	261.040	391.637	504.648
	σ_{rt}	101.015	145.224	277.569	314.843
	μ_q	57.566	59.936	88.321	89.463

Table 3.2: Case study 1: mean (μ_{rt}) and standard deviation (σ_{rt}) – in seconds – of the response time for each customer class at each service point of the airport simulation. Customer classes 0, 1 and 2 correspond to Pilots & Flight Attendants, EU passengers and non-EU passengers respectively. μ_q represents the mean queue length at each service point.



Simulation loaded successfully

Figure 3.17: Case study 1: the on-going simulation of the airport model as time progresses.

service point – is, relatively, close to the mean of their assigned service time distribution, i.e. 2 seconds at passport control and 5 seconds at passenger screening. Furthermore, the standard deviation of their response time is low. The discrepancy between the mean of the assigned service time distribution and the obtained mean response time is the result of delays inflicted by customers of lower priority whose service was already in progress upon the arrival of high priority customers at the service point. The mean and standard deviation of the response time for each customer class at the system level demonstrate that pilots and flight attendants consistently receive better service than EU and non-EU passengers (see Table 3.3).

	Class 0	Class 1	Class 2	Aggregate
μ_{rt}	161.023	704.522	634.237	533.790
σ_{rt}	42.855	321.807	322.123	360.561

Table 3.3: Case study 1: mean (μ_{rt}) and standard deviation (σ_{rt}) – in seconds – of the response time for each customer class, as well as the aggregate, for the whole system. Customer classes 0, 1 and 2 correspond to Pilots & Flight Attendants, EU passengers and non-EU passengers respectively.

The results of the second case study are shown in Table 3.4. We observe that the computed average of utilisation, mean response time and mean queue length, obtained by simulating the system depicted in Figure 3.15 ten times, approximate the corresponding analytically computed values well. Furthermore, the analytically computed values lie well within the 95% confidence

Performance Measure	Simulated Value (Avg.)	95% Confidence Interval	Analytical Solution
Utilisation (ρ)	0.622	$(0.586, 0.659)$	0.6
Mean Response Time (μ_{rt})	50.453 s	$(37.537, 63.368)$	50 s
Mean Queue Length (N_q)	0.887 customers	$(0.731, 1.043)$	0.9 customers

Table 3.4: Case study 2: standard performance measures obtained through simulation and their corresponding analytically computed values, for the system depicted in Figure 3.15. The simulated values and the corresponding 95% confidence intervals are given to three decimal places.

intervals of the computed means. In addition to the results shown in Table 3.4, we also perform two-sided, one-sample t-tests in order to test whether the mean of each performance measure sample is equal to the corresponding analytically computed steady state value (cf. Table 3.5).

The results obtained in the second case study, as well as the values of the performance measures obtained in case study one, suggest that LocTrackJINQS operates correctly and produces data that preserve the properties of the specified Queueing Network model.

Performance Measure	Null Hypothesis	Test Statistic	Critical Value	Accepted ?
Utilisation (ρ)	$\rho = 0.6$	1.3735	2.2622	Yes
Mean Response Time (μ_{rt})	$\mu_{rt} = 50\,\mathrm{s}$	0.0793	2.2622	Yes
Mean Response Time (N_q)	$N_q = 0.9$ customers	-0.1847	2.2622	Yes

Table 3.5: Two-sided, one-sample t-test at significance level 0.05 applied to each sample of performance measures obtained in case study two. The test statistics and critical values are given to four decimal places.

3.3.3 Conclusion

This chapter presented LocTrackJINQS, a location-aware simulation tool, developed to support location-based research. Its development is based on the discrete-event simulation library for Queueing Networks, JINQS. LocTrackJINQS maintains the simplicity and extensibility of JINQS, and allows low-level location information to be incorporated in its simulation models, i.e. the location of entities and their spatiotemporal dependencies. Thus, it can generate location tracking data, approximating those of an actual RTLS, as the simulation progresses.

Furthermore, simulations in LocTrackJINQS can be either specified programmatically or visually; simulations can be constructed visually through LocTrackJINQS's GUI. In addition, its GUI enables the graphical visualisation of on-going simulations, as it was demonstrated by the first case study presented in Section 3.3.1. Standard performance measures can also be computed and displayed at the end of each simulation.

In addition to the results of the two case studies presented in Section 3.3.2, LocTrackJINQS is continuously evaluated through the course of this thesis: it is used to generate location tracking data for the six case studies presented in Chapters 4, 5 and 6. As we will see in detail later on, the results obtained from these case studies suggest that the PNPMs, inferred by our methodology, match the structure and parameters of the abstract system, prior to its simulation in LocTrackJINQS (cf. Figure 4.15). This evidence reinforces our conjecture regarding the correct operation of LocTrackJINQS and its ability to generate data that accurately reflect the properties of the specified Queueing Network, i.e. routing probabilities of the customer flow and service time distributions.

Chapter 4

Model Inference Pipeline

In this chapter we present an automated technique which takes as input high-precision location tracking data – potentially collected from a real life system – and constructs a hierarchical Generalised Stochastic Petri Net (GSPN) performance model of the underlying system. This is based on a coarse-grained approach which aims to provide a high-level description of the physical agent flow in the system.

The methodology presented here follows the four-stage data processing pipeline shown in Figure 4.1, which takes as input raw location tracking traces and outputs the Petri Net Performance Model (PNPM) in a format compatible with PIPE2 [20]. This methodology operates based on the following assumptions (see also Section 1.2):

1. The system's service centres, also referred to as service areas, are static.
2. The system employs single-server semantics and a random service discipline. Although the methodology can successfully extract the service time of service areas which employ a First-Come-First-Served (FCFS) discipline, it is, in general, hard to represent FCFS in GSPNs.
3. No tag recycling is supported.
4. The customers stop or slow down inside a service area in order to receive service.

The first stage of the pipeline prepares the input data for processing. The second stage infers the locations and radii of the service areas in the system. We note that during this stage additional service areas may be inferred which will not correspond to a physical service process. Such service areas serve as an indication of a high congestion point or a spatial bottleneck in the system which would probably not be captured through a traditional (manual) model construction process. The third stage creates the initial structure of the PNPM with the required places and transitions. Here, samples of sojourn times in each service area and samples of travelling times between each pair of service areas are also extracted. The extracted sojourn time samples are then processed to yield the service time samples. The final task of this stage is the computation of the initial routing probability of the customer flow. In the final stage we fit a hyper-Erlang distribution to each set of extracted service or travelling time samples using the G-FIT tool [106] and refine the structure of the model accordingly.

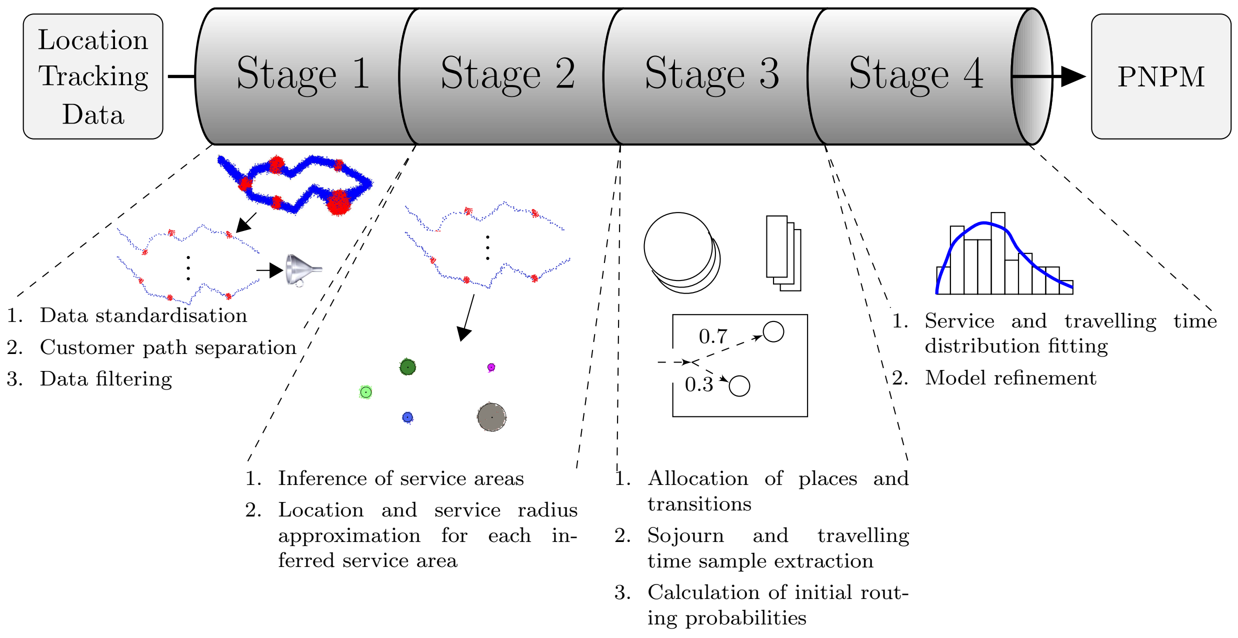

Location Tracking Data
Stage 1
Stage 2
Stage 3
Stage 4
PNPM
1. Data standardisation
2. Customer path separation
3. Data filtering
1. Inference of service areas
2. Location and service radius approximation for each inferred service area
0.7
0.3
1. Allocation of places and transitions
2. Sojourn and travelling time sample extraction
3. Calculation of initial routing probabilities
1. Service and travelling time distribution fitting
2. Model refinement

Figure 4.1: The four-stage data processing pipeline.

Naturally, since the GSPN performance model is constructed by analysing the location traces of the customer flow in the underlying customer-processing system, there is dependency between the inferred PNPM and the collected location tracking data. Ultimately this means that the model will reflect the underlying system during the period that the customers of the system were monitored. In particular, especially in customer-processing systems where resources are dynamically allocated during the system's operation according to service demand, e.g. operating bank cashiers, airline check-in points, supermarket tills etc., heavier customer traffic loads may lead to the discovery of additional service areas and bottlenecks. Nevertheless, we note that the response time of the service areas that have been discovered during lower workloads, will be valid under heavier workloads since our methodology only infers the service time distribution of each service area; the waiting time is thus implicitly modelled according the number of waiting customers at each service area. However, if additional service areas are discovered, a similar argument does not hold for the overall system's response time.

The remainder of this chapter provides further details for each stage of the developed data processing pipeline. We then present two case studies based on synthetic location tracking data, generated using LocTrackJINQS [56] (cf. Chapter 3), on which we applied our methodology. This chapter concludes with an evaluation of the obtained results and a discussion of several limitations of the current methodology which are addressed in the following two chapters.

4.1 Stage 1

The aim of this stage is to prepare the raw location tracking data, retrieved from a customer-processing system via an RTLS or simulation, for processing by the subsequent stages. This data is essentially a stream of spatiotemporal updates where each such update contains – amongst other information – the position of its associated tag at a particular time (see Figure 4.2).

The structure of a location update can vary depending on the type of RTLS being used and on the software component used to extract the location trace file. Therefore, the raw input data are converted into a standardized, source-independent format. Currently, the data conversion component supports UWB-based location tracking data generated from a Ubisense RTLS and synthetic location tracking data generated by LocTrackJINQS [56], but it can be easily extended to support location tracking data from other sources.

The standardised data are then separated into paths, one for each recorded customer, identified by the `tagName`. A path is the set of location updates which have the same `tagName`. A speed-

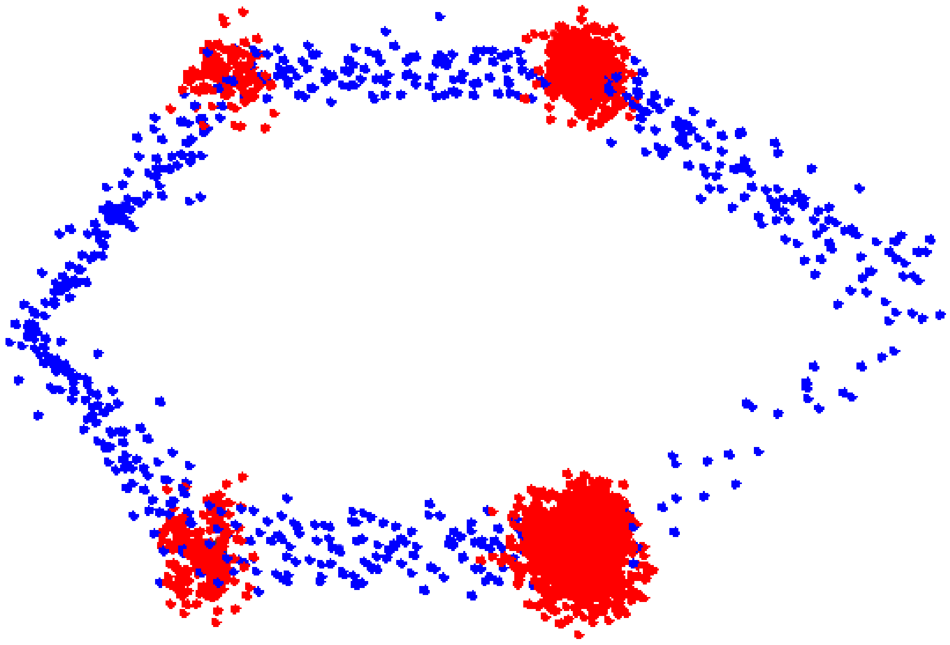


Figure 4.2: Graphical representation of a stream of location traces, i.e. the input of the first stage of the processing pipeline, obtained from a system with four service areas. Blue dots represent location updates recorded during customer movement. Red dots represent location updates recorded when customers were stationary.

based filter is then applied on each path to remove possible erroneous location updates which are usually caused by signal reflection and refraction, and bad sensor geometry. This filter imposes a speed threshold which defines the maximum allowable average speed that a customer can move within the monitored environment. Its operation is described below.

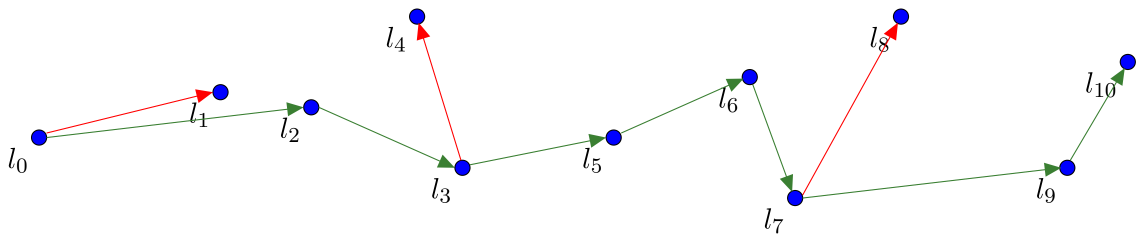


Figure 4.3: A graphical representation of the operation of the speed-based filter used to eliminate erroneous location updates. Each l_k, $k = 0, \dots, 10$ represents a location update and red arrows denote rejected path segments. The filtered path is denoted by green arrows.

Given two location updates, say l_i and l_{i+j}, for some j, the average speed is computed by $d_{i+j}/(t_{i+j} - t_i)$ where d_{i+j} denotes the distance between them and t_i, t_{i+j} their timestamps. If the value of the average speed is less than or equal to the threshold value, then l_{i+j} is considered valid and added to the filtered customer path; otherwise it is discarded and the process is repeated for l_i and the next available location update. The value of the speed threshold is by default set equal to 1.667 m/s which is sufficient to accommodate for systems where customers (people) walk, and it can be easily adjusted according to the nature of the system to be modelled. This filtering process is illustrated in Figure 4.3.

4.2 Stage 2

This stage infers the number of service areas in the system as well as their corresponding locations and service radii. The main assumption here is that customers stop, or slow down while receiving service and thus we can identify the regions where the customer movement is relatively 'slow', as the likely service areas. This task consists of a three-layer technique (see Figure 4.4): speed filtering, density filtering and the application of the DBSCAN clustering algorithm [41].

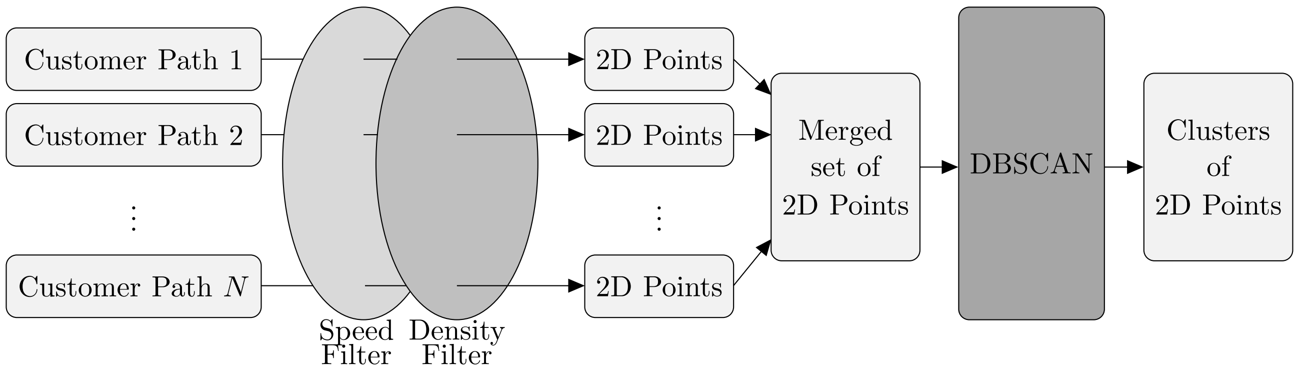


Figure 4.4: Stage two of the data processing pipeline. The centroid of each cluster is likely to approximate the location of a service area.

4.2.1 Speed Filtering

The speed filter is applied on each customer path (computed during the first stage) and aims to identify the 2D spatial points[1] of the location updates that were recorded when the customer was stationary or moving at a relatively low speed. The operation of this filter is as follows:

[1]We employ a two-dimensional framework.

for each customer path we calculate a raw speed curve describing the average speed of the customer for each time interval between consecutive readings. That is, if the distance between location readings l_i, l_{i+1} with timestamps t_i and t_{i+1} is d_{i+1}, the corresponding point on the raw speed curve is $(\bar{t_i}, v_i)$ where

$$\bar{t_i} = \frac{t_i + t_{i+1}}{2} \ , \ v_i = \frac{d_{i+1}}{t_{i+1} - t_i} \ .$$

An example of the resulting curve is shown Figure 4.5(a). We then compute the 10th percentile of the raw speed vector v, $v = (v_0, v_1, \ldots, v_{|v|-1})$, and use it as a threshold to identify the lowest speed values (cf. Figure 4.5(b)).

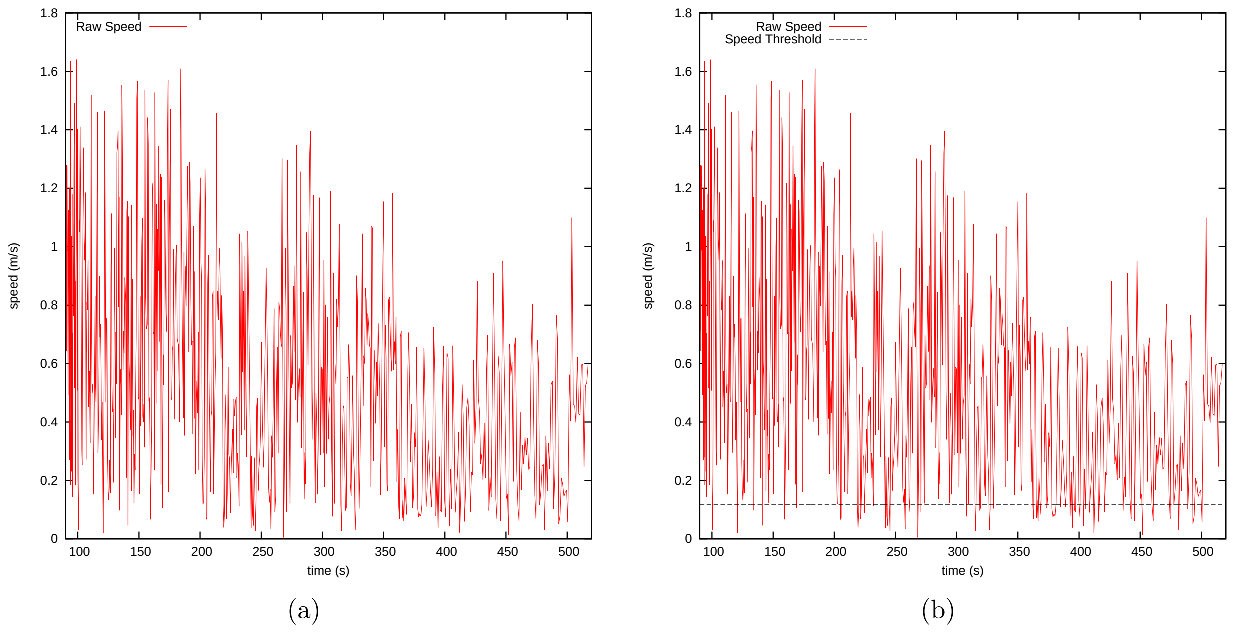


Figure 4.5: The raw speed curve (left) and the speed threshold applied to identify 'low' speed values which suggest that the customer was in a service area waiting to be or being serviced (right).

The spatial points of the location updates that constituted the computation of each identified v_i, i.e. the position of the location readings l_i and l_{i+1}, are retrieved and stored in a list (one list per customer). We note that this list only includes 2D points from distinct location updates. For example, if v_i and v_{i+1} have been identified as 'low' speed values, then we retrieve the position of l_i and l_{i+1} for v_i and similarly, the position of l_{i+1} and l_{i+2} for v_{i+1}. However, we only store three points in the list: the position of l_i, l_{i+1} and l_{i+2}.

4.2.2 Density Filtering

The spatial points retrieved from the previous layer do not necessarily correspond to the position of a customer located in a service area since the customer may move at variable speed and/or may pause en route between service areas. To ensure that most[2] such points are removed we apply this density filter on each customer path, immediately after the application of the speed filter (see Figure 4.4).

Here we make use of the notion of the $Eps - neighbourhood$ of a point p [41], which for a dataset D is defined as

$$N_{Eps} = \{q \in D \mid dist(p,q) \leq Eps\}$$

That is, the set of points in D that lie in the circular area of radius Eps around the point p. For each point p that has "survived" the speed filter we compute $N_{Eps}(p)$ and we examine if the number of points contained in this circular area exceeds a certain threshold $MinPts$, i.e. $|N_{Eps}(p)| \geq MinPts$. This condition specifies the minimum number of points that should be present within the set N_{Eps} so that the point p remains in the list (see Figure 4.6). In our implementation the value of $MinPts$ is chosen to be four, according to the suggestion of [41]. Eps is chosen to be 0.25 (metres) because the error of typical UWB-based RTLS is approximately this value. When this process is completed for every list of two-dimensional points obtained by the speed filter, the remaining points from each list are merged (cf. Figure 4.4).

We note that we allow the reconfiguration of these values, i.e. $MinPts$ and Eps, in order to make this filter more sensitive (increase $MinPts$), or insensitive (decrease $MinPts$ and/or increase Eps). This is useful when modelling systems with high congestion where customers move throughout the system at a very low speed or stop on several instances. In this case one would increase the sensitivity of the density filter so that only true service areas are inferred; in order to allow the inference of possible bottlenecks, e.g. regions of high traffic, one would decrease the filter's sensitivity.

[2]If a customer pauses en route between service areas, the points of those location updates may not be filtered out. This depends on the duration of the customer's pause and the tag update rate. Action against such cases is taken after then DBSCAN clustering algorithm is applied.

Figure 4.6: Illustration of the density filtering process. Point p remains in the list as the number of points present in $N_{Eps}(p)$ is five (left), and p is eliminated from the list in the case where $|N_{Eps}(p)| = 3$ (right).

4.2.3 DBSCAN clustering

In the third layer, the DBSCAN algorithm [41] is applied to group the points contained in the aggregated filtered dataset, emerging from the previous layer (cf. Figure 4.4), into clusters, provided that they satisfy a density criterion (see Section 2.5.2). As in the previous subsection, this criterion is specified by two parameters, $MinPts'$ and Eps'.

Again, we choose $MinPts'$ to be four, as our experience and that of others is that larger choices do not produce any significant difference in results, while they rapidly become computationally prohibitive [41]. The first step in finding a suitable value of Eps' is to compute the *4-dist* value for each point p. As mentioned earlier (Section 2.5.2), a suitable value of Eps' is identified by finding the *4-dist* value of the first “valley” in the sorted *4-dist* graph. This value of Eps' best differentiates noise (points to the left of the valley) from points that potentially lie within service areas (points to the right of the valley).

Ideally, the filtered dataset should contain no noise points. In the context of this work, we consider noise points to be points that correspond to location updates recorded while a customer was moving towards a service area. The speed and density filters presented in the previous two subsections are applied to remove such points from the dataset. Thus, in order for all the points in the filtered dataset to be assigned to some cluster, we require that the Eps' value is set equal to the maximum *4-dist* value. That is the first *4-dist* value of the *4-dist* graph, since the *4-dist* values are sorted in a descending order.

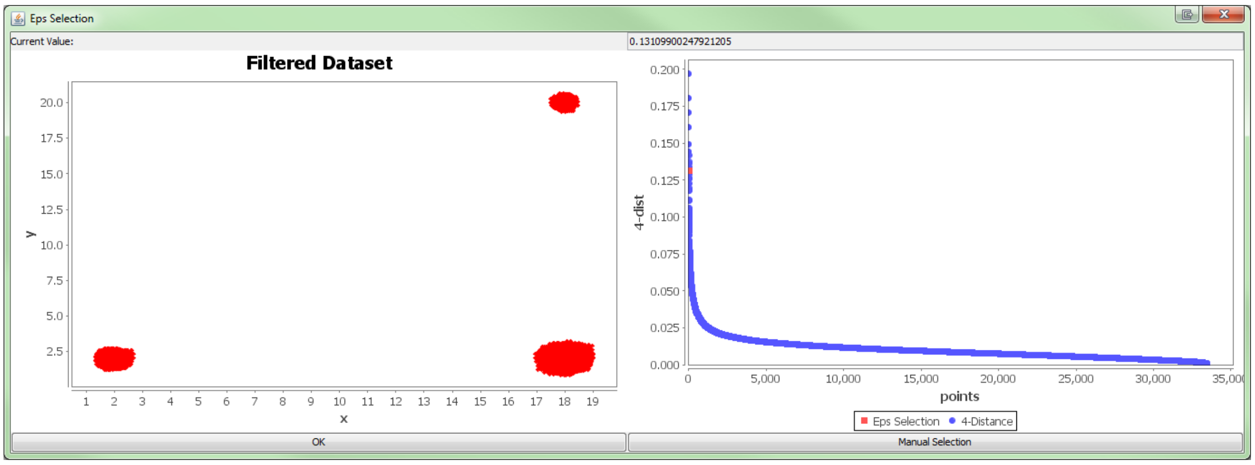
However, in some cases not all noise points are removed. When this occurs, the challenge is to choose the appropriate Eps' value that will best distinguish noise points from points that actually form the clusters corresponding to service areas. As we intend to keep user input at a minimum in our approach, we applied a simple, empirical technique to approximate the value of Eps' automatically when noise points are present in the dataset. First, we obtain the set of *4-dist* values that lie in the first half percentile, i.e. 0.5% of the total number of values. We then compute their mean (μ_{hp}) and standard deviation (σ_{hp}). We approximate the Eps' value by the first *4-dist* value greater than two standard deviations above the mean. That is:

$$Eps' \approx \min(\{\textit{4-dist} \,|\, \textit{4-dist} > \mu_{hp} + 2\sigma_{hp}\})$$

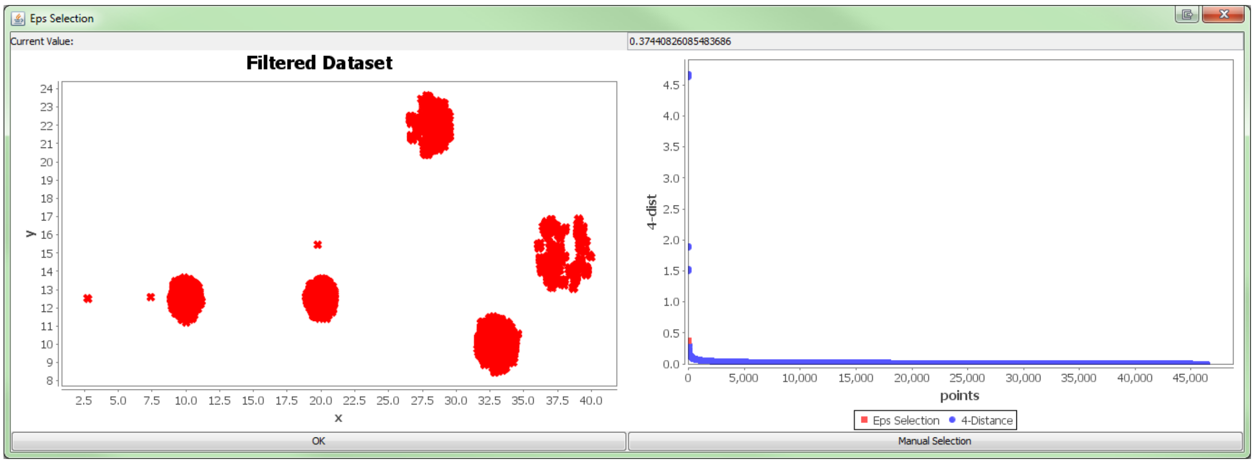
Unfortunately, we were not able to automatically detect the cases where noise points are present in the filtered dataset. Thus, we always apply the latter technique and provide a graphical user interface (see Figure 4.7) which displays the automatically selected value for Eps', the sorted *4-dist* graph, as well as the filtered dataset. If no "significant" noise points are contained in the filtered dataset (e.g. Figure 4.7(a)) – such points are usually isolated and thus can be easily detected by the user (e.g. Figure 4.7(b)) – a manual selection is supported through this GUI. The user can select the correct Eps' value by clicking on the first point on the *4-dist* graph (if its not already selected).

We consider a cluster which was output by DBSCAN to be valid if the cluster contains a substantial number of points. In particular, we require that the number of points contained in a cluster exceeds 1% of the total number of clustered points. This action is mainly taken to account for clusters created by limited groups of points which may have "survived" the speed and density filters and satisfied DBSCAN's density criterion.

The centroids of the valid clusters approximate the real locations of the corresponding service areas. The radius of each service area is conservatively approximated as the 110% of the 95th percentile of the distance between each point in the cluster and the cluster's centroid. An example of this process can be seen in Figure 4.8.


Current Value: 0.13109900247921205

(a) The filtered dataset does not contain any noise points. The user should select the first *4-dist* value.

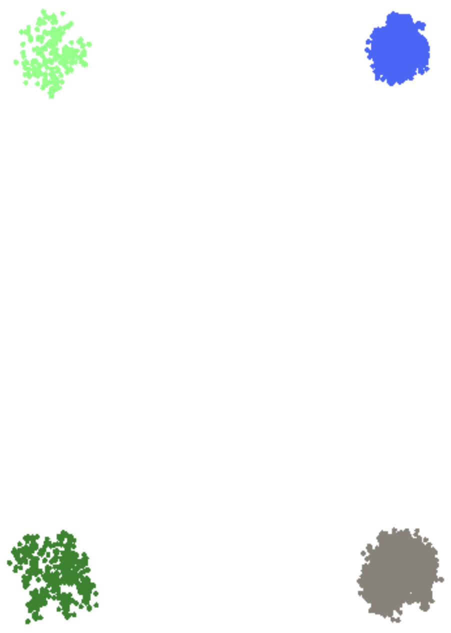

Current Value: 0.37440826085483686

(b) The filtered dataset contains noise points. The correct *Eps′* value has been successfully selected automatically.

Figure 4.7: *Eps′* selection graphical user interface.

4.3 Stage 3

Stage 3 constructs the basic structure of the derived PNPM. Here, we will use the example shown in Figure 4.8 to visualise each step of this stage. We first create places associated with the service areas inferred from the previous stage (see Figure 4.9). The next step is to create places associated with customer movement between service areas, one for every pair of service areas between which customer movement was observed, and transitions connecting places representing service areas to those representing customer movement (see Figure 4.10(a)). We call these transitions, service area service time transitions. Transitions are then created to connect the places associated with customer movement to places representing destination service areas (see Figure 4.10(b)). We call these transitions travelling time transitions.


(a)

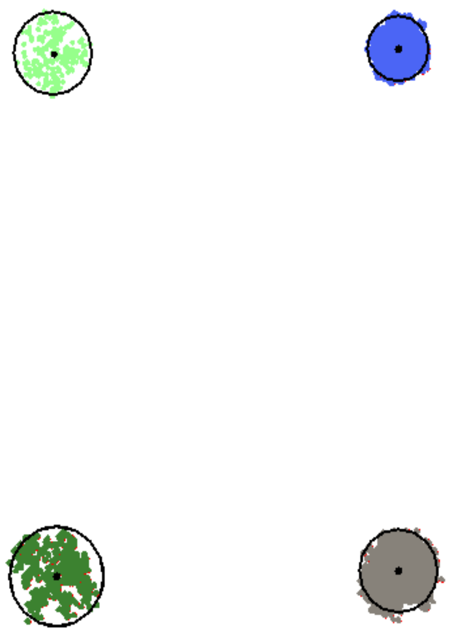

(b)

Figure 4.8: Figure 4.8(a) shows the colour-coded clusters produced by DBSCAN for the set of location traces depicted in Figure 4.2, and Figure 4.8(b) shows each service area's location and service radius approximation.

For the moment, we do not parameterise the rates of the transitions; in fact in the next section we show how we replace each transition by a GSPN subnet that accurately reflects the distribution of the relevant time delays. In preparation for this, here we compute – for each customer – samples of their sojourn times inside service areas (response time samples) broken down into waiting time and service time.

In order to estimate the service time a customer receives at a service area, we first estimate when the customer enters the service area (entry time) by taking the average of two timestamps called the first appearance time and last disappearance time [57]. Based on the customer's location traces, the first appearance time corresponds to the first timestamp when the customer is identified to be inside the service area; the last disappearance time is defined as the last timestamp when the customer is considered to be outside the service area (prior to the first appearance time). The customer's exit time is computed in a similar way: we take the average of the timestamps of the two location updates that correspond to the last appearance and first disappearance [57] (see Figure 4.11). We maintain a list for each service area which contains

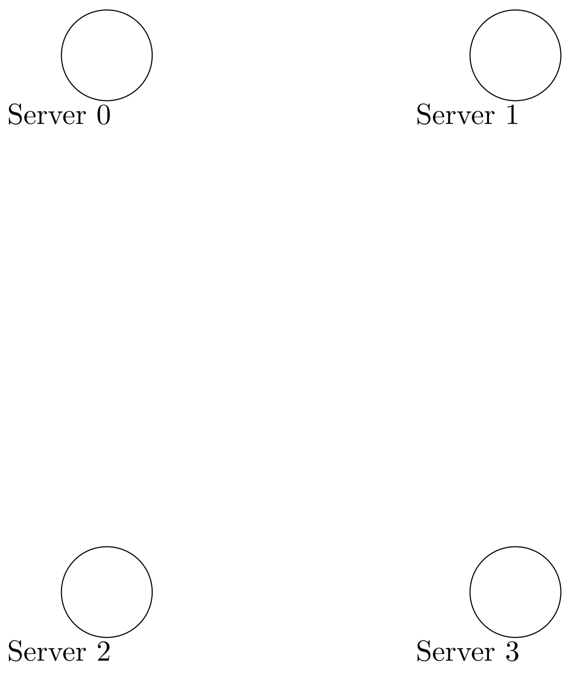


Figure 4.9: Creation of places representing service areas, one for each inferred service area from Figure 4.8.

instances of a data structure which holds – amongst other information – each customer's entry and exit times and allows its instances to be sorted according to exit time[3]. At each exit time of a customer, we check if the previous customer's exit time is larger than this particular customer's entry time. In this case, the estimated service time is the difference between the current customer's exit time and the previous customer's exit time. Otherwise, the server was idle upon the customer's arrival so the service time is simply the difference between the customer's exit and entry times. An example of this process is shown in Table 4.1. We then compute travelling time samples by subtracting the customer's exit time at the upstream service area from the customer's entry time at the downstream service area along the customer's path.

Customer Id	Entry Time (s)	Exit Time (s)	Service Time (s)
Customer 1	10.50	13.40	2.90
Customer 2	12.47	18.33	4.93
Customer 4	8.11	20.27	1.94
Customer 3	22.10	26.41	4.31
Customer 5	25.03	33.84	7.43

Table 4.1: Service time estimation process from the recorded entry and exit times.

[3]We assume that a random service discipline is employed in line with the conventional service discipline used in Petri Net models.

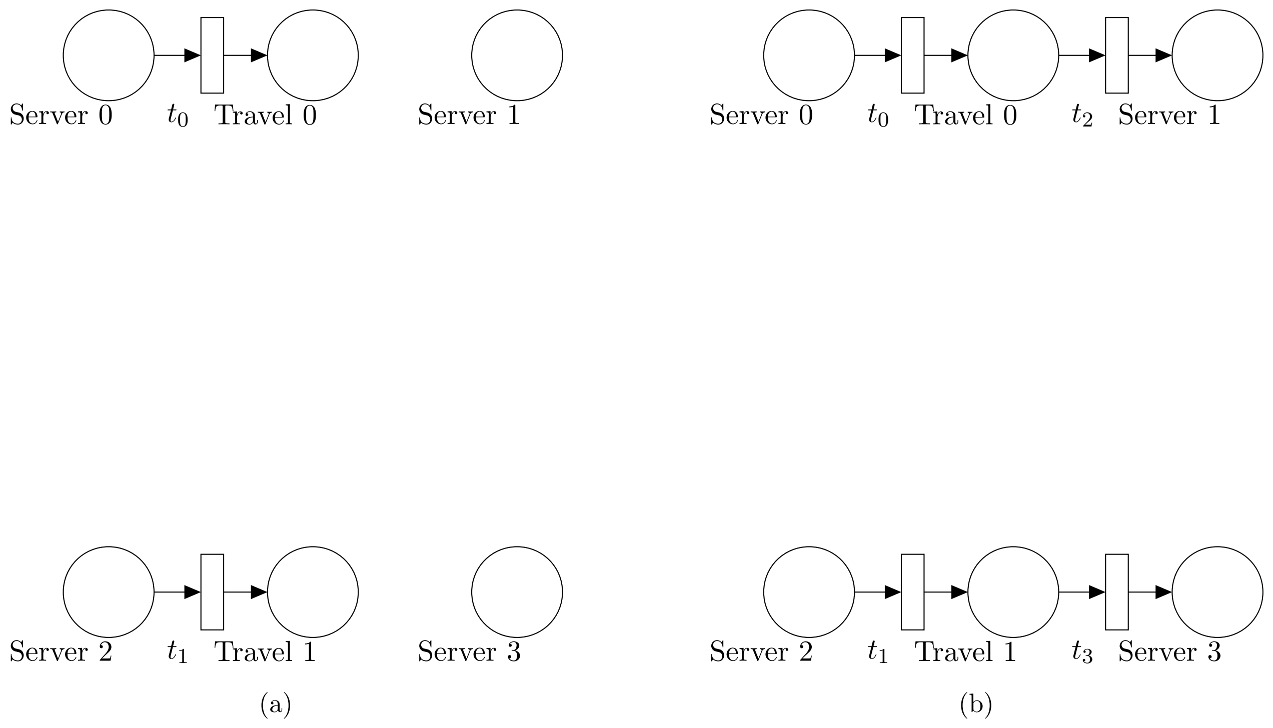


Figure 4.10: Addition of travel places between pairs of places representing service areas as well as transitions. Figure 4.10(a) shows the addition of service time transitions t_0, t_1 and Figure 4.10(b) the addition of travelling time transitions t_2, t_3.

One of the key challenges in this stage is to distinguish the cases where a customer simply passes through a service area without requesting service. To spot these cases, we use a speed-based heuristic tailored for each particular customer path. We compute the average speed of the customer over a time window before it reaches a service area and compare it with the customer's average speed inside the service area. If the latter is less than the former we consider the customer as having waited to be serviced by the server. A look-ahead action similar to [57] is also employed, in order to judge whether a real departure event occurred by checking if the departed customer returns to the service area within a short period of time; this also removes issues caused by possible erroneous location updates which might indicate that the customer's location is briefly outside the service area while, in reality, this is not the case.

Finally, a simple counting mechanism is used to calculate the initial routing probabilities of the customer flow structure. These are represented as immediate transitions in the resulting PNPM with weight equal to the probability for each corresponding destination service area. Several other places and transitions are also used in our model (see Figure 4.12):

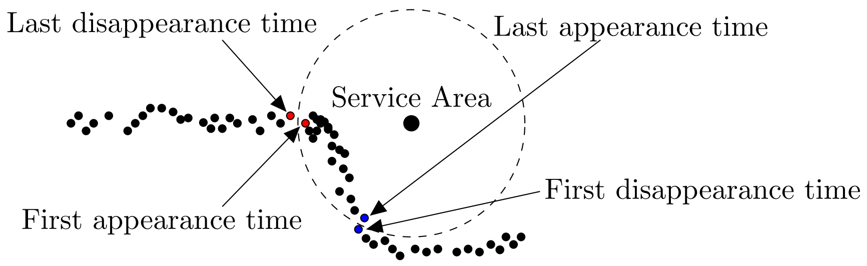


Figure 4.11: An example of a customer path, as the customer passes through a service area. Red dots represent the location updates whose timestamps are used to compute the customer's entry time. Similarly, blue dots denote the location updates whose timestamps are used to compute the customer's exit time.

- A repository place. This place holds the total number of customers to be processed by the system (represented as tokens).
- A place labelled as "Arrived" which is connected to the repository place via a timed transition. The transition is used to emulate the travelling delay of customers before reaching the first service area.
- Intermediate places connected to some service areas' service time transitions and to immediate transitions. These immediate transitions have weight equal to one and restore the number of tokens contained in the repository. Such intermediate places are created when customers exit the system immediately after obtaining service from a service area.

4.4 Stage 4

The final stage aims to replace the service area service time and travelling time transitions with GSPN subnets that accurately reflect the distributions of the corresponding service time and travelling time samples collected in Stage 3. For this task we will use the hyper-Erlang distribution (HErD).

As discussed in Section 2.6.3 a HErD, given a sufficient number of phases, can be used to approximate any non-negative type distribution. Here, the main issue is the accurate representation of the HErD in terms of GSPNs, since they only support transitions which fire with an exponentially distributed time delay. As illustrated in Figure 4.13, the places and transitions in

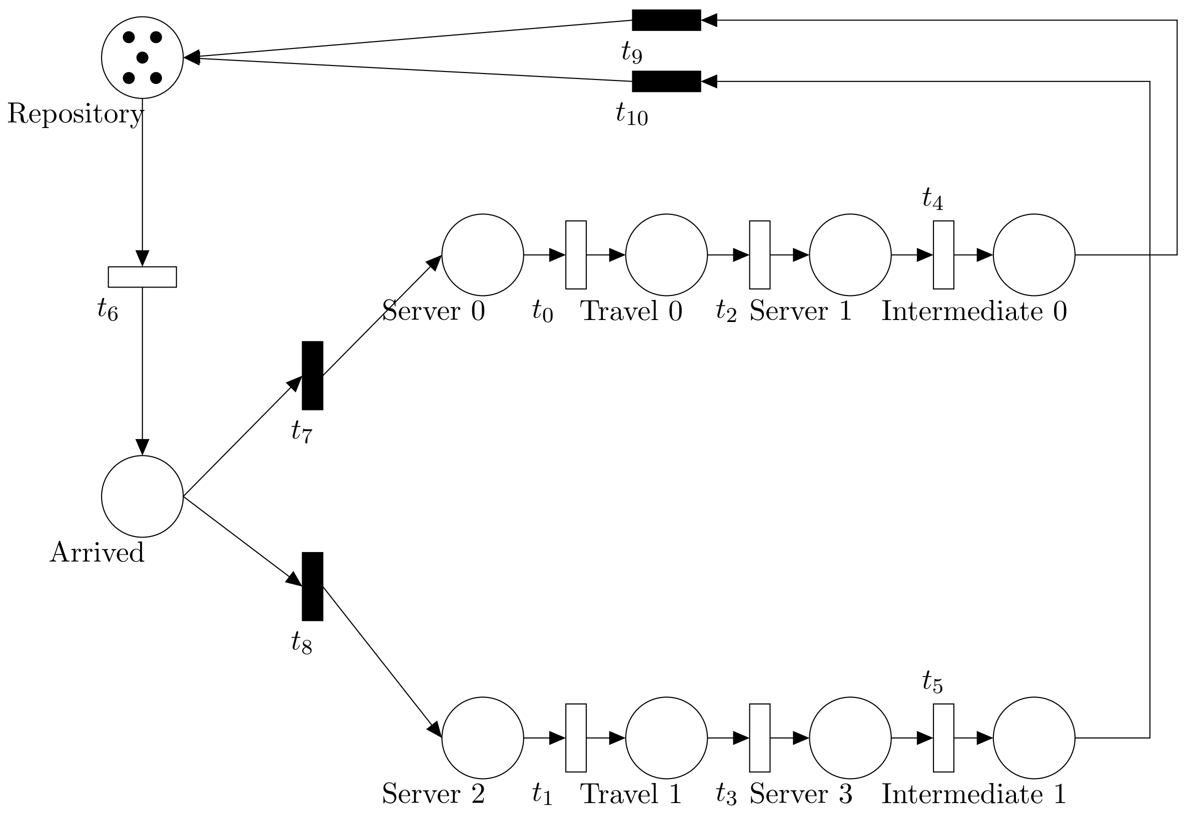


Figure 4.12: A complete overview of the initial structure of a PNPM model obtained at the end of the third stage of the processing pipeline for the set of location traces depicted in Figure 4.2. The immediate transitions t_7 and t_8 model the initial routing probability of the customer flow.

the GSPN subnets correspond to the phases of the HErD which best fits the extracted timing samples.

The weights of the Erlang branches are represented by the immediate transitions $t_1, \ldots, t_M$, and each Erlang branch is modelled using Molloy's representation [79, 32], i.e. each phase of the Erlang distribution is modelled by a timed transition. The immediate transitions $t_{inst_1}, \ldots, t_{inst_M}$ have weight equal to one and are simply used to transfer tokens out of the subnet. When the GSPN subnet replaces a service time transition, places p_0 and p_1 correspond to places representing service areas and customer movement respectively. If a travelling time transition is replaced by the subnet then p_0 corresponds to a place which represents customer movement and p_1 to a place which represents a service area.

We assume infinite-server semantics for all timed transitions included in the subnet. Therefore, in the representation of the service time transitions we employ an additional complementary place to control the number of tokens (customers) that are allowed to be inside the subnet

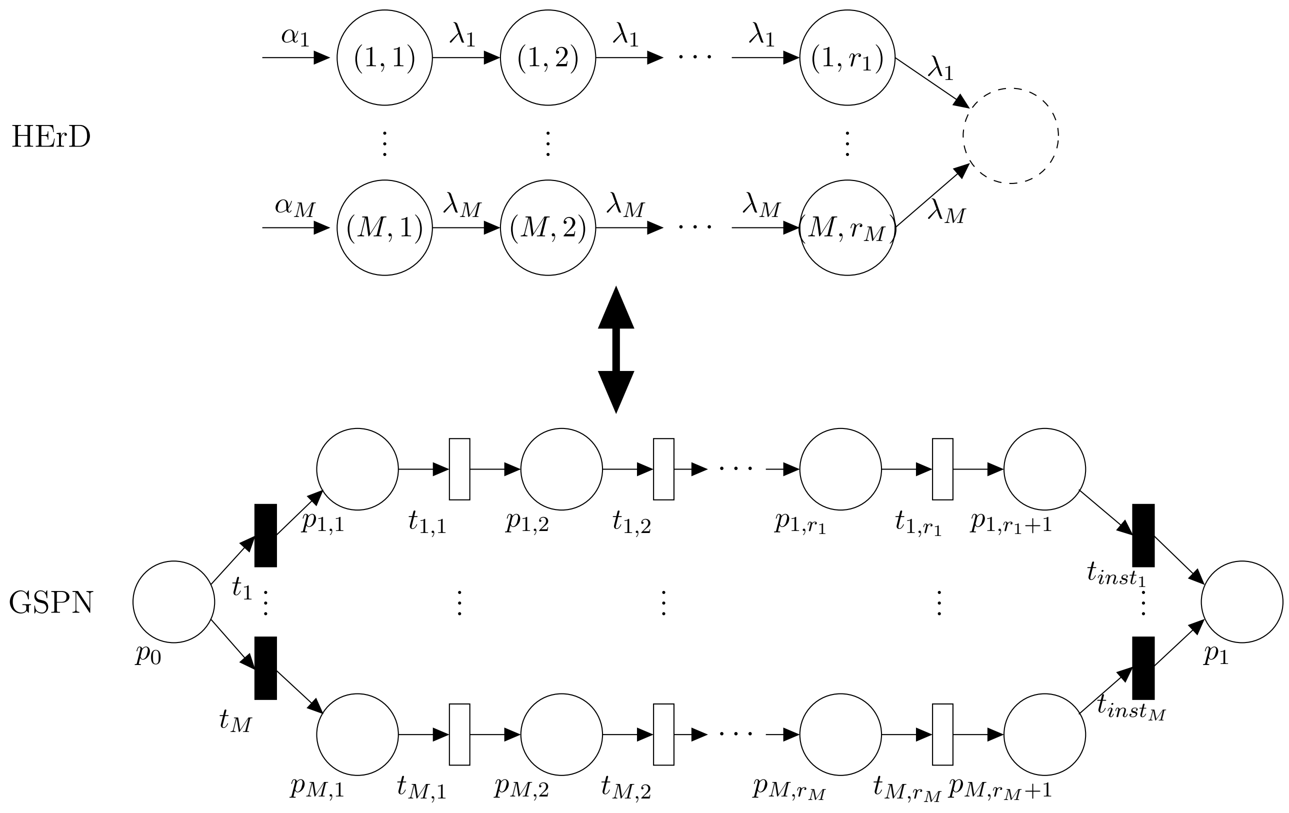


Figure 4.13: HErD to GSPN correspondence.

simultaneously. In this research, we assume service areas with single-server semantics and therefore the initial marking of this place is one (see Figure 4.14). There is no such restriction on travelling time transitions.

To compute the candidate best-fit HErDs from our extracted timing samples, we make use of the G-FIT tool [106]. We perform an exhaustive enumeration of all possible HErDs up to a maximum number of states; we run G-FIT repeatedly, increasing the maximum allowed number of states. At each run G-FIT returns the parameters of the best-fit HErD, i.e. the number of Erlang branches, their weights, the number of phases and the rate for each branch. We have also modified G-FIT to return the log-likelihood value of the selected fit. This information is stored in a list, in respective order for each run, and is used in conjunction with the Akaike Information Criterion (AIC) to select the best-fit HErD; the one which achieves the lowest score under the AIC (this is computed using equation 2.23 which is appropriate for small samples – cf. Section 2.6.4). If AIC was not used, the chosen HErD would always be the one having the maximum number of states, and thus we risk over-fitting the data.

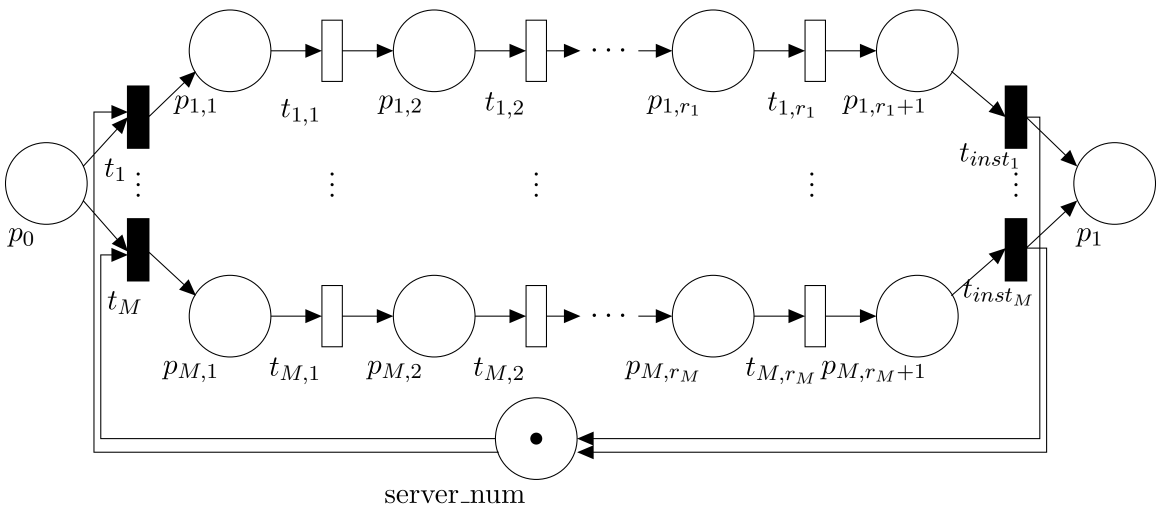


Figure 4.14: GSPN subnet representation for service time transitions.

The maximum number of states for the HErD to be fitted is set equal to ten, i.e. $N = 10$, for all cases when the coefficient of variation of the extracted service or travelling time sample is greater than 0.4 and twenty five, i.e. $N = 25$, when it is less. This action is taken to allow for better approximations of very low-variance distributions to be obtained. In fact we expect that in such cases the best-fit HErD has only one branch and k states, $10 < k \leq 25$, where each state has rate λ, for some λ. We conjecture that twenty five states are sufficient to achieve relatively good approximations for such low-variance distributions with regard to the overhead of fitting a HErD with more states. Nevertheless, we allow the user to increase the maximum number of states if the computational cost – both in terms of the fitting process and the subsequent GSPN model analysis[4] – is not an issue.

4.5 Evaluation

In this section we present two case studies that were conducted in order to assess the applicability and accuracy of our proposed approach. We also discuss limitations of the developed pipeline. For the two case studies we have generated location tracking data using an extended version of the simulator JINQS [43], namely LocTrackJINQS [56] (see Chapter 3). The use of synthetic data provides two advantages over real traces: it allows us to characterise the

[4]The larger the maximum number of states is, the larger the state space of the underlying semi-Markov process will be.

degree of accuracy of the inferred distributions and their parameters – as the exact model parameters and processes are known – and their generation is performed in a time efficient manner rather than engaging in long experimental procedures. The philosophy behind our evaluation process for these, as well as for the subsequent case studies, is to formulate and parameterise an abstract system, model and simulate it in LocTrackJINQS, and then process the simulation's output using our methodology. The inferred GSPN performance model is considered to be accurate if its structure and parameters approximate those of the abstract system. A graphical representation of this evaluation process is shown in Figure 4.15.

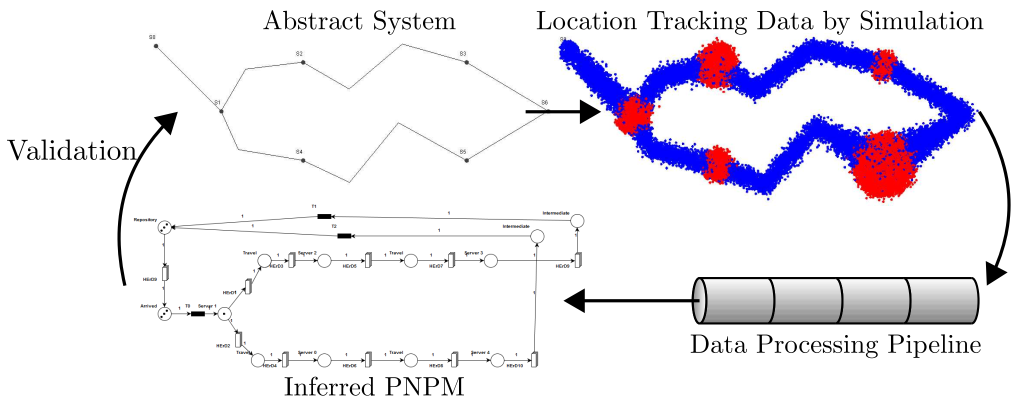


Figure 4.15: Graphical representation of our evaluation process.

4.5.1 Case Studies

We focus on the service area inference (second stage of the pipeline) and the extraction of the service area service time distributions (fourth stage). The experimental setup for each case study is depicted in Figure 4.16. The simulations take place in a virtual $25\,\text{m} \times 25\,\text{m}$ environment with customer movements as illustrated. Customers are assumed to travel between service areas in such a way that each journey has a speed drawn from a normal distribution with mean 0.5 m/s and standard deviation 0.1 m/s for case study 1, and a speed drawn from a normal distribution with mean 0.3 m/s and standard deviation 0.1 m/s for case study 2. The location update error is normally distributed with mean 0.15 m and standard deviation 0.2 m.

Each service area consists of a single customer-processing server and a random customer service discipline. The service time for each service area follows a different density function. Table 4.2

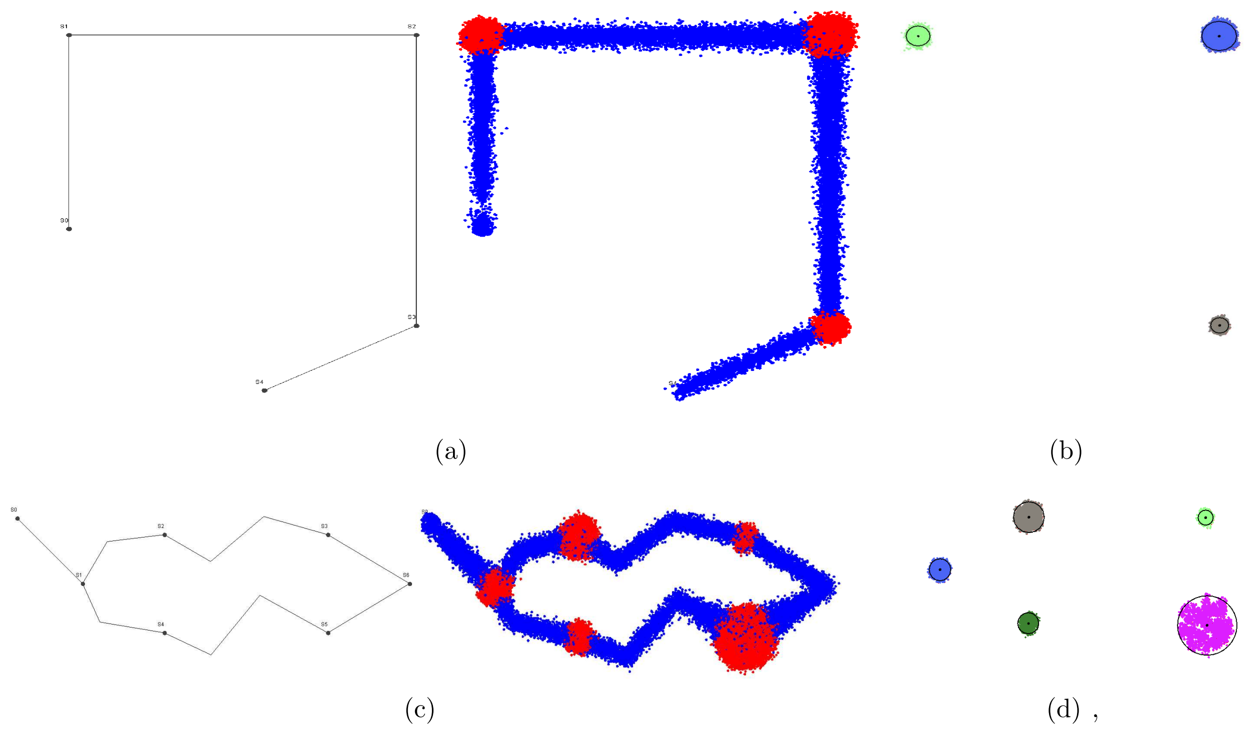


Figure 4.16: The experimental setup in terms of abstract system structure and generated location tracking data (a), (c) and clustering results (b), (d) of the first (above) and second (below) case studies. Blue and red traces indicate customer movement and stationarity respectively.

shows each service area's actual location and service radius as well as its service time density for each case study.

4.5.2 Results

Figures 4.16(b) and 4.16(d) show the results of the second stage of our processing pipeline. Table 4.3 displays the estimated location and service radius for each service area as well as the error between these and their real values (in terms of the distance between the real and inferred points). From these results we can see that the inferred location matches the real location almost perfectly with a maximum error of 0.079 metres for the two case studies. The fitted radius for each service area has larger error in general (max 0.128 metres).

For the purpose of evaluating the service time sample extraction and HErD fitting we compute the relative entropy between the theoretical and fitted service time distribution, shown in Table 4.4. We also conduct a Kolmogorov–Smirnov test, examining compatibility of the extracted service time samples for each service point with its best-fit HErD (see Table 4.5).

		Server Location	Service Radius	Service Time Density
Case Study 1	S1	(2.0,2.0)	0.5	HErD(2,2;0.5,0.5;0.05,0.48)
	S2	(18.0,2.0)	0.8	Erlang(3,0.065)
	S3	(18.0,20.0)	0.35	Exp(0.03)
Case Study 2	S1	(5.0,5.0)	0.5	HErD(2,2,4;0.3,0.3,0.4;0.05,0.25,0.6)
	S2	(10.0,2.0)	0.75	Erlang(4,0.12)
	S3	(20.0,2.0)	0.3	Normal(35.5,6.5)
	S4	(10.0,8.0)	0.45	Exp(0.025)
	S5	(20.0,8.0)	1.5	Hyper-Exp(0.4,0.3,0.3;0.04,0.1,0.01)

Table 4.2: The parameters for each service area in the system, for each case study. Server locations and their corresponding service radii are given in metres. The parameters of the HErDs represent the phase lengths, weights and rate for each branch respectively, separated by a semi-colon. The parameters of the hyper-exponential distribution represent the weights and rates of each exponential distribution respectively, separated by a semi-colon. The unit of the rate parameters for phase-type distributions is customers per second. The mean and standard deviation of the Normal distribution are given in seconds.

		Server Location			Service Radius		
		Real	Inferred (3 d.p.)	Error (3 d.p.)	Real	Inferred (3 d.p.)	Absolute Error
Case Study 1	S1	(2.0,2.0)	(1.996,1.987)	0.014	0.5	0.567	0.067
	S2	(18.0,2.0)	(18.024,2.029)	0.038	0.8	0.896	0.096
	S3	(18.0,20.0)	(17.966,19.983)	0.038	0.35	0.428	0.078
Case Study 2	S1	(5.0,5.0)	(5.031,5.016)	0.034	0.5	0.566	0.066
	S2	(10.0,2.0)	(9.921,1.992)	0.079	0.75	0.846	0.096
	S3	(20.0,2.0)	(20.015,2.008)	0.017	0.3	0.378	0.078
	S4	(10.0,8.0)	(9.983,8.004)	0.017	0.45	0.519	0.069
	S5	(20.0,8.0)	(19.946,7.962)	0.066	1.5	1.628	0.128

Table 4.3: The inferred location and service radius for each service area in the system accompanied with the absolute error, for each case study. These values are given in metres.

Although the approximation of the actual service time density by the best-fit HErD is very good in all cases (see Figures 4.19 and 4.20), the parameters of the best-fit HErD do not match the

		Service Time Density	Fitted HErD Parameters			Relative Entropy
			Phase Lengths	Rate (3 d.p.)	Weights (3 d.p.)	(3 d.p.)
Case Study 1	S1	HErD(2,2; 0.5,0.5; 0.05,0.48)	1,2	0.038,0.600	0.744,0.256	0.029
	S2	Erlang(3,0.065)	3,7	0.065,4.055	0.991,0.009	0.007
	S3	Exp(0.03)	1	0.030	1.000	0.000
Case Study 2	S1	HErD(2,2,4;0.3,0.3,0.4;0.05,0.25,0.6)	2,2,4	0.048,1.699,0.0469	0.295,0.084,0.622	0.049
	S2	Erlang(4,0.12)	5,5	0.139,0.514	0.953,0.047	0.011
	S3	Normal(35.5,6.5)	25	0.666	1.000	0.055
	S4	Exp(0.025)	1,3	0.022,0.144	0.785,0.215	0.012
	S5	Hyper-Exp(0.4,0.3,0.3;0.04,0.1,0.01)	1,1	0.010,0.049	0.315,0.685	0.010

Table 4.4: The HErD parameters fitted by G-FIT for each service area's service time density with the relative entropy (in nat) between the theoretical and fitted probability density function, for each case study. The parameters of the HErDs represent the phase lengths, weights and rate for each branch respectively, separated by a semi-colon.

		Case Study 1		Case Study 2	
S1	Test Statistic	0.0263		0.0194	
	α	0.1	0.05	0.1	0.05
	Critical Values	0.0494	0.0563	0.0388	0.0442
	Compatible ?	Yes	Yes	Yes	Yes
S2	Test Statistic	0.0420		0.0402	
	α	0.1	0.05	0.1	0.05
	Critical Values	0.0730	0.0832	0.0552	0.0629
	Compatible ?	Yes	Yes	Yes	Yes
S3	Test Statistic	0.0470		0.0442	
	α	0.1	0.05	0.1	0.05
	Critical Values	0.0757	0.0863	0.0558	0.0636
	Compatible ?	Yes	Yes	Yes	Yes
S4	Test Statistic	N/A		0.0285	
	α			0.1	0.05
	Critical Values			0.0579	0.0661
	Compatible ?			Yes	Yes
S5	Test Statistic	N/A		0.0333	
	α			0.1	0.05
	Critical Values			0.0613	0.0699
	Compatible ?			Yes	Yes

Table 4.5: Kolmogorov-Smirnov test at significance levels 0.1 and 0.05 applied to the extracted service time samples for each service point from case studies one and two. The null hypothesis is that each extracted sample belongs to the corresponding best-fitted HErD.

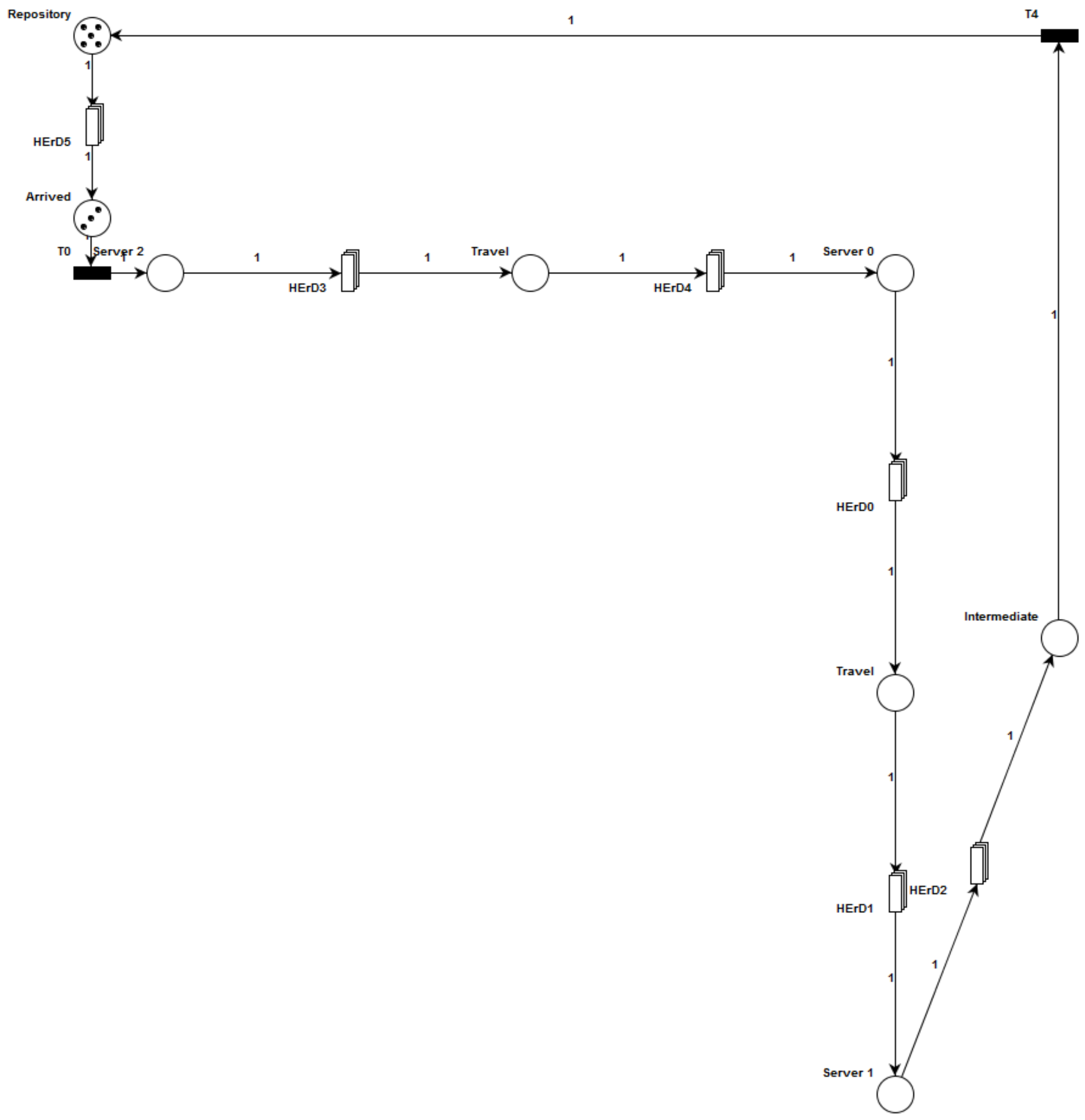


Figure 4.17: The inferred GSPN performance model for case study 1, visualised in PIPE2 (in compact transition form).

parameters of the actual density in every case. For example, if we consider S2 in the first case study we see that the best fitted HErD has an additional branch with seven phases and rate equal to 4.055. Its weight though is only 0.009 and this suggests that the contribution from this branch is not of great significance. A similar case is also observed for S4 (case study 2) where an additional branch with three phases is allocated to the fitted hyper-Erlang distribution; however, in this example the weight of the additional branch is 0.215. Furthermore, the HErD approximation of the normally distributed service time (S3 – case study 2) is as expected:

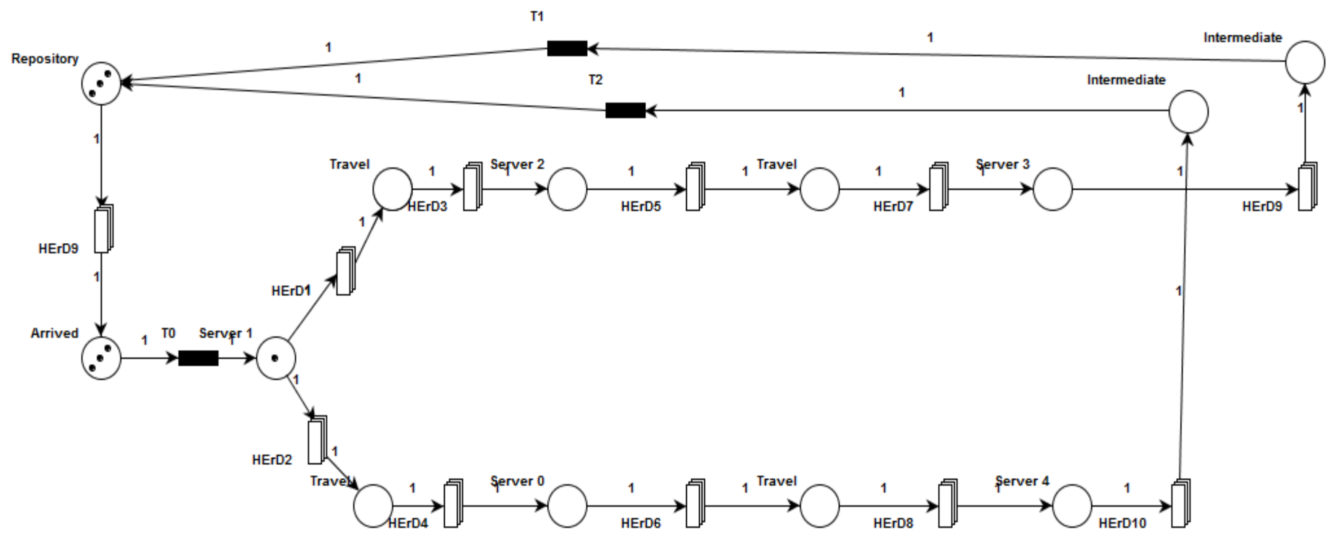
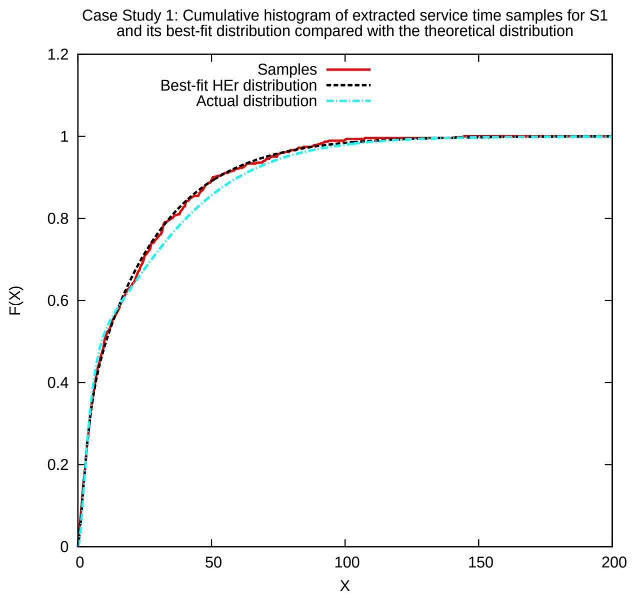


Figure 4.18: The inferred GSPN performance model for case study 2, visualised in PIPE2 (in compact transition form).

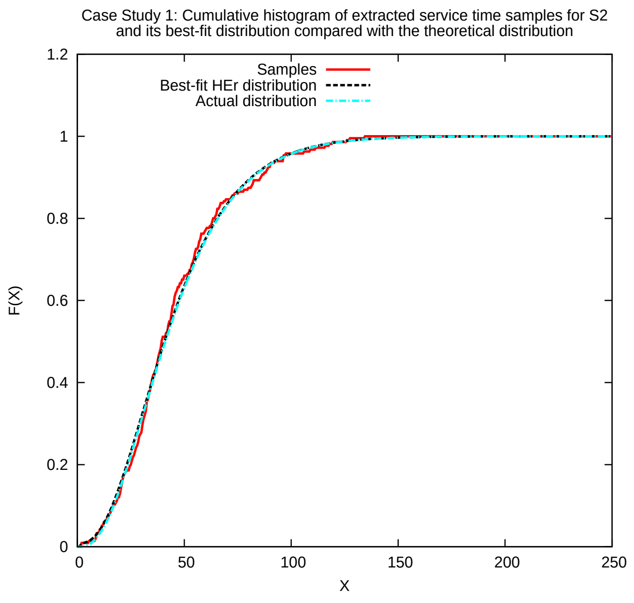
one branch with a sufficiently large number of phases. In particular, by the Central Limit Theorem (CLT) [4], the fitted HErD approximates a normal distribution with mean 37.538 s and standard deviation 7.508 s.

Figures 4.17 and 4.18 illustrate the inferred PNPMs for both case studies. We observe that the structure of both models matches the abstract system structure of the models used to set up the simulation, i.e. for each service area there exists a corresponding place and an associated subnet – in compact transition form – which models the service area's service time. In particular, server places with labels Server 2, Server 0 and Server 1 correspond to S1, S2 and S3 in the first case study. Similarly, for the second case study, places Server 1, Server 2, Server 0, Server 3 and Server 4 correspond to S1, S2, S4, S3 and S5. Also, between each pair of places representing service areas we can see intermediate travel places and their associated travelling time subnets.

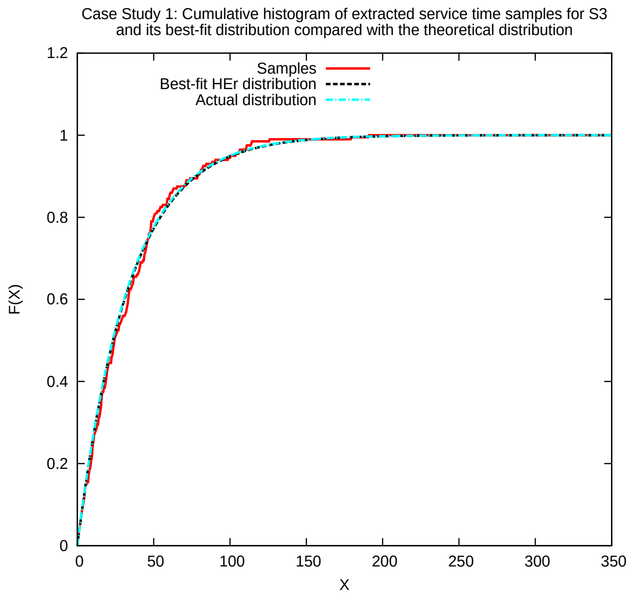
As it was briefly discussed in the introduction of this thesis, the constructed GSPN performance models can be used to analyse the underlying system's performance through the computation of end-to-end response time distributions [39, 52]. This kind of analysis is supported in PIPE2 via the DNAmaca analysis module [70]. In addition, one can modify the the model's workload (by adding/removing tokens in the Repository place) or the model's resources (duplicate existing server places along with their corresponding GSPN subnets that reflect their service delay) to examine the system's performance under hypothetical scenarios ("what-if" analysis).



(a)



(b)

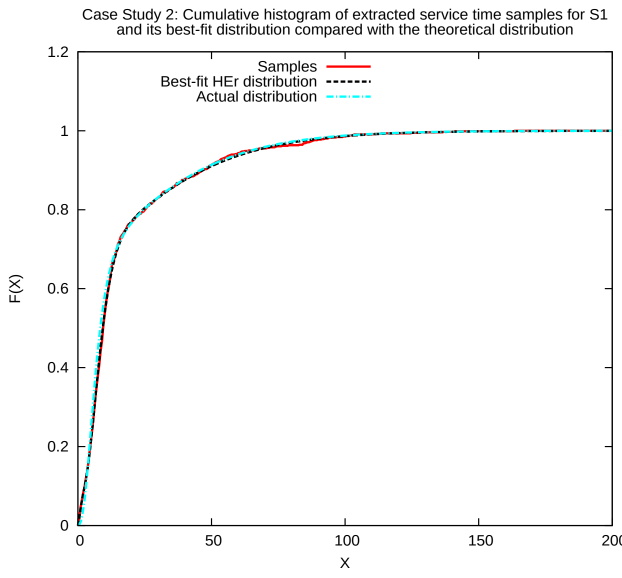


(c)

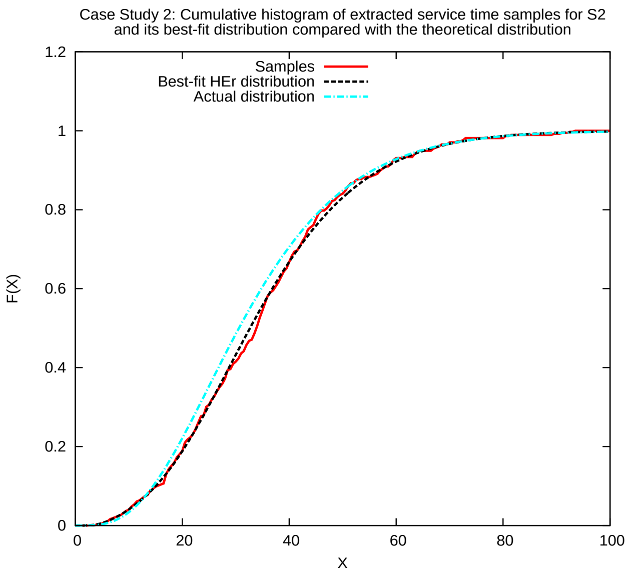
Figure 4.19: Case study 1: Graphs 4.19(a), 4.19(b) and 4.19(c) show the cumulative histogram of the extracted service time samples and its best-fit hyper-Erlang distribution compared with the theoretical distribution for S1, S2 and S3 respectively.

4.5.3 Conclusion

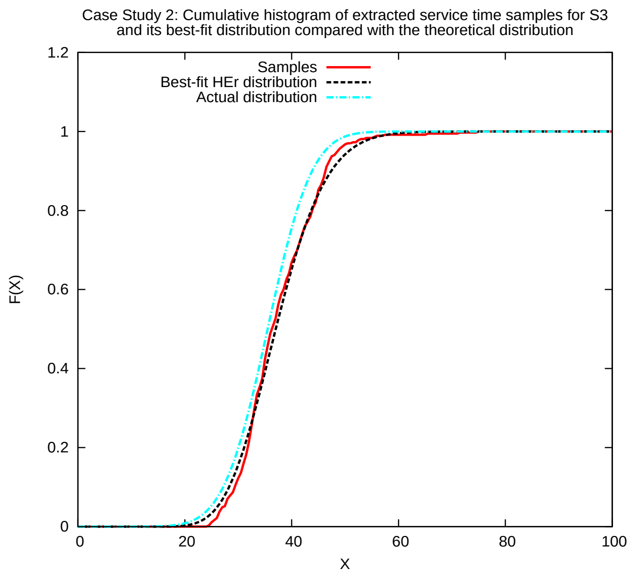
In this chapter we have presented a data processing pipeline which successfully derives PNPMs from high-precision location tracking data collected from a class of simple customer-processing systems.



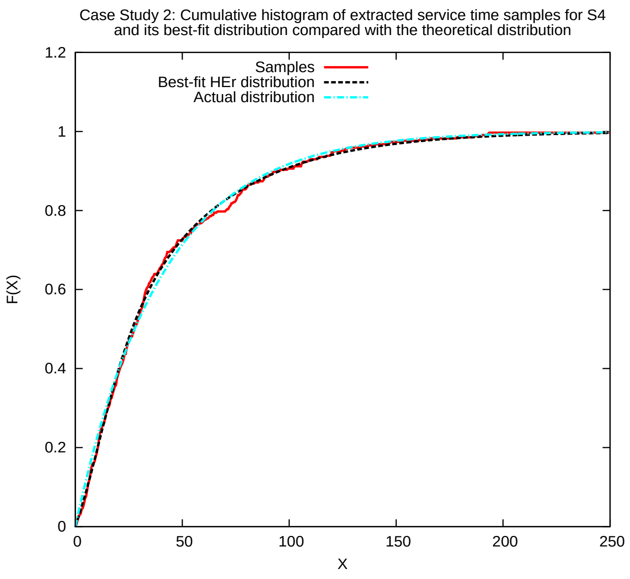
(a)



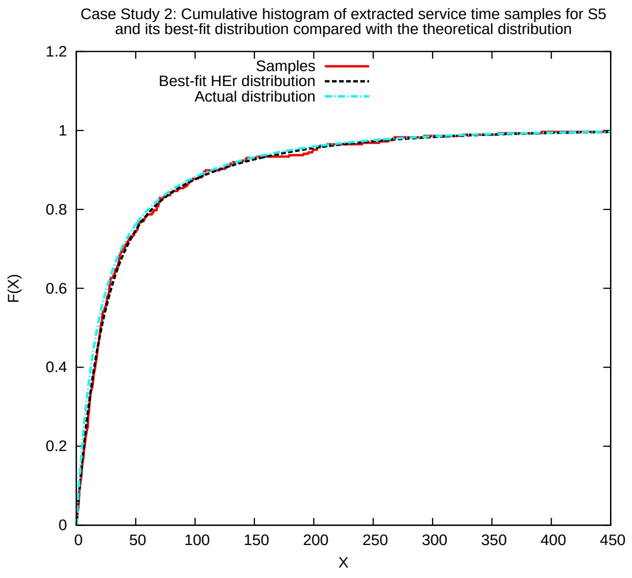
(b)



(c)



(d)



(e)

Figure 4.20: Case study 2: Graphs 4.20(a), 4.20(b), 4.20(c), 4.20(d) and 4.20(e) show the cumulative histogram of the extracted service time samples and its best-fit hyper-Erlang distribution compared with the theoretical distribution for S1, S2, S3, S4 and S5 respectively.

The first stage of the pipeline prepares the raw location traces for processing by the subsequent stages. Stage two employs a variety of well-known and established techniques to infer the location and radii of stationary service areas in the system. Stage three constructs the initial non-parameterised model from the processed location tracking traces and extracts the observed service and travelling time samples. The fourth stage of the pipeline uses the G-FIT tool to fit matching HEr distributions to the extracted time samples and accurately represents them in terms of GPSNs. Their incorporation in the initial non-parameterised model results in a hierarchical GSPN model compatible with PIPE2, an open source Petri Net editor.

The results of the two case studies indicate that the developed approach has the potential to infer the stochastic features of simple systems accurately, at least when synthetically-generated location tracking data is used.

Currently our approach has made several assumptions such as a single class of customers, single-server semantics and random service discipline. Also, in real customer processing systems it is very often the case that the service of customers is subject to some synchronisation conditions. The current methodology lacks the possibility of synchronisation inference. In particular, we would like to provide mechanisms that allow the accurate representation of multiple customer classes and synchronisation specified in terms of the physical presence of customers at different service areas. These two features often arise in real customer-processing systems and it is critical to incorporate them in the data processing pipeline so that it can be applied in more realistic scenarios. This task is tackled in the following two chapters.

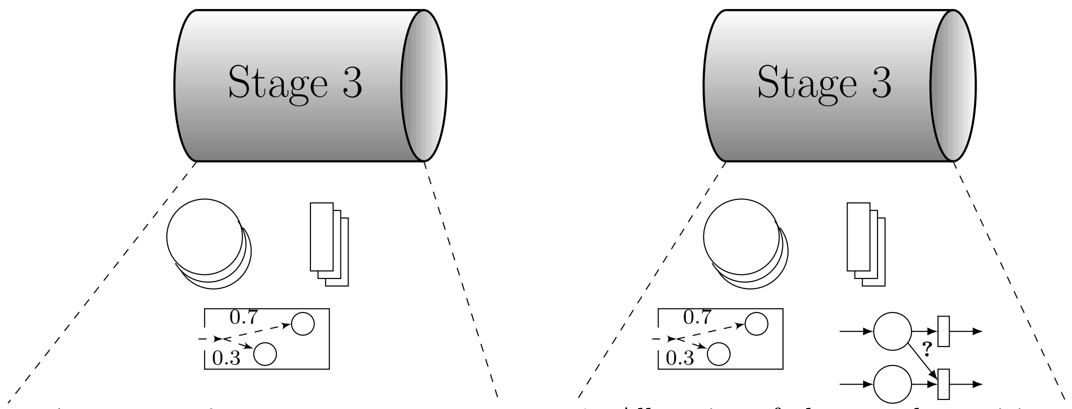
Chapter 5

Synchronisation Detection and Representation

In the previous chapter we presented a methodology that is able to automatically construct Generalised Stochastic Petri Net [14, 78] performance models from raw location tracking data. However, synchronisation between service centres – the natural expression of which is one of the most fundamental advantages of Petri Nets as a modelling formalism – was not explicitly captured. This is potentially a serious deficiency since many physical customer-processing systems, such as hospitals, airports and car assembly lines, exhibit many instances of synchronisation. For example, if we consider a treatment room in a hospital, the examination of a patient cannot take place without the presence of a doctor.

Here, we introduce a mechanism for automatically detecting presence-based synchronisation from location tracking traces. In other words, synchronisation is detected only if it is defined in terms of the physical presence of customers or resources in service areas. Of course, as we discussed in the introduction of Chapter 4, the constructed PNPM depends on the collected location tracking traces and in particular, on the period during which the customer-processing system was monitored. This also presents an effect on the synchronisation detection mechanism introduced in this chapter. Specifically, if the flow of customers in the system is monitored during periods of heavy customer traffic or more fine-grained sampling is employed additional synchronisation conditions may be exposed.

This chapter also illustrates how this mechanism is incorporated into the existing methodology (third stage of the processing pipeline – see Figure 5.1) and presents the modifications that are required to be made to the structure of the constructed PNPMs to enable the accurate representation of synchronisation (fourth stage of the processing pipeline). We conclude this chapter with a case study which demonstrates and evaluates the modified data processing pipeline.



1. Allocation of places and transitions
2. Sojourn and travelling time sample extraction

   service time estimation
3. Calculation of initial routing probabilities

1. Allocation of places and transitions
2. Sojourn and travelling time sample extraction

   service time estimation
3. Synchronisation detection

   service time re-adjustment
4. Calculation of initial routing probabilities

Figure 5.1: A high-level description of the third stage of the processing pipeline before (left) and after (right) the incorporation of the synchronisation detection mechanism.

5.1 Synchronisation Detection Mechanism

Our aim is to construct a conservative scheme to determine whether the processing of customers at each service area is subject to some synchronisation conditions (expressed as conditions on the number of customers present within other service areas) at certain time points. To this end, we have designed three functions (see Algorithms 6, 7 and 8) that can be applied together to perform the synchronisation detection task.

To formalise our approach, we introduce some notation:

- N is the total number of service areas inferred from the second stage of the processing pipeline,
- $P = \{P_1, P_2, \ldots, P_N\}$ is the set of inferred services areas (subsequently represented by places in the derived Petri Net model),
- $C_i = \{c_i^{(1)}, c_i^{(2)}, \ldots, c_i^{(n_i)}\}$ is the multiset[1] of all customers processed by P_i, with $n_i = |C_i|$,
- $e_i^{(j)}$ is the timestamp of the entry of $c_i^{(j)}$ into P_i,
- $s_i^{(j)}$ is the service initiation timestamp of $c_i^{(j)}$, and
- $f_i^{(j)}$ is the service termination timestamp of $c_i^{(j)}$.
- $M_i(t)$ is the number of customers present on service area P_i at timestamp t. This is also referred to as the *marking* of P_i at time t.
- $M_i(t_1, t_2)$ is the maximum number of customers observed on service area P_i during the time interval $[t_1, t_2)$. This is also referred to as the *maximum marking* of P_i during the time interval $[t_1, t_2)$. $M_i(t_1, t_2)$ and $M_i(t_2)$ are the two components of the evidence we require to (possibly) infer the synchronisation conditions between service areas. This is explained in more detail in the remainder of this section.

Given a service area P_i, we consider the processing of each customer that receives service there in turn. Our approach is based on finding evidence – for each customer – that its processing may have been dependent on the presence of customers on other service areas. This evidence has two components: one is the maximum marking observed on each of the other service areas in the interval during the customer was serviced; the other is the marking observed on each of the other service areas at the instant of termination of the customer's service. These

[1]Thus supporting the possibility of multiple service periods for the same customer.

two components are combined across all customers processed by P_i – taking into account the possibility of error and noise – to yield the likely synchronisation conditions of service at P_i[2]. Formally:

Definition 5.1. *The* j*th customer* $c_i^{(j)} \in C_i$ *is said to receive service at* P_i *with possible synchronisation from each service area* $P_k, k = 1, \ldots, N$ *and* $k \neq i$*, if:*

$$M_k(s_i^{(j)}, f_i^{(j)}) > 0, \tag{5.1}$$

and

$$M_k(f_i^{(j)}) > 0 \tag{5.2}$$

Definition 5.2. *Synchronisation between service area* P_i *and service area(s)* $P_k, k = 1, \ldots, N$ *and* $k \neq i$*, is inferred if the synchronisation percentage* s_p *defined as,*

$$s_p(i,k) = \frac{|\{c_i^{(j)} \mid M_k(s_i^{(j)}, f_i^{(j)}) > 0, M_k(f_i^{(j)}) > 0,\ j = 1, \ldots, n_i\}|}{n_i} \tag{5.3}$$

satisfies

$$s_p(i,k) \geq s_{thresh} \tag{5.4}$$

where s_{thresh} *is the hypothesis acceptance threshold.*

The value of the acceptance threshold (typically in the range $[0.8, 1]$) can be chosen according to factors such as the precision of the location tracking system used, the tag update rate and topology of the system being modelled. Of course, in systems with high traffic (even with the synchronisation acceptance threshold set equal to one) this synchronisation detection mechanism is susceptible to false positives. Nevertheless, other techniques, e.g. the discretisation of the service time interval of a customer into several others and the examination of the two conditions specified in Definition 5.1 for each subinterval, would perform similarly under high traffic or worse (increase the number of false positives) under lower workloads.

[2]Here we assume a single class of customers. It is straightforward to apply the combination across each customer class in a scenario with multiple customer classes.

5.2 Algorithm Description

In this section, we present and describe the three functions we have developed to perform the synchronisation detection task.

The first algorithm (see Algorithm 6) implements the $M_k(t)$ method which returns the number of customers that are present on service area P_k at timestamp t. A counter variable is initialised to zero and the method iteratively examines – for all customers processed by P_k – if each customer was present at timestamp t. In this case the counter variable is incremented.

Algorithm 6 : `M(`k`,`t`) : int`

```
1: marking ← 0
2: for $j = 1$ to $n_k$ do
3:   if $c_k^{(j)}$ was present in $P_k$ at $t$ then
4:     marking ← marking + 1
5:   end if
6: end for
7: return marking
```

The second algorithm (see Algorithm 7) implements the $M_k(t_1, t_2)$ method which returns the maximum number of customers that were present on service area P_k during some time interval $[t_1, t_2)$. Here, we use the $M_k(t)$ method to obtain the marking of the service area at certain timestamps. First, we initialise a counter variable (`maxMarking`) to the marking of the service area at timestamp t_s, i.e. the service initiation timestamp of some customer who is present at some other service area. Then we check – for each customer processed by P_k – if the customer's arrival in P_k occurred within the time interval (t_s, t_f), i.e. $t_s < e_k^{(j)} < t_f$. If this is the case then

Algorithm 7 : `M(`k`,`t_s`,`t_f`) : int`

```
1: maxMarking ← M($k$,$t_s$)
2: for $j = 1$ to $n_k$ do
3:   if $t_s < e_k^{(j)} < t_f$ then
4:     instMarking ←M($k$, $e_k^{(j)}$)
5:     if instMarking > maxMarking then
6:       maxMarking ← instMarking
7:     end if
8:   end if
9: end for
10: return maxMarking
```

we compute the instantaneous marking of P_k at timestamp $e_k^{(j)}$ and check if its value is greater than the value of `maxMarking`. When the latter condition is satisfied, we update the counter variable to the value of the instantaneous marking.

Algorithm 8 : `computeSynchronisation(`i`,`s_{thresh}`) : int[`N`]`

```
 1: customersWithSynch ← new int[$N$] = {0,0,...,0}
 2: synchMarking ← new int[$N$] = {0,0,...,0}
 3: csmMatrix ← new int[$n_i$][$N$] = { {0,0,...,0}, {0,0,...,0}, ..., {0,0,...,0} }
 4: for $j = 1$ to $n_i$ do
 5:   for $k = 1$ to $N$ do
 6:     if $k != i$ then
 7:       csmMatrix[$j$][$k$] ← min(M($k, s_i^{(j)}$,$f_i^{(j)}$),M($k$,$f_i^{(j)}$))
 8:       if csmMatrix[$j$][$k$] > 0 then
 9:         customersWithSynch[$k$] ← customersWithSynch[$k$] + 1
10:       end if
11:     end if
12:   end for
13: end for
14: for $k = 1$ to $N$ do
15:   if $k != i$ then
16:     if customersWithSynch[$k$] / $n_i \geq s_{thresh}$ then
17:       synchMarking[$k$] ← percentile({csmMatrix[1][$k$],...,csmMatrix[$n_i$][$k$]},5)
18:     end if
19:   end if
20: end for
21: return synchMarking
```

The previous two Algorithms introduce auxiliary functions that are used in the main function, `computeSynchronisation` (see Algorithm 8), which is applied in turn to every service area $P_i \in P$. There are two phases in this algorithm. In the first phase (see lines 4 to 13) we construct the `csmMatrix`, which describes the possible set of synchronisation dependencies between the service of customer j at P_i, and the markings of the other service areas. In this phase, for each P_k, $i \neq k$, we also count the number of customers for which some potential synchronisation was observed, during their service at P_i. In the second phase (see lines 14 to 20) we calculate $s_p(i,k)$ and if its value exceeds the value of s_{thresh} we compute the synchronisation marking on P_k required to support service at P_i. This is computed as a low percentile of the set of synchronisation markings; this is preferred to simply taking the minimum because it is more robust to measurement errors inherent in location tracking systems. This percentile is determined by the function `percentile(`M`,`α`)` (see line 17, Algorithm 8) which computes the

αth percentile of the set M.

If we assume that $\forall i,\ n_i = n$, then the worst-case time complexity of the main function `computeSynchronisation` and synchronisation detection for the entire network are $O(N \cdot n^3)$ and $O(N^2 \cdot n^3)$ respectively. Based on the same assumption, space complexity is bounded above by the size of `csmMatrix` (see Algorithm 8) and is $O(N \cdot n)$.

5.3 Service Time Adjustment

Whenever synchronisation is detected involving the processing of customers at P_i, the corresponding service time samples of those customers need to be adjusted to take into account the proportion of time during which the synchronisation condition(s) is(are) satisfied. This is because we assume that service only progresses when the synchronisation condition(s) is(are) met. To perform the service adjustment process we developed a function which examines the synchronisation conditions only at the timestamps of arrival and departure events of the customers instead of the entire pre-adjusted service time interval (see Figure 5.2).

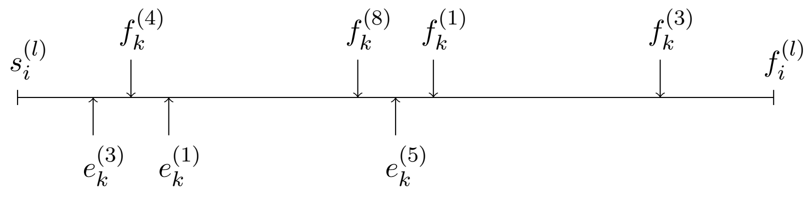


Figure 5.2: Three arrival and four departure events occurring at service area P_k during the lth customer's service time interval $[s_i^{(l)}, f_i^{(l)}]$ at P_i. During the customer's service time adjustment process the synchronisation conditions are checked nine times: at $s_i^{(l)}$, $f_i^{(l)}$, and on the occurrence of each arrival or departure event.

To demonstrate how this function works let us consider a simple example. Assume that synchronisation has been detected involving the processing of customers at P_i and the presence of customers at P_k, $k \neq i$, and that the synchronisation marking has been calculated. We retrieve the multiset[3] of the entry and exit timestamps of all customers processed by P_k, i.e. $e_k^{(j)}, f_k^{(j)}$,

[3]Multiple customers may have a common timestamp; thus a multiset allows for such cases.

for $j = 1, \ldots, n_k$, and then form the timestamp intersection multiset, defined as

$$T_\cap = \{e_k^{(j)}, f_k^{(j)} | s_i^{(l)} \leq e_k^{(j)} \leq f_i^{(l)}, s_i^{(l)} \leq f_k^{(j)} \leq f_i^{(l)}, j = 1, \ldots, n_k\}$$

where $s_i^{(l)}$ and $f_i^{(l)}$ are the service initiation and termination timestamps respectively of the lth customer who received service at P_i and define the service time interval to be adjusted. First $T_\cap$ is sorted (in ascending order). We begin the service time adjustment process by retrieving the marking of P_k at $s_i^{(l)}$ – using $M_k(t)$ – and check if it satisfies the corresponding synchronisation marking, i.e. $M_k(t) \geq$ `synchMarking[k]`. If it does, we create an open-ended time interval, starting at $s_i^{(l)}$, i.e. $[s_i^{(l)},\ .\ ]$ and proceed to the first timestamp in $T_\cap$; otherwise we directly proceed to the first timestamp. We then repeat the same process for every timestamp contained in $T_\cap$, while creating new and closing existing time intervals where appropriate. An existing time interval closes at a timestamp t if the synchronisation marking of P_k at t is not satisfied. A new time interval is created if the synchronisation marking of P_k at time t is satisfied and no open-ended time interval already exists. Finally, the last existing open-ended time interval is closed by the service termination timestamp $f_i^{(l)}$.

We obtain the actual[4] service time by adding the time difference between the end and the start of each of the previously formed subintervals. We note that if $T_\cap$ is empty, meaning that no departure nor arrival events occurred during the customer's $c_i^{(l)}$ service time interval, then the actual service time is simply the difference between $f_i^{(l)}$ and $s_i^{(l)}$.

5.4 Synchronisation Representation in our Models

After the synchronisation between service areas is detected, it needs to be incorporated into the GSPN performance model that is constructed during stage four of the data processing pipeline.

Considering the place representing P_i and its outgoing service transition t_i then for every place representing P_k such that `synchMarking[k]` > 0, we connect P_k to t_i via a double-headed arc

[4]The time during which all the synchronisation conditions where met, i.e. when service was indeed progressing.

with weight equal to `synchMarking[`k`]`. We use this representation since we are dealing with location tracking environments where customer entities are preserved.

In Section 4.4 we described how we fit a HErD to the extracted service time samples of each service area. We also discussed how we represent each HErD in terms of GSPN components (see Figure 4.13) and how we substitute each transition created in the first task of the third stage by a GSPN subnet which reflects the fitted HErD. Now, let us consider the synchronisation condition between server places p_1 and p_2 shown in Figure 5.3. In this example, the synchronisation condition specifies that at least two customers must be present in service area P_2 so that customers can be processed in P_1. As explained in the previous paragraph, this condition is represented by a double-headed arc – of weight two – connecting p_2 to transition t_3.

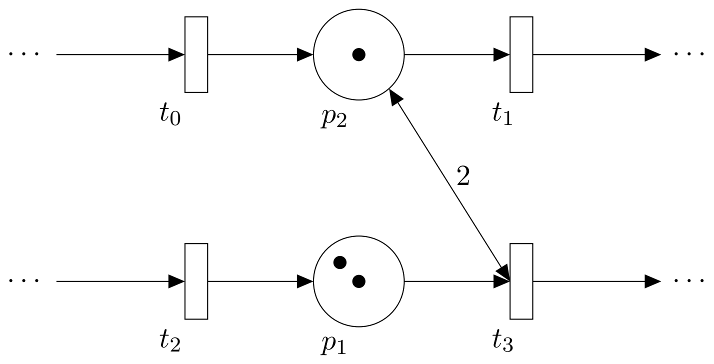


Figure 5.3: Modelling synchronisation between server places p_1 and p_2 with p_2 being the synchronising place with a synchronisation marking equal to two.

In order to preserve the synchronisation condition between p_1 and p_2 when t_3 is replaced by a GSPN subnet, we need to connect p_2 to every immediate transition on the left-hand side of the GSPN subnet with double-headed arcs – one for each immediate transition – of weight two. These are the transitions whose firing allows tokens to enter the subnet and their firing rates represent the weights of the Erlang branches of the HErD. Figure 5.4 shows the resulting GSPN model under the assumption that t_3 is replaced by a subnet which reflects a four-state HErD with two Erlang branches, where each branch has two states.

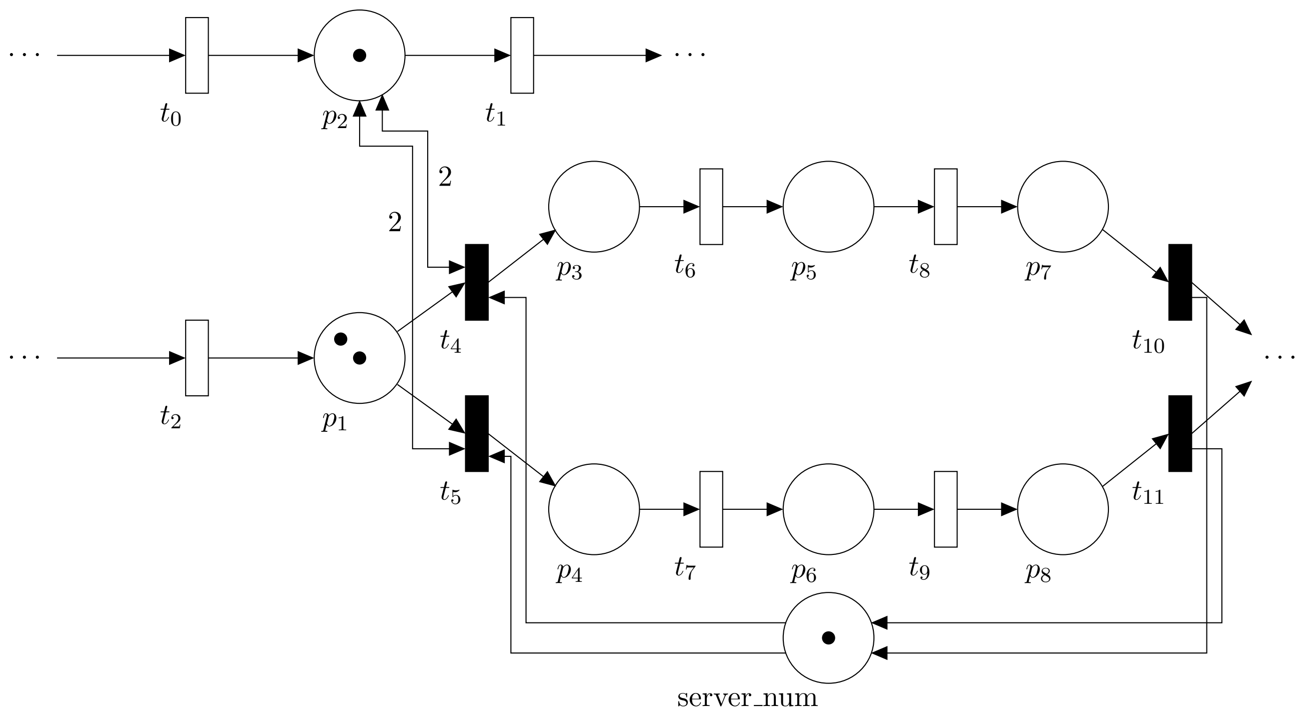


Figure 5.4: Modelling synchronisation between server places p_1 and p_2, when t_3 (cf. Figure 5.3) is replaced by a GSPN subnet. Service initiates at p_1 only if p_2 is marked with at least two tokens and p_1 with one.

5.5 Evaluation

In this section we conduct a case study to test and demonstrate the developed synchronisation detection mechanism. For this case study we have generated location tracking data using an extended version of LocTrackJINQS [56], which supports synchronisation between service areas of the system (see Section 3.2.1). Results for the service area location and service radii inference, the synchronisation detection mechanism and the service time distribution fitting (adjusted for synchronisation when appropriate) are presented. We conclude this chapter with a discussion on the obtained results and future development.

5.5.1 Case Study

Figure 5.5 shows the experimental setup for the case study and the flow of customers in the system (indicated by arrows). The simulation takes place in a virtual 25 m × 25 m environment and the customers are assumed to travel within the system at a speed drawn from a normal distribution with mean 0.5 m/s and standard deviation 0.15 m/s. The location update error, which emulates the standard error of a real life location tracking system, is also normally distributed with mean 0.15 m and standard deviation 0.2 m.

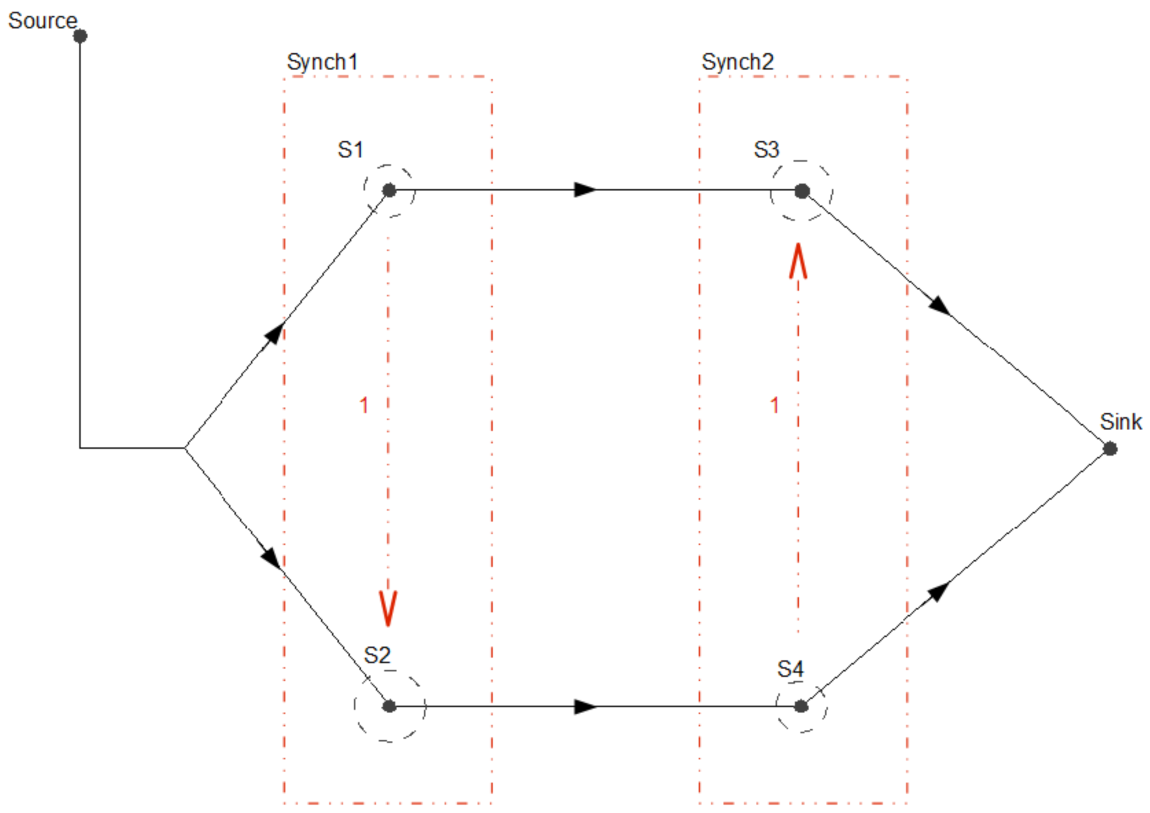


Figure 5.5: The experimental setup in terms of abstract system structure. The arrows represent the customer flow in the system and the branching to S1 and S2 occurs with equal probabilities. The service areas contained in the dotted red rectangles Synch1 and Synch2 are subject to synchronisation. The synchronisation condition is represented by the dotted red arrow. Its source indicates the synchronising service are and its target the service area to be synchronised. The number of customers required to be present in the synchronising service area so that service can be supported in the synchronised service area is denoted by the weight of the dotted red arrow.

Each service area consists of a single customer-processing server and employs a random customer service discipline. Service areas S2 and S3 require at least one customer to be present in service areas S1 and S4 respectively in order to service their customers. The service time for each service area follows a different density function. The actual location and service radius of each service area as well as its service time density can be seen in Table 5.1.

	Server Location	Service Radius	Service Time Density
S1	(8.0,5.0)	0.5	Erlang(2, 0.1)
S2	(8.0,15.0)	0.7	Exp(0.1)
S3	(16.0,5.0)	0.6	Erlang(4, 0.3)
S4	(16.0,15.0)	0.5	HErD(2, 3; 0.5, 0.5; 0.08, 0.12)

Table 5.1: The parameters for each service area in the system, for this case study. Server locations and their corresponding service radii are given in metres. The parameters of the HErD represent the phase lengths, weights and rates for each branch respectively, separated by a semi-colon. The unit of the rate parameters for phase-type distributions is customers per second.

5.5.2 Results

The inferred locations and service radii of the service areas, as well as the error between these and their actual values, are depicted in Table 5.2. From these results we can see that the location and radii of the service areas are approximated very well. The maximum error for the location inference is 0.145 metres and for the service radius approximation is 0.096 metres.

	Server Location			Service Radius		
	Real	Inferred (3 d.p.)	Error (3 d.p.)	Real	Inferred (3 d.p.)	Absolute Error
S1	(8.0,5.0)	(7.900,5.009)	0.100	0.5	0.526	0.026
S2	(8.0,15.0)	(7.980,14.936)	0.067	0.7	0.773	0.073
S3	(16.0,5.0)	(16.143,4.974)	0.145	0.6	0.696	0.096
S4	(16.0,15.0)	(16.017,14.988)	0.021	0.5	0.575	0.075

Table 5.2: The inferred location and service radius for each service area in the system accompanied with the absolute error, for this case study. These values are given in metres.

For the evaluation of the sample extraction and HErD fitting process we conduct Kolmogorov-Smirnov tests, to examine the compatibility of the extracted service time samples for each service area with its best-fit HErD (see Table 5.4). The parameters of HErDs fitted for each service area's service time density, along with the computed relative entropy between the simulated and fitted distributions, are depicted in Table 5.3. The selected HErD for each set of extracted service time samples is plotted against the corresponding theoretical service time density which was used in our simulation (see Figure 5.6).

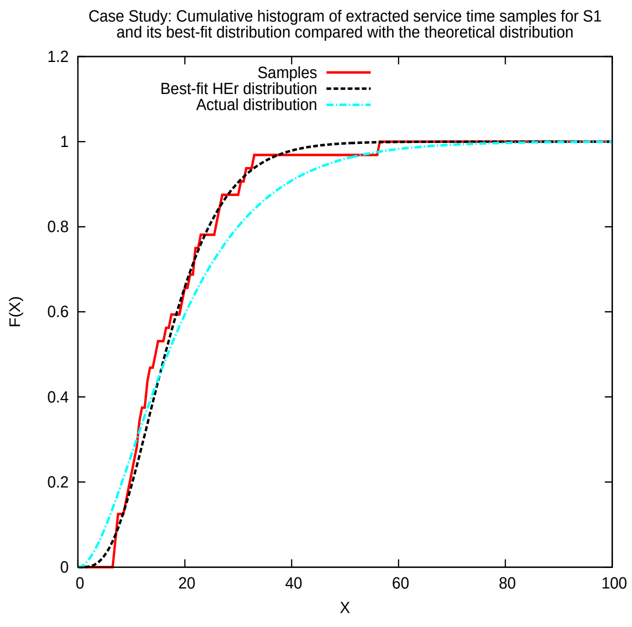
	Service Time Density	Fitted HErD Parameters			Relative Entropy (3 d.p.)
		Phase Lengths	Rate (3 d.p.)	Weights (3 d.p.)	
S1	Erlang(2, 0.1)	4	0.225	1.0	0.203
S2	Exp(0.1)	1	0.088	1.0	0.008
S3	Erlang(4, 0.3)	3	0.186	1.0	0.071
S4	HErD(2,3;0.5,0.5;0.08,0.12)	3	0.137	1.0	0.042

Table 5.3: The parameters of the HErD fitted for each service area's service time density with the relative entropy (in nat) between the theoretical and fitted probability density function. The parameters of the HErD represent the phase lengths, weights and rate for each branch respectively, separated by a semi-colon.

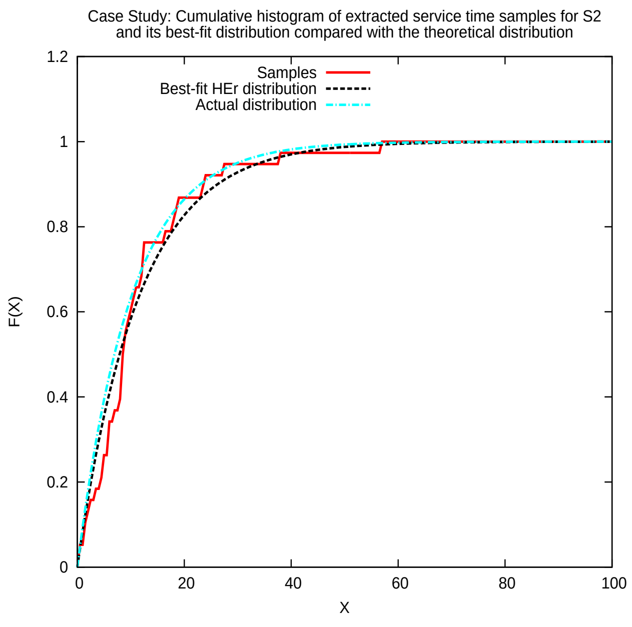
S1	Test Statistic	0.1196	
	α	0.1	0.05
	Critical Values	0.1844	0.2108
	Compatible ?	Yes	Yes
S2	Test Statistic	0.1250	
	α	0.1	0.05
	Critical Values	0.1697	0.1939
	Compatible ?	Yes	Yes
S3	Test Statistic	0.1137	
	α	0.1	0.05
	Critical Values	0.1903	0.2176
	Compatible ?	Yes	Yes
S4	Test Statistic	0.0959	
	α	0.1	0.05
	Critical Values	0.1719	0.1965
	Compatible ?	Yes	Yes

Table 5.4: Kolmogorov-Smirnov test at significance levels 0.1 and 0.05 applied to the extracted service time samples (adjusted for synchronisation) for each service area in the case study. The null hypothesis is that the extracted samples belong to the corresponding best-fit HErD.

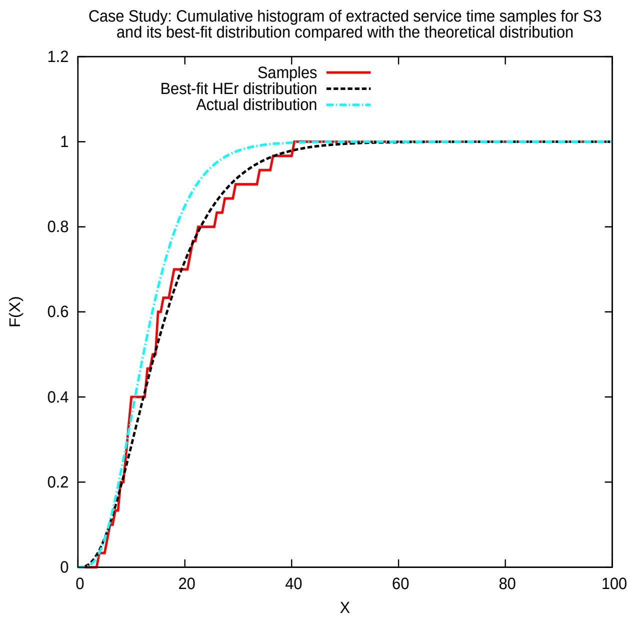
Here, as in Section 4.5.2, the parameters of the fitted HErDs do not match the parameters of the actual distributions in every case; however, we can see from the graphs that the fit is, in general, very good. We also note the successful application of the service time adjustment process for service areas S2 and S3; if the service time samples were not successfully adjusted according to the synchronisation conditions, a significant difference would have been observed between the theoretical and best-fit hyper-Erlang distributions. Additional evidence which



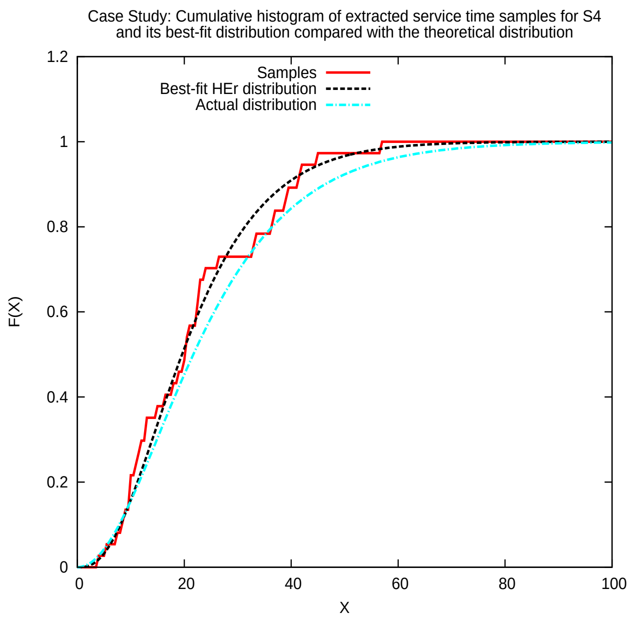
(a)



(b)



(c)



(d)

Figure 5.6: Case Study: Graphs 5.6(a), 5.6(b), 5.6(c) and 5.6(d) show the cumulative histogram of the extracted service time samples (adjusted for synchronisation for S2 and S3) and its best-fit hyper-Erlang distribution compared with the theoretical distribution for S1, S2, S3 and S4 respectively.

supports the compatibility between the simulated and fitted distributions is provided by the relative entropy which has low values (≤ 0.203).

Figure 5.7 shows the constructed GSPN performance model in compact transition form. We observe that the structure of the inferred model matches the structure of the abstract simulated system. The transitions between pairs of server places (places that correspond to the service areas) represent the travelling time for each particular pair. The weights of the immediate transitions T1 and T0 are 0.457 and 0.543 respectively, approximately matching the simulated initial routing probabilities of the customer flow which are 0.5 and 0.5. In the model we can also see the constructed synchronisation between S2 and S1 (synchronising service area) as well as between S3 and S4 (synchronising service area).

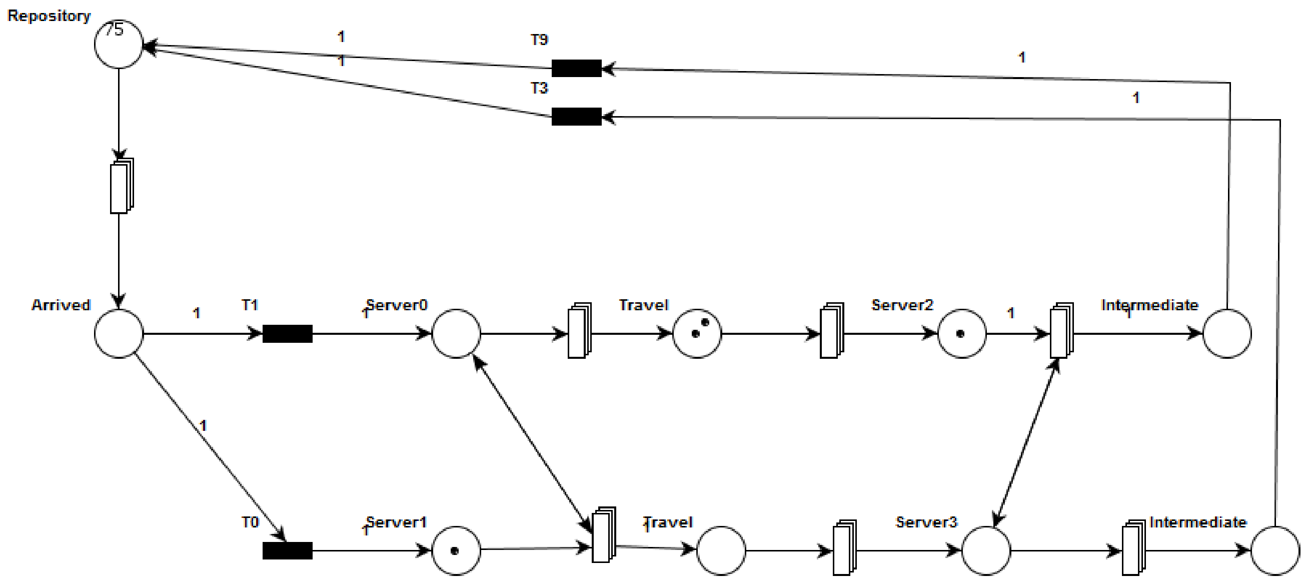


Figure 5.7: The inferred GSPN performance model for this case study, visualised in PIPE2 (in compact transition form).

5.5.3 Conclusion

This chapter has presented an automated mechanism for presence-based synchronisation detection between service areas in a customer-processing system. This mechanism has been implemented and incorporated into our existing data processing pipeline to extend its range of applicability; it allows the automated inference of GSPN performance models of more complex customer-processing systems. We conjecture that our methodology can be applied to a variety of systems whose underlying GSPN structure includes simple nets (SPL-nets). An example of a real life situation where this methodology could be successfully applied is a Magnetic Resonance Imaging (MRI) unit in a hospital. The MRI control room is physically separate from the MRI chamber and the scanning process is initiated if and only if two conditions are satisfied:

the patient – the person who receives the scan – must be located in the MRI chamber and the radiologist who operates the MRI scanner must be located in the MRI control room.

The case study results indicate that the developed methodology can infer the presence and conditions of synchronisation in simple systems accurately and provide stronger evidence to the correctness of the existing processing pipeline (stages two and four).

Chapter 6

Pipeline Extensions

Under the assumptions and limitations described in Chapter 4, the developed methodology was able to successfully infer a GSPN performance model capturing both the physical structure and the stochastic behaviour of the underlying system from location tracking data. An additional mechanism was subsequently incorporated into the initial methodology to enable the detection and representation of presence-based synchronisation between the system's service areas (see Chapter 5). This was all possible without needing to extend the modelling power of GSPNs.

In Section 4.5.3 we discussed some limitations of the data processing pipeline, including its inability to represent multiple customer classes. This is a serious deficiency as it prohibits the application of the pipeline to real life systems dealing with multiple customer classes. Examples of such systems include passport control checkpoints at airports, Accident and Emergency (A&E) departments of hospitals and, in general, systems where not all customers are treated in a homogeneous fashion.

In this chapter, we augment the PNPMs constructed by our methodology to support coloured tokens, using CGSPNs [74] instead of GSPNs. We describe how the use of CGSPNs allows us to facilitate multiple customer classes and, furthermore, to control the routing of customers as they pass through various processing stages in a dynamic way. Here, we also present another extension of the data processing pipeline which enables the representation of inter-routing probabilities of the customer flow in scenarios where the customers follow different routes in the system after receiving service at a service area. These extensions have been incorporated within the third stage of the processing pipeline. We conclude this chapter by presenting three case studies to evaluate these new features.

6.1 Multiple Customer Classes

The main assumption here is that the class of each customer in the system is known and provided through the customer's associated location updates, e.g. the `type` field of a typical location update contains the category the monitoring tag belongs to, or by a static mapping of each customer's unique identifier (`tagName`) to the class it belongs to. The tag `type` is a generic field and it can be modified by the user, through the RTLS's software, to define custom categories. For example, one can set the tag category to be one of the types "Person", "Patient", "Doctor", "Package", and so on, depending on the application. For simplicity and to have the ability to support general systems we use distinct non-negative integers to distinguish between such categories (customer classes).

This extension presents no effect to the first, second and fourth stage of the data processing pipeline. Also no modifications or additions are required in the service time and travelling time sample extraction process, taking place in the third stage of the pipeline. However, in order to allow the differentiation of the extracted time samples into groups – one for each customer class – we store the class of the customer each sample corresponds to. During the latter process we also infer the total number of customer classes in the system by simply counting all the different classes observed. Once the service time and travelling time extraction process has been completed, the samples associated with the service of customers at each service area, or movement of customers between pairs of service areas, are grouped according to each observed customer class. For example, if n classes of customers are serviced at a service area, then we will have n groups of service time samples (and subsequently, in the fourth stage of the pipeline n HErD fits) associated with that service area.

The main difference in the third stage of the data processing pipeline is the way we form the

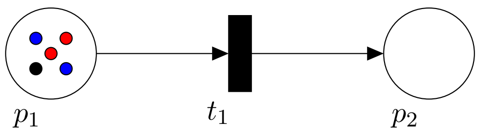


Figure 6.1: A simple CGSPN with three different token classes.

initial structure of PNPM and, in particular, the transitions contained in the model. Normally, in a CGSPN, or even a CPN, each transition can support many different firing modes. For example, let us consider the CGSPN depicted in Figure 6.1 where p_1 is marked with three different types of tokens: two blue, two red and one black. The transition t_1 may require two blue tokens to be enabled, or one red, or one black or even a combination of different token types, i.e. one black and two red. Similarly, the firing of t_1 may place any number and combination of token types in p_2. It is left to the net designer to specify the conditions under which t_1 fires so that it accurately reflects the behaviour of the system being modelled. We also note that a different firing rate or weight can be associated with each supported firing mode of a timed or immediate transition respectively.

To facilitate the unambiguous visualisation of a transition's firing modes, we use a different transition for each firing mode that we wish to support. That is, if we consider the latter example, assuming that t_1 has three firing modes (see Table 6.1), we use three transitions; this is shown in Figure 6.2. This decision was heavily influenced by the design of the extension

Mode	$I^-(p_1,t_1)$(red)	$I^-(p_1,t_1)$(blue)	$I^-(p_1,t_1)$(black)	$I^+(p_2,t_1)$(red)	$I^+(p_2,t_1)$(blue)	$I^+(p_2,t_1)$(black)
1	1	0	0	1	0	0
2	0	1	0	0	1	0
3	1	1	1	1	1	1

Table 6.1: The three firing modes for transition t_1 shown in Figure 6.1. $I^-(p_i,t_j)(c_k)$ denotes the number of tokens of colour c_k that must be present in p_i in order for t_j to be enabled. $I^+(p_i,t_j)(c_k)$ denotes the number of tokens of colour c_k that are created in p_i when t_j fires.

that was made to PIPE2 to support coloured tokens [31].

In Section 4.3 we demonstrated how we create the initial structure of the PNPM under the assumption of one customer class. When dealing with multiple customer classes, the process of place construction is the same as before. Service area service time transitions are created in a similar manner but, instead of having a single transition between pairs of service areas

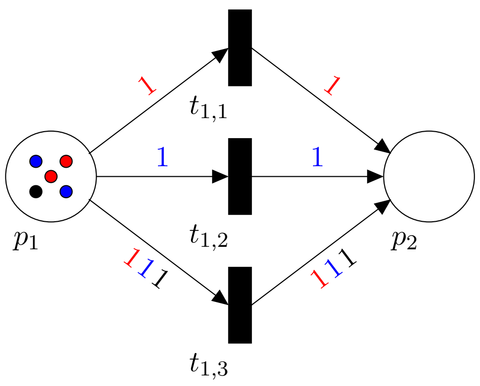


Figure 6.2: The CGSPN shown in Figure 6.1 with one transition for each different firing mode of t_1.

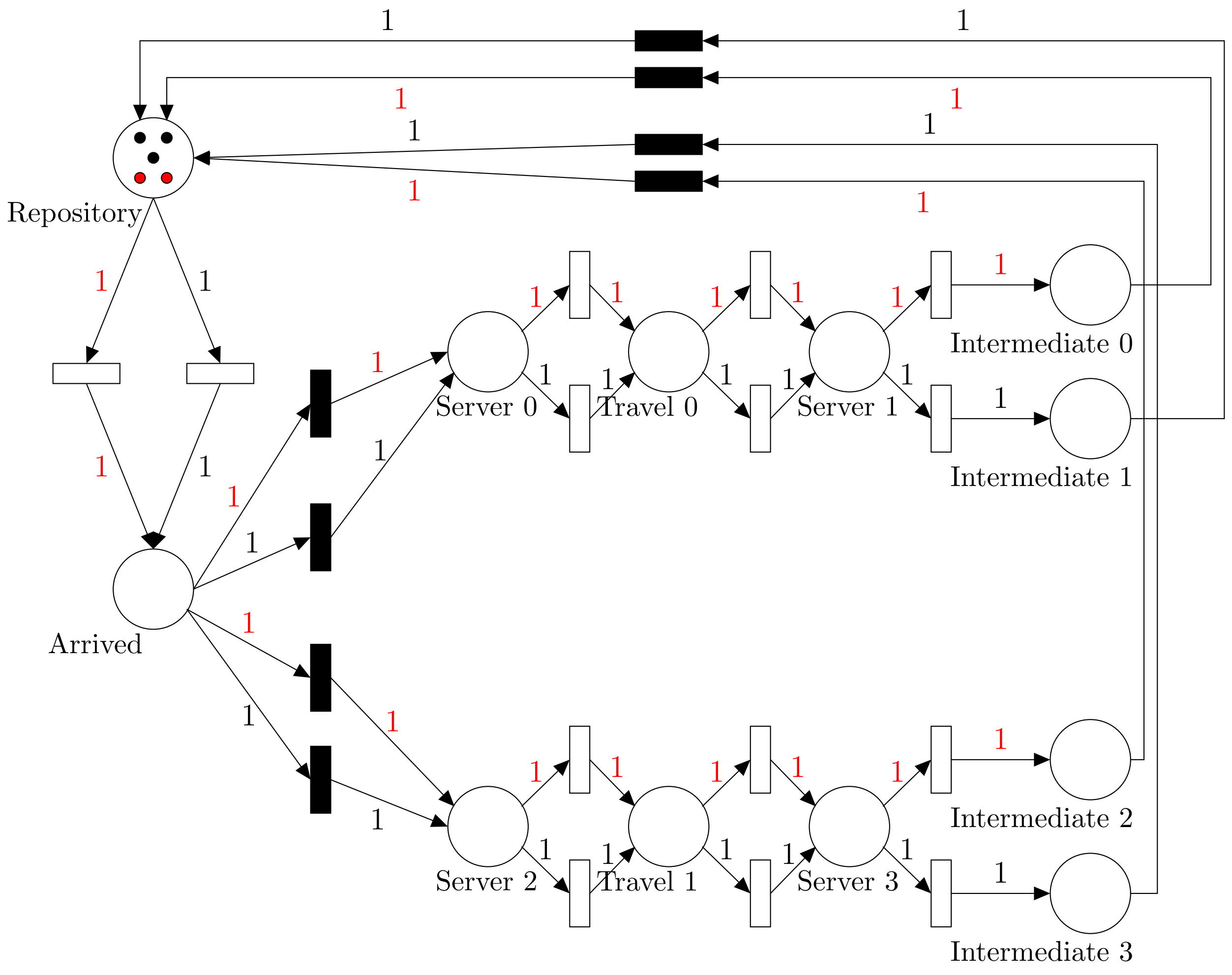


Figure 6.3: A complete overview of the initial structure of a PNPM model obtained at the end of the third stage of the processing pipeline of a system with two customer classes.

where customer movement was observed, we have multiple transitions: one for each class of customers that was processed by the corresponding service area. The same principle applies for the construction of travelling time transitions. Additional intermediate places are used to transfer the tokens back to the repository place through immediate transitions where appropriate. Figure 6.3 shows the initial structure of the PNPM depicted in Figure 4.12, assuming that now we have two customer classes.

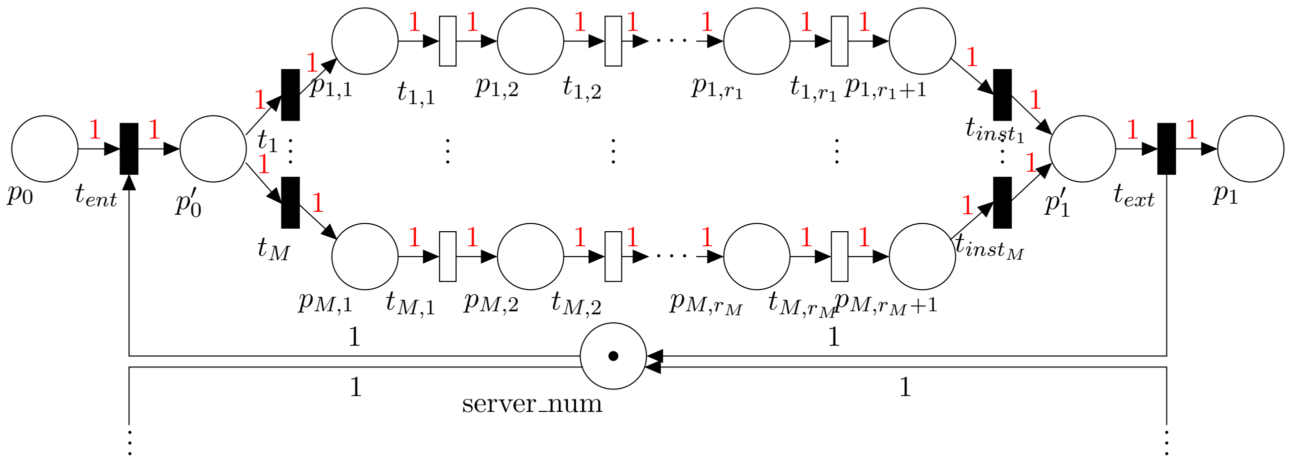


Figure 6.4: CGSPN subnet representation for service time transitions. The colour of tokens supported varies according to the customer class which the net applies for. The complementary place labelled as server_num is shared among subnets associated with the same service area. This allows only one token to be in a subnet at a time and prohibits parallel service of distinct customer classes (we assume single-server semantics). p_0 represents a service area and p_1 is associated with the movement of customers between p_0 and some other service area.

Like before, all timed transitions at this stage are not parameterised. In the fourth stage a HErD is fitted to each set of extracted samples – one for each customer class – and now, a CGSPN subnet is used to represent it. The CGSPN subnet's structure is similar to the one used in the single customer class scenario (see Figures 4.13 and 4.14 for travelling time and service time transitions respectively) and the type (colour) of tokens allowed in each subnet is defined by the arc weights accordingly. However, we make two adjustments to the structure of the CGSPN subnets used to replace service time transitions so that the service of customers remains accurately modelled: firstly, the complementary place server_num[1] (earlier used to ensure that no more than one token is allowed in a subnet) is now shared among all subnets associated with a particular service area. This prohibits the parallel service of customers of different classes. We note that the token type contained in the place server_num is irrelevant to the customer class that each subnet corresponds to; therefore, no colour is required. Secondly, we introduce two additional places and two immediate transitions to each subnet (cf. Figure 6.4). These two transitions t_{ent} and t_{ext}, in conjunction with the place server_num, enforce mutual

[1]Note that this place is found only in subnets that are used to replace service area service time transitions.

exclusion to the service area's resources. Also, the marking dependent weight of t_{ent} (defined as the number of tokens of some colour c that are contained in place p_0 – cf. Figure 6.4 – given that colour c is handled by the subnet) ensures that the random service policy is retained when multiple customer classes exist. The transition t_{ext} is simply used to allow tokens to exit the subnet and to restore the shared resource to server_num; thus, its weight is set equal to one.

6.2 Customer Routing

In this section we present another limitation of the existing processing pipeline that can be overcome with the use of coloured tokens. Often, in customer-processing systems, routing is not probabilistic but deterministic with several phases of processing. Also, the journey through these processing phases may involve repeated visits to the same service centres. An example is shown in Figure 6.5 whereby customers enter the system and immediately proceed to Service Area A. Then customers are processed by Service Area B and Service Area C in respective order. Finally, they revisit Service Area A and as soon as they complete service there, they exit the system. Examples of real life customer-processing systems which exhibit such behaviour include hospitals and assembly lines. We also note that customers may request a different type of service when revisiting a service area. For this reason we keep separate sets of service time samples for repeated visits of customers to the same service area. Of course, there are also systems which exhibit more complex behaviour; customers may perform the same cyclic service sequence multiple times. However, in this research we restrict our approach to a more common and useful use-case where the cyclic service sequence is performed only once.

If we model the system described above (cf. Figure 6.5) using GSPNs we would obtain the model depicted in Figure 6.6. Although the structure of the model appears to be correct, it fails to express the desired behaviour and routing of the customers in the system. This failure arises due to the race condition between the outgoing transitions of the place Server 0, t_1 and t_4. When a token is created in Server 0 after t_9 fires, t_1 and t_4 are simultaneously enabled. In particular, the selection of the transition to be fired is purely probabilistic and the probability

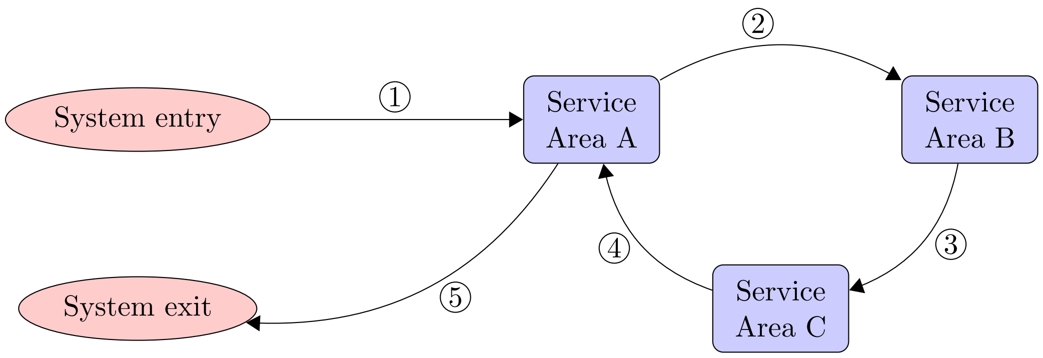


Figure 6.5: A system which provides services to customers in a cyclic fashion. Arrows indicate the flow of customers in the system.

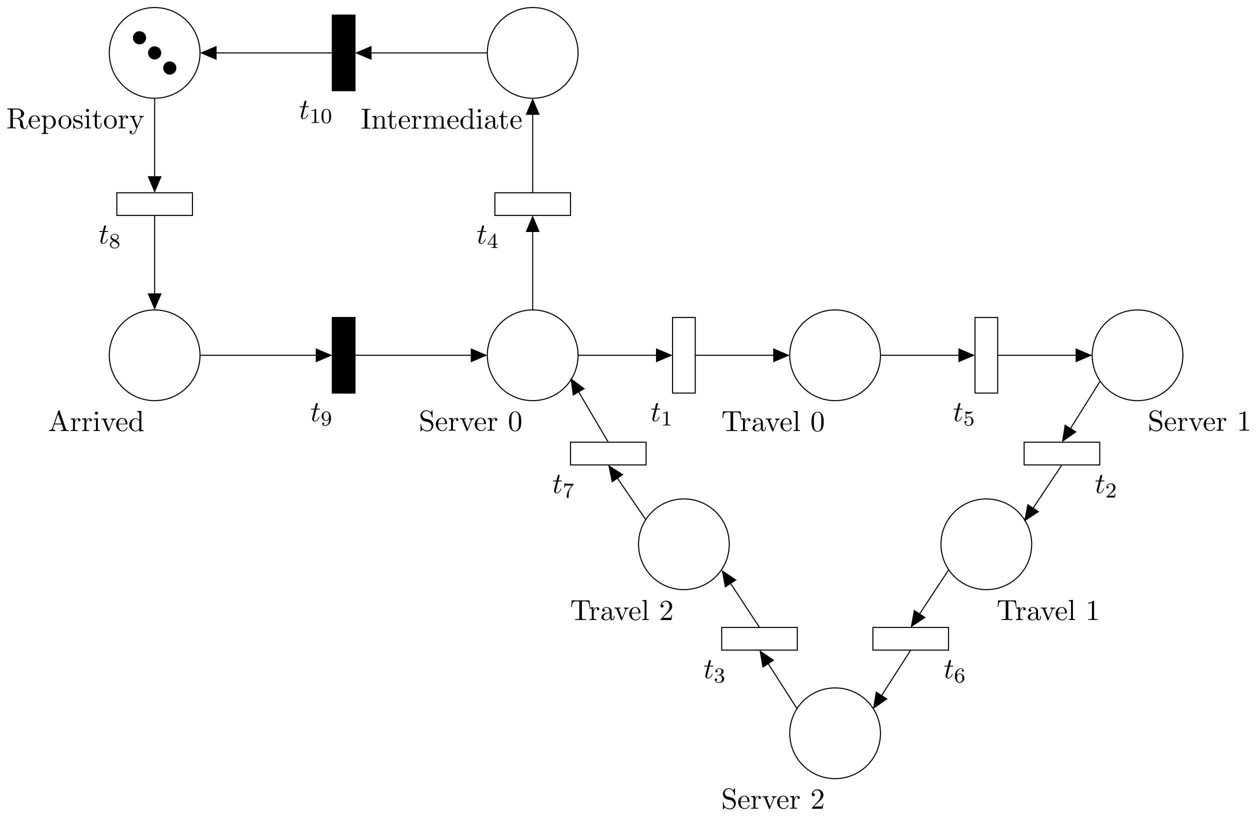


Figure 6.6: The GSPN model (non-parameterised version) for the system depicted in Figure 6.5, using our existing methodology.

depends on the firing rates of the two transitions. Under this scenario, two mutually exclusive events can occur: t_1 fires and a token is transferred to the place Travel 0 connecting Server 0 and Server 1, or t_4 fires and a token is eventually transferred to the Repository place, meaning that the customer has exited the system. The latter scenario must be avoided as it does not model the true behaviour of the customers in the system, i.e. a customer (token) arriving at Server 0 for the first time must obtain the services of Server 1 and Server 2 before routed to the Repository.

The main idea is to identify when and where this type of routing occurs and to use different types of tokens to express the "phases of service" of the customers. When cyclic services are identified in a system, we need to differentiate between the customers who have obtained service from each service area contained within the cycle and the ones about to enter it. We classify the former and latter groups of customers as serviced and unserviced respectively. Since we assume that neither knowledge of the system's structure nor behaviour is readily available – apart from the customer traces provided by the RTLS – it is clear that firstly we need to detect whether cyclic services are incorporated in the system being modelled.

6.2.1 Cycle Detection

Here, we exploit the elegant graphical representation of Petri Nets in order to convert the non-parameterised version of the PNPM into a directed graph. On this graph we apply Johnson's algorithm [64] to examine the presence of cycles in the model's structure. By definition, Petri Nets are directed bipartile graphs, i.e. places can only be connected to transitions and vice-versa [14]. We map each place in the model to a vertex and each transition – along with its incident arcs – to a directed edge (see Figure 6.7). This mapping is possible since in our models the set of pre-places and post-places of each transition, i.e. $\bullet t$ and $t\bullet$ respectively, has exactly one element. We note that the latter is not true for the service time transitions of service areas that are subject to some synchronisation condition(s) (cf. Figure 5.3) and for the immediate transitions t_{ent}, t_{ext}, contained in the (C)GSPN subnet which is used to reflect the time delay of the fitted HErD (only for the service time transition representation – cf. Figure 6.4). However, these two cases do not pose a problem here since they occur after the model is examined for the presence of cycles. In particular, the presence-based synchronisation detection mechanism is applied immediately after cycle detection is performed and the replacement of timed transitions by (C)GSPN subnets takes place during the fourth stage of the processing pipeline.

To convert a PNPM into a directed graph we first create an $N \times N$ integer matrix, the adjacency matrix, where N is the number of places in the model and every entry is initialised to zero. We

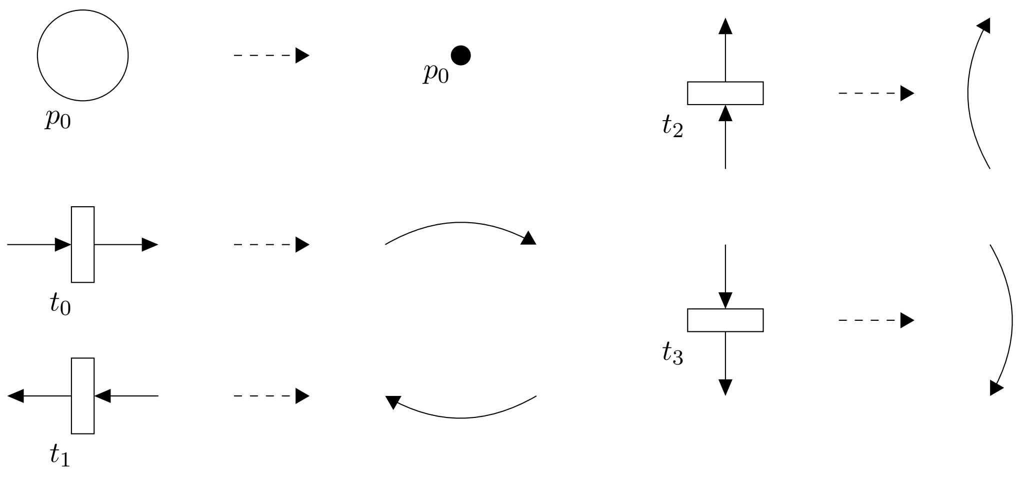


Figure 6.7: Converting a PNPM (non-parameterised version) to a directed graph.

then enumerate all constructed transitions and for each transition t retrieve the identities of its input and output places. The identities of the places in $\bullet t$ and $t\bullet$ are used to identify the row and column of the element to be marked in the adjacency matrix respectively. For example, if a transition is used to connect the places p_1 and p_5, the entry located in the first row and fifth column of the matrix is set equal to one. The graph can be visualised by drawing all the places in the model as vertices and directed edges connecting the places whose corresponding entry in the adjacency matrix is not zero. Figure 6.8 shows the directed graph for the GSPN model depicted in Figure 6.6.

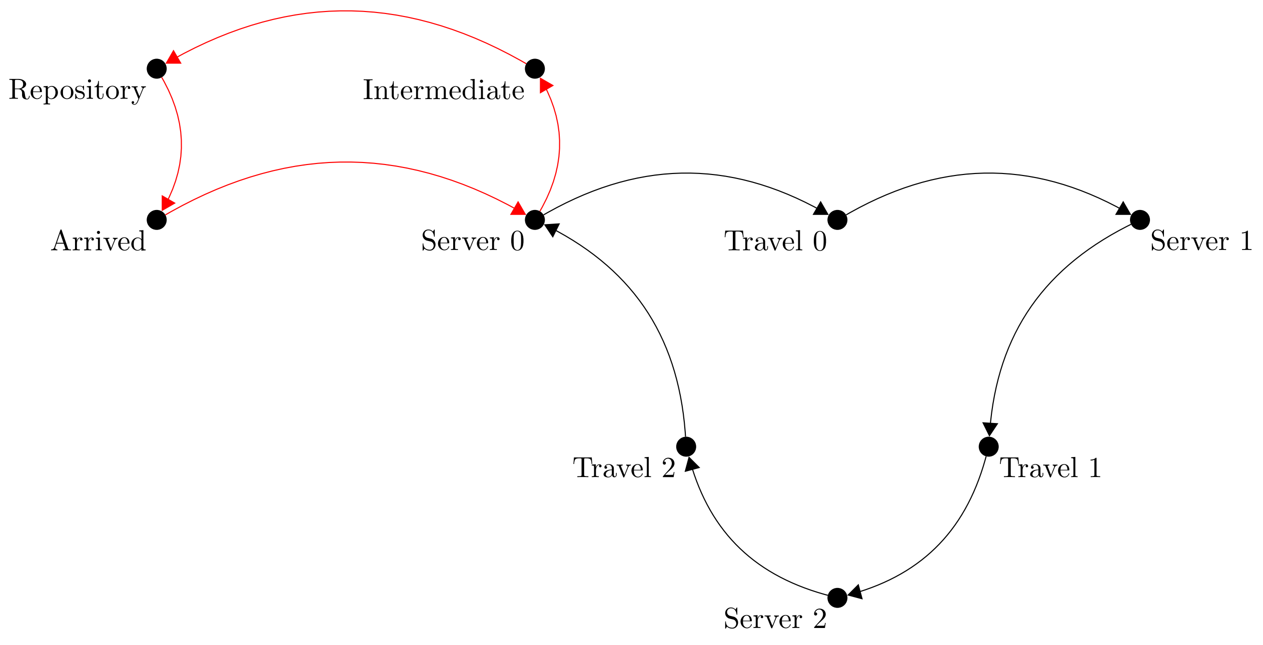


Figure 6.8: The directed graph produced for the Generalised Stochastic PNPM depicted in Figure 6.6. The paths consisting of black and red edges are two elementary cycles beginning at vertex Server 0.

Before we apply Johnson's algorithm on the graph we take advantage of some structural properties of the underlying PNPM. First, let us formally define the concept of a service cycle in a customer-processing system.

Definition 6.1. *Consider a physical customer-processing system consisting of the set of service areas $S = \{S_i \,|\, i = 1, \ldots, N\}$, for some finite $N \in \mathbb{N}$. A finite sequence of distinct service areas from which the customers of the system obtained some service, is said to be a service cycle given that the following conditions hold:*

1. *If the customers initiate their service sequence at S_i, $S_i \in S$, they must also terminate it at S_i after service completion at S_k, $S_k \in S$ and $i \neq k$.*

2. *When the customers complete their service at S_i for the second time, i.e. after obtaining service at S_k, they do not repeat the same service sequence.*

The service area which marks the initiation and termination of the service cycle is called the head of the service cycle.

If we observe Figure 6.8 we notice that only the vertex which corresponds to Server 0 has four adjacent vertices: two successors and two predecessors. This gives rise to the following theorem:

Theorem 6.1. *Given a customer-processing system and its corresponding directed graph, a service cycle exists only if the vertex v_j which corresponds to head of the service cycle S_j has at least two predecessors and at least two successors.*

Proof: We consider a physical customer-processing system consisting of the set of service areas $S = \{S_i \mid i = 1, \ldots, N\}$, for some finite $N \in \mathbb{N}$. Assume that a service cycle exists in the given customer-processing system, and consider the directed graph which corresponds to the PNPM (non-parameterised version) of the underlying system. We define:

1. $S_c \subseteq S$ to be the subset of service areas that consist the service cycle,

2. $P_c = \{v_j, v_{j+1}, \ldots, v_{j+k}\}$ to be the set of vertices which correspond either to the places representing service areas $S_i \in S_c$ or to the travel places that exist between pairs of service areas S_i, S_k, $i \neq k$, $S_i, S_k \in S_c$.

We examine the two conditions of the theorem individually:

At least two predecessors are required.
If no predecessor of the vertex v_j which corresponds to S_j, exists, then it is trivial that no service cycle exists. Now let us assume that only one predecessor of the vertex v_j exists, say v_p with $p \neq j$. We need to consider two cases. If $v_p \in P_c$, it corresponds to the travel place which connects the last service area of the service cycle, say S_l, $l \neq j$, $S_l \in S_c$, to the head of the service cycle S_j. This means that the customers initiated their service sequence at the service area $S_l \in S_c$. But since S_j is the head of the service cycle and $j \neq l$, we reach a contradiction. If $v_p \notin P_c$, then v_j is not accessible from any vertex $v \in P_c$, and consequently customers cannot reach S_j from the last service area of the service cycle, meaning that no service cycle exists. Thus, we obtain a contradiction.

At least two successors are required.
If no successor of the vertex v_j which corresponds to S_j exists, then it is trivial that no service cycle exists. Now assume that only one successor of the vertex v_j exists, say v_p, $p \neq j$. If $v_p \in P_c$, it corresponds to the travel place which connects S_j to the next service area in the service cycle, say S_{j+1}. Since no other vertex is a successor of v_j this means that all customers completing their service cycle will be re-routed to S_{j+1}, thus repeating the same service cycle. This violates the second condition of definition 6.1 and therefore, a contradiction is obtained. If $v_p \notin P_c$, then none of the vertices corresponding to service areas S_l, $l = j+1, \ldots, j+k$, are accessible from S_j, hence no service cycle exists. Again, we reach a contradiction.

□

Johnson's algorithm, as explained in Section 2.4.2, is used to enumerate all elementary cycles of a directed graph. The direct application of this algorithm the directed graph G, obtained from the non-parameterised version of a PNPM, would always return at least one elementary cycle which will include the Repository place, e.g. see Figure 6.8. As we mentioned earlier, this place of the PNPM represents the entry/exit point of the underlying system and thus, is actually not part of any service cycle. To avoid such unwanted cycles will be detected by the algorithm we examine the subgraph of G induced by the set of vertices $V - \{\text{Repository}\}$ instead of G. Also, instead of examining for cycles starting at each vertex of the strong component of the graph, we use Theorem 6.1 and restrict the algorithm's search to begin only with vertices v_i which have at least two predecessors and at least two successors. This eliminates the "requirement" to calculate the strongly connected component of the graph (see Section 2.4.2), thus making the cycle detection process even faster.

Consider the previous example of the customer-processing system shown in Figure 6.5 and its corresponding PNPM (Figure 6.6). When we apply Johnson's algorithm on the graph – formed as described above – we obtain the ordered list of vertices which constitute the elementary cycle and subsequently the ordered list of places that correspond to the representation of the service cycle in our model, i.e. Server 0, Travel 0, Server 1, Travel 1, Server 2, Travel 2, Server 0.

In order to eliminate the race condition described earlier we must identify the participating transitions and change their firing mode. That is, referring to the previous example, if transitions t_1 and t_4 have different firing modes with respect to the type of tokens they support, the race condition between them ceases to exist. In this way, we are able to distinguish between customers who are about to enter the service cycle and customers who have just completed it. The desired coloured version of the PNPM which implements the latter idea is depicted in Figure 6.9. We notice that the race condition between t_1 and t_4 is resolved since the two transitions now require different types of tokens to be enabled; t_1 requires one black (default) token and t_4 one red. The behaviour of this model is as follows (for simplicity assume that the Repository place is marked with only one black token): one black token is destroyed from the Repository place and another is created in the Arrived place. Then t_9 is enabled and fires instantly, transferring the token to Server 0. Now, since t_4 is no more enabled, the only enabled transition is t_1 which, when fired, transfers the black token to Travel 0. A sequence of transition firings will eventually transfer the token to the travel place Travel 2. When t_7 fires, the black token in Travel 2 will be destroyed and a red token will be placed on Server 0. Since

t_1 requires a black token to be enabled, the token cannot re-enter the service cycle and it will be routed though t_4 towards the Repository place once again. If we consider the token to be a customer, and its colour to represent its service status, we see that the behaviour of the real system is modelled accurately using this CGSPN model since the earlier probabilistic routing of a customer at Server 0 (see Figure 6.6) is now deterministic. Note that the transition t_{10} is used to "recycle" tokens, i.e. restore them to their original state and place them back to the Repository place. The question that follows is how we achieve this, given the original GSPN model and the sequence of places which constitute the service cycle.

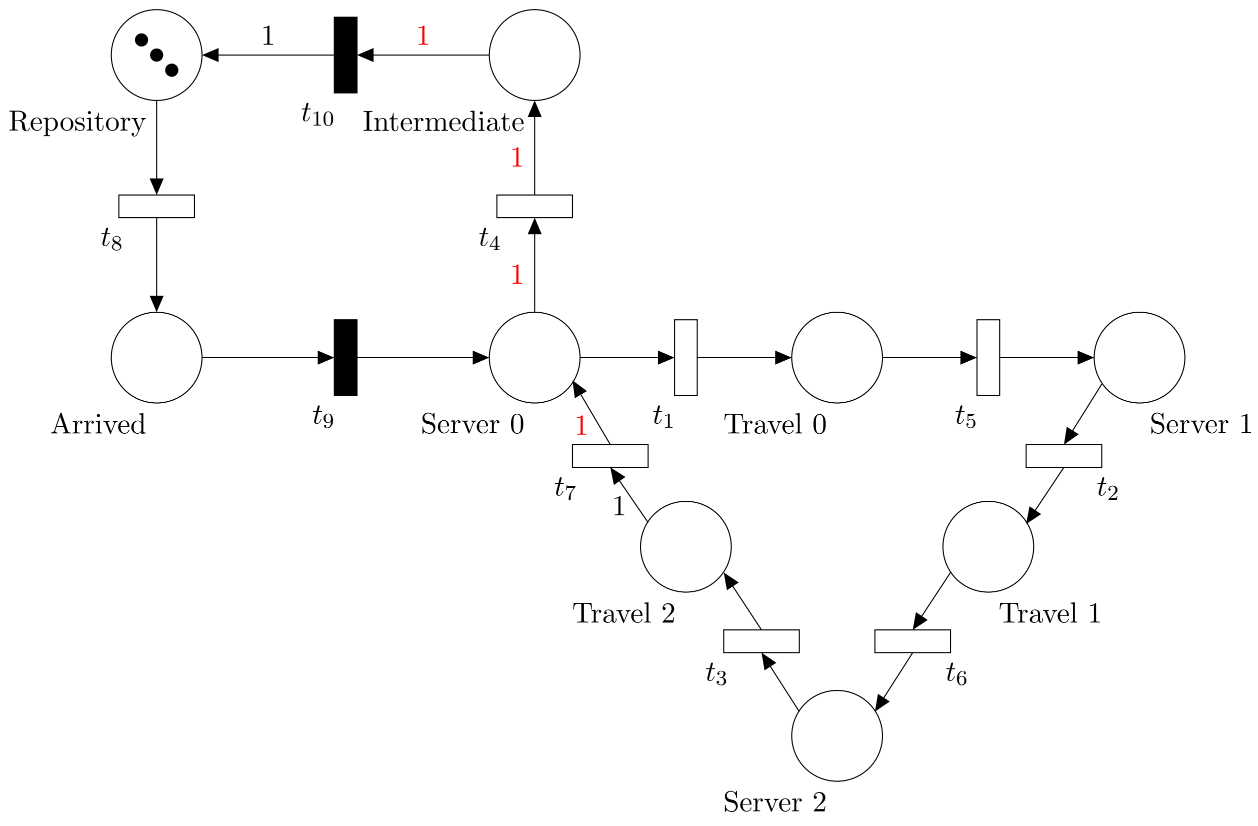


Figure 6.9: The CGSPN model (non-parameterised version) for the system depicted in Figure 6.5, which enables the distinction between serviced (red tokens) and unserviced (black tokens) customers. Where no arc inscription is explicitly shown one black (default) token is assumed.

We begin by retrieving the sets of input and output transitions of the place $p_h^{(i)}$ which corresponds to the head of the ith service cycle. For each transition t_{in} contained in $\bullet p_h^{(i)}$ we obtain its input place which is the only element of $\bullet t_{in}$ and similarly, for each transition t_{out} contained in $p_h^{(i)}\bullet$ we obtain its output place which is the only element contained in $t_{out}\bullet$. Again, considering the previous example, we have:

- $p_h^{(0)} =$ Server 0,
- $\bullet$Server 0 $= \{t_7, t_9\}$,
- Server 0$\bullet$ $= \{t_1, t_4\}$,
- $\bullet t_7 = \{\text{Travel 2}\}$, $\bullet t_9 = \{\text{Arrived}\}$,
- $t_1\bullet = \{\text{Travel 0}\}$, $t_4\bullet = \{\text{Intermediate}\}$.

The next step is to determine the transition t_{in}, $t_{in} \in \bullet p_h^{(i)}$ whose input place is contained within the sequence of places which define the ith service cycle and change its firing mode. In particular, we only need to modify its forward incidence function to support another token type. That is,

$$I^+(p_h^{(i)}, t_{in})(\text{black}) = 1 \rightarrow I^+(p_h^{(i)}, t_{in})(c) = 1$$

for some (hitherto unused) colour c other than black. Now, we need to accommodate the introduction of the new colour c in the model so that the required behaviour can be achieved, e.g. in the latter example, the new token of colour c placed in $p_h^{(0)}$ must be routed towards the Repository place. This is performed in the following way.

First, we identify the transition t_{out} in $p_h^{(i)}\bullet$ whose output place p_k is not contained within the sequence of places which make up the service cycle i, and change both its backward and forward incidence functions accordingly, i.e.

$$I^-(p_h^{(i)}, t_{out})(\text{black}) = 1 \rightarrow I^-(p_h^{(i)}, t_{out})(c) = 1$$

$$I^+(p_k, t_{out})(\text{black}) = 1 \rightarrow I^+(p_k, t_{out})(c) = 1.$$

We proceed by recursively changing the backward and forward incidence functions of all subsequent connected transitions in the non-parameterised model; we consider two transitions t_i, t_j to be connected if $t_i \bullet \cap \bullet\, t_j \neq \emptyset$. This process continues until we reach the transition whose output place is the Repository. For this transition we only change its backward incidence function since we wish the original colour of the token to be restored in the Repository. Table 6.2

lists the required changes of the transitions' firing mode for our example.

Transition	Original Firing Mode		Modified Firing Mode	
t_4	$I^-(\text{Server } 0, t_4)(\text{black}) = 1$	$I^+(\text{Interm.}, t_4)(\text{black}) = 1$	$I^-(\text{Server } 0, t_4)(\text{red}) = 1$	$I^+(\text{Interm.}, t_4)(\text{red}) = 1$
t_7	$I^-(\text{Travel } 2, t_7)(\text{black}) = 1$	$I^+(\text{Server } 0, t_7)(\text{black}) = 1$	$I^-(\text{Travel } 2, t_7)(\text{black}) = 1$	$I^+(\text{Server } 0, t_7)(\text{red}) = 1$
t_{10}	$I^-(\text{Interm.}, t_{10})(\text{black}) = 1$	$I^+(\text{Rep.}, t_{10})(\text{black}) = 1$	$I^-(\text{Interm.}, t_{10})(\text{red}) = 1$	$I^+(\text{Rep.}, t_{10})(\text{red}) = 1$

Table 6.2: The required firing mode modifications, if any, for each transition of the PNPM shown in Figure 6.6 to enable the accurate modelling of the underlying system's customer flow (Figure 6.5). Interm. and Rep. denote the places labelled as Intermediate and Repository.

6.2.2 Integration with Multiple Customer Classes

The approach presented in Section 6.2.1 assumed the presence of one customer class in the system. In Section 6.1 we described the modifications of the data processing pipeline to support multiple customer classes and their representation in the coloured PNPM. Unfortunately, it transpires that the developed approach for detecting cycles contained in the underlying graph of our model is not applicable when multiple customer classes exist. Here, we demonstrate the reason the latter approach fails and propose a solution.

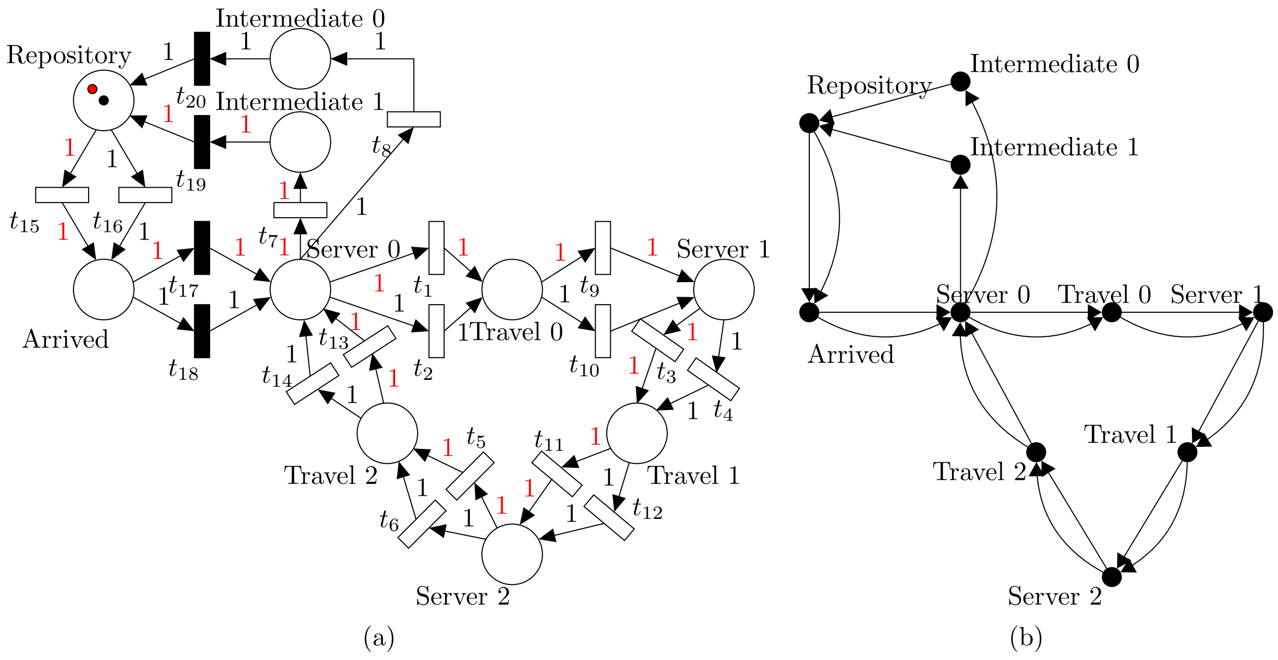

Intermediate 0
Repository
Intermediate 1
Arrived
Server 0
Server 1
Travel 0
Travel 1
Travel 2
Server 2
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12} t_{13} t_{14} t_{15} t_{16} t_{17} t_{18} t_{19} t_{20}
(a)
(b)

Figure 6.10: Figure 6.10(a) shows the CGSPN model (non-parameterised version) for the system depicted in Figure 6.5 using our existing methodology, when two customer classes exist. Its corresponding 2-graph is shown in Figure 6.10(b).

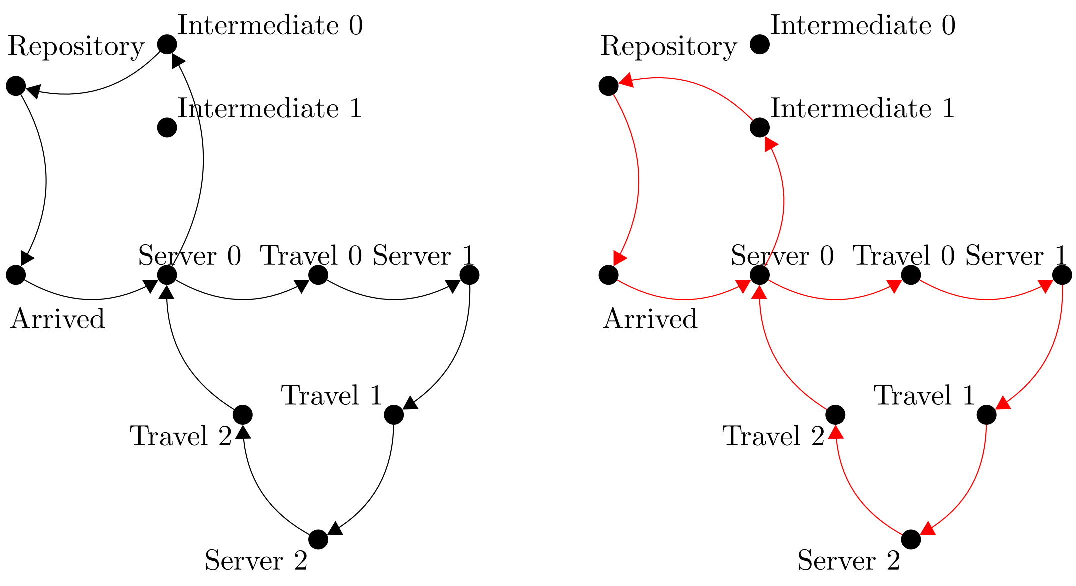
Consider the customer-processing system shown in Figure 6.5 and assume the existence of two customer classes. Following the methodology presented in Section 6.1 we obtain the CGSPN model depicted in Figure 6.10. If we were to map the places and transitions of that model to vertices and edges to form the corresponding directed graph (as described in Section 6.2.1), we would obtain a directed multigraph, also known as p-graph with $p = 2$ (see Figure 6.10(b)). In a p-graph, p denotes the maximum number of arcs having the same initial and terminal vertices. If we have a system with three customer classes where each class performs the same service cycle, the corresponding graph of the PNPM for that system would be a 3-graph. While Theorem 6.1 is still applicable for this graph, Johnson's algorithm considers the three elementary cycles as being the same and thus outputs only one. The original approach fails since we can neither generalise the detected service cycle to all customer classes, nor deduce the class which performs the service cycle. Even if an alternative approach to represent the customer flow in the model was employed, i.e. one transition with multiple firing modes (one firing mode for each customer class), the problem would remain; in this case we would need to consider each supported firing mode as a different edge in the corresponding graph.

The key idea to resolve this issue is to obtain the directed graph that represents the flow of customers of each class – one graph for each customer class – and then iteratively apply the same approach as before, on each graph. To find these graphs we proceed as follows. Assuming that we have N customer classes in the system, we decompose the set of the model's transitions T into disjoint subsets T_{c_i}, i.e.

$$T = \bigcup_{i=1}^{N} T_{c_i} \ , \bigcap_{i=1}^{N} T_{c_i} = \emptyset,$$

where c_i, $i = 1, \ldots, N$, denotes the token type that a transition's firing mode supports. We note that this decomposition is unique since at this stage of model construction there is a one-to-one correspondence between each customer class and each token type. Of course, when the modelling approach presented in Section 6.2.1 is applied, a one-to-many, in particular $1 : k + 1$, correspondence will exist between each customer class and each token type. Here, k denotes the number of service cycles that are sequentially performed by a particular customer class.

Then, for each T_{c_i} – along with the complete set of places in the model – we apply the same mapping as before to derive the directed graph that corresponds to the journey undertaken



(a) $G_1 = (V, E_{\text{black}})$ – Customer Class 1 (b) $G_2 = (V, E_{\text{red}})$ – Customer Class 2

Figure 6.11: The directed graph produced from the CGSPN model of the system (Figure 6.10(a)), for each supported customer class.

by each customer class. In our example we assumed two classes of customers, represented by black and red tokens respectively, i.e. $N = 2$, $c_1 :=$ black, $c_2 :=$ red. If we denote the set of edges that were formed from the mapping of T_{black} as E_{black}, and similarly those formed from the mapping of T_{red} as E_{red}, then the directed graphs that corresponds to the first and second customer classes are, respectively, $G_1 = (V, E_{\text{black}})$ and $G_2 = (V, E_{\text{red}})$ (see Figure 6.11).

The CGSPN model which is obtained using this extended version of our initial approach, for the latter example, is shown in Figure 6.12.

In Section 6.1 we mentioned that if multiple, say n, customer classes are serviced at a particular service area then we have n sets of service time samples, one for each customer class, associated with that service area. If the same service area is also the head of a service cycle, then $2n$ sets of service time samples will be associated with it. That is, assuming that all customer classes perform the service cycle, one set of service time samples for each customer class and each visit.

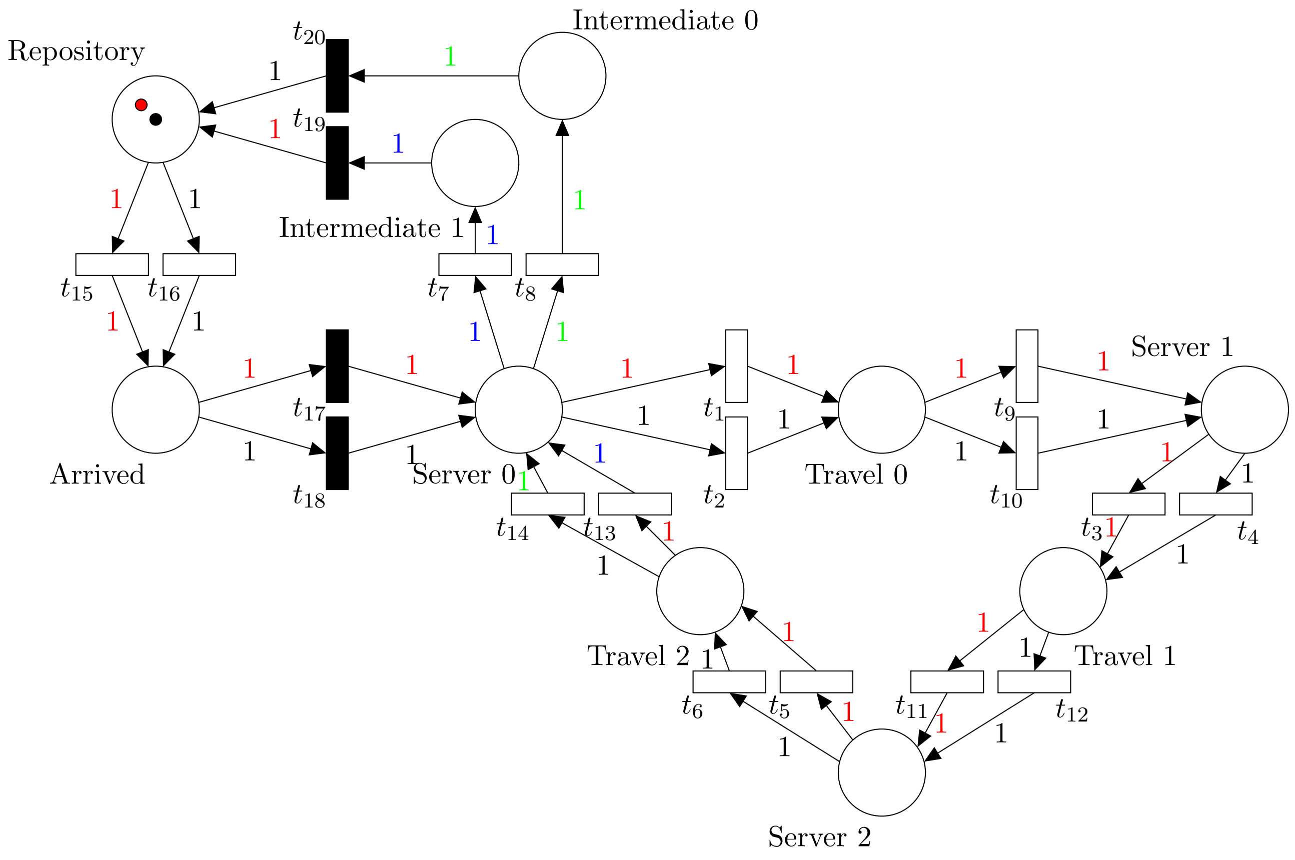


Figure 6.12: The CGSPN model (non-parameterised version) of the system depicted in Figure 6.10(a) using the extended cycle detection approach when two customer classes exist. Black and green tokens are used to distinguish between unserviced and serviced customers of class one. Similarly, red and blue tokens are used to distinguish between unserviced and serviced customers of class two.

6.3 Inter-routing probabilities

In this section we present a straightforward extension of the processing pipeline which allows the calculation and representation of inter-routing probabilities of the system's customer flow.

During the service and travelling time extraction process (Stage 3), we count the total number of customers of each class who received service at each service area. For each service area S_i we maintain tables – one for each customer class – whose tuples consist of two fields: the identity of another service area S_k, $k \neq i$, and the number of customers who visited S_k immediately after they completed their service at S_i.

To calculate the routing probability from S_i to S_k for some k, $k \neq i$, we divide the number of customers of class j who visited S_k immediately after S_i, say $n_{i,k}^{(j)}$, by the total number of customers of the same class who were serviced by S_i, $n_i^{(j)}$. The routing of each customer class

j is represented in the model as follows:

1. For each service area S_i and for each customer class j, an intermediate place is created, where tokens are placed after the associated service time transition fires.

2. For each destination service area S_k of each origin service area S_i, immediate transitions are created from the intermediate places of S_i to a travel place. Their weights are set equal to $n_{i,k}^{(j)}/n_i^{(j)}$.

3. Travelling time transitions are created, connecting each constructed travel place to the corresponding destination service area S_k.

Figure 6.13 shows an example of a part of system that supports only one class of customers and that has multiple routes, as well as the way it is modelled as a GSPN using the latter approach.

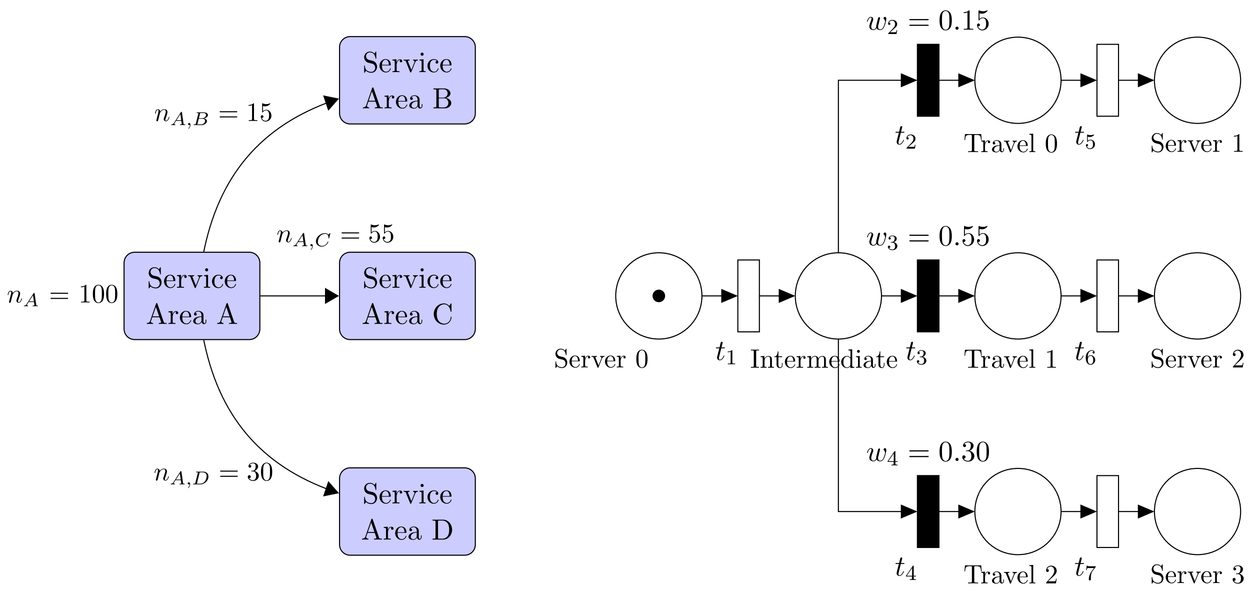


Figure 6.13: Representing the inter-routing probabilities of the customer flow of a system (left) in a Generalised Stochastic PNPM (right). The immediate transitions t_2, t_3 and t_4 have weights equal to the routing probability of service areas B, C and D respectively, i.e. 0.15, 0.55 and 0.30.

Another more compact way of representing the routing probabilities other than using immediate transitions was considered, yet discarded. As we know from the stochastic theory of Petri Nets, when n timed transitions are simultaneously enabled at a particular marking M, the probability of a transition t_i, $i = 0 \ldots n-1$, to fire first is given by $\frac{\lambda_i}{\sum_{j=0}^{n-1} \lambda_j}$ (Equation 2.9). We also note that the sojourn time in M is exponentially distributed with parameter λ, where $\lambda = \sum_{j=0}^{n-1} \lambda_j$ [14].

Thus, one timed transition can be split into several ones with the appropriate firing rates so that the required probability is formed. Given that t_1 fires with rate λ and we wish to split it into three transitions – one for each possible destination – and form the required probabilities, we solve

$$\Lambda = \lambda \cdot P \tag{6.1}$$

where $\Lambda = (\lambda_1, \lambda_2, \lambda_3)$ is the vector of rates, one for each transition and $P = (p_1, p_2, p_3)$ is the vector which contains the probabilities for each transition respectively. In the example depicted in Figure 6.13 we have, $P = (0.15, 0.55, 0.30)$ for destinations Server 1, 2 and 3 respectively. Assume that t_1 fires with rate equal to four, i.e. $\lambda = 4$ customers per second. Using Equation 6.1 we obtain that the rates for the transitions that will be used to replace t_1 are 0.6, 2.2 and 1.2 in respective order. The structure of the updated PNPM can be seen in Figure 6.14.

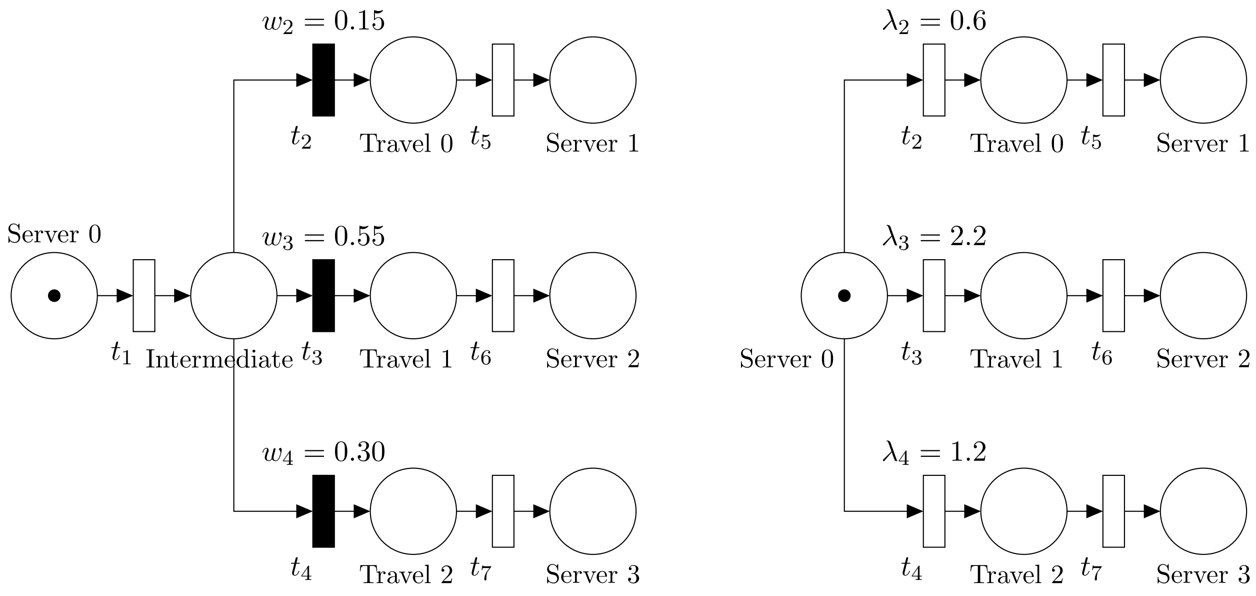


Figure 6.14: Representing the inter-routing probabilities of the customer flow of a system in a Generalised Stochastic PNPM using immediate transitions (left), and by splitting transition t_1 (right). Assume that $\lambda_1 = 4$ customers per second.

However, the timed transitions that are contained in the model at this point (third stage of the pipeline) are not yet parameterised and thus the latter technique cannot be applied. Furthermore, the service and travelling time transitions are subsequently replaced by the GSPN[2] subnets (stage four) which are used to represent the HErDs that were fitted to the extracted service or travelling time samples . The structure of these subnets, e.g. see Figures 4.13 and 4.14, pro-

[2]Could also be CGSPN subnets in the scenario of multiple customer classes, e.g. see Figure 6.4.

hibits the application of the latter technique since each branch of the subnet has an associated probability (the weight of the corresponding immediate transition) and the timed transitions of each branch have fixed firing rates. We conjecture that the technique of splitting the transition rates to represent the routing probabilities of customers cannot be applied in parallel with our existing methodology, at least not while we incorporate the HErD in our models.

Sometimes, during the service and travelling time extraction process, the speed-based heuristic which is used to distinguish the cases where a customer simply passes through a service area without requesting service (cf. Section 4.3), fails. This failure causes the routing of customers to false destinations with a minor probability. To avoid this problem we set a probability cut-off threshold and eliminate destinations whose corresponding routing probability is less than this threshold. The removed probability is equally distributed among the remaining destinations which satisfy the latter condition. The default value of the cut-off threshold is set to 2% but we do provide the user with the option to change it (see Section 7.1). For example, if rare events are important in a customer-processing system, e.g. failures or faulty parts in an assembly line, then the value of this threshold should be reduced.

6.4 Evaluation

In this section we present three case studies and analyse their results. The purpose of these case studies is to evaluate the pipeline's newly added features presented in this chapter, and demonstrate their integration in our methodology. As before, we use the extended version of LocTrackJINQS [56], presented in Chapter 3, to generate the input customer location tracking traces.

6.4.1 Case Studies

For all case studies presented in this section, we assume that each service area consists of a single customer-processing server and a random service discipline, and that the customers' speed within the system, as well as the location update error are normally distributed. The error distribution has a mean of 0.15 m and a standard deviation of 0.2 m, while the parameters of the customers' speed distribution are varied in each case study.

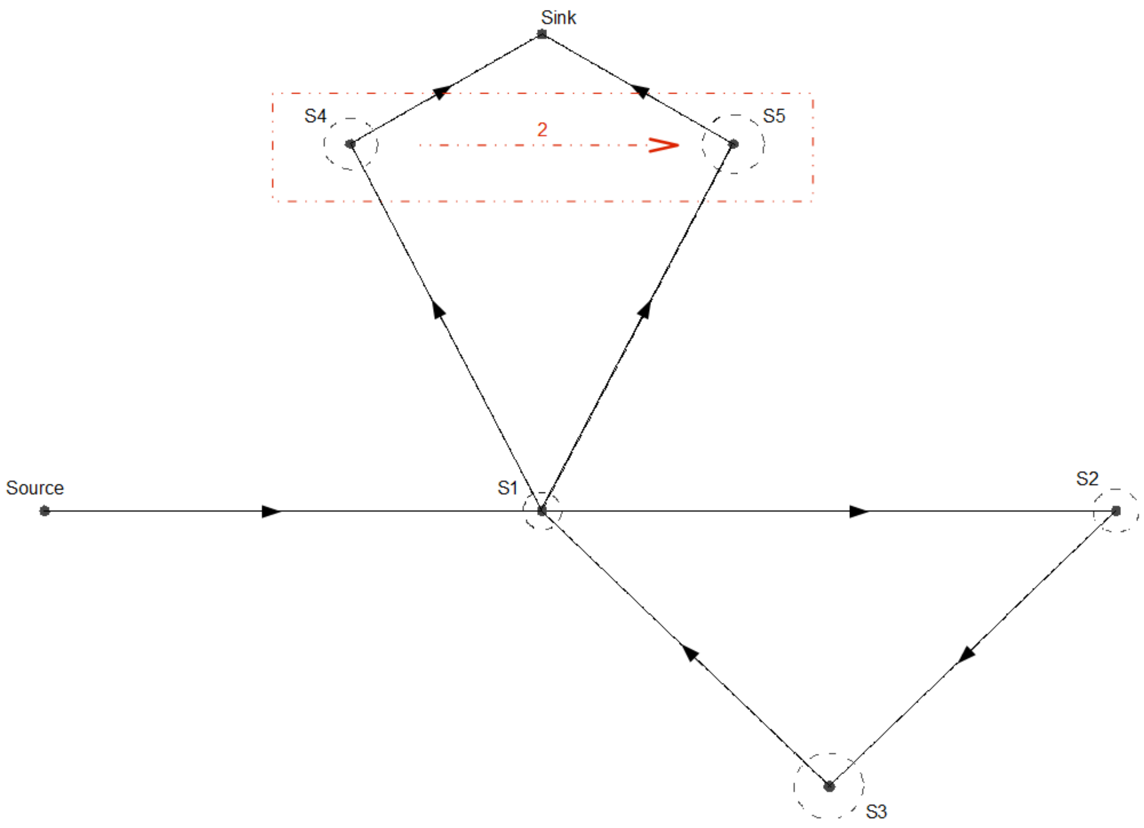


Figure 6.15: The experimental setup in terms of abstract system structure for case study 1. The arrows represent the customer flow in the system and customers initially must complete the service cycle defined by S1, S2, S3 and S1. The service areas contained in the dotted rectangle are subject to a synchronisation condition which is represented by the dotted red arrow. Its source indicates the synchronising service area and its target the service area to be synchronised. The number of customers required to be present in the synchronising service area so that service can be performed in the synchronised service area is indicated by the weight of the dotted red arrow. Furthermore, the branching of the customer flow to S4 and S5 occurs with probability 0.65 and 0.35 respectively.

The experimental setup for the first case study is depicted in Figure 6.15. The simulation takes place in a virtual 35 m × 25 m environment with customer movements as illustrated by the arrows. This customer-processing system contains a service cycle which consists of the service areas S1, S2, S3 and S1, and imposes a synchronisation condition on service area S5. In particular, service area S5 requires the presence of at least two customers in service area S4 in order to service its customers. For this case study we assume one class of customers and that the customer speed distribution has mean 0.5 m/s and standard deviation 0.1 m/s. Furthermore, whenever a customer completes the service cycle, i.e. obtains service from S1 for the second time, he is routed to service area S4 or S5 with probability 0.65 and 0.35 respectively.

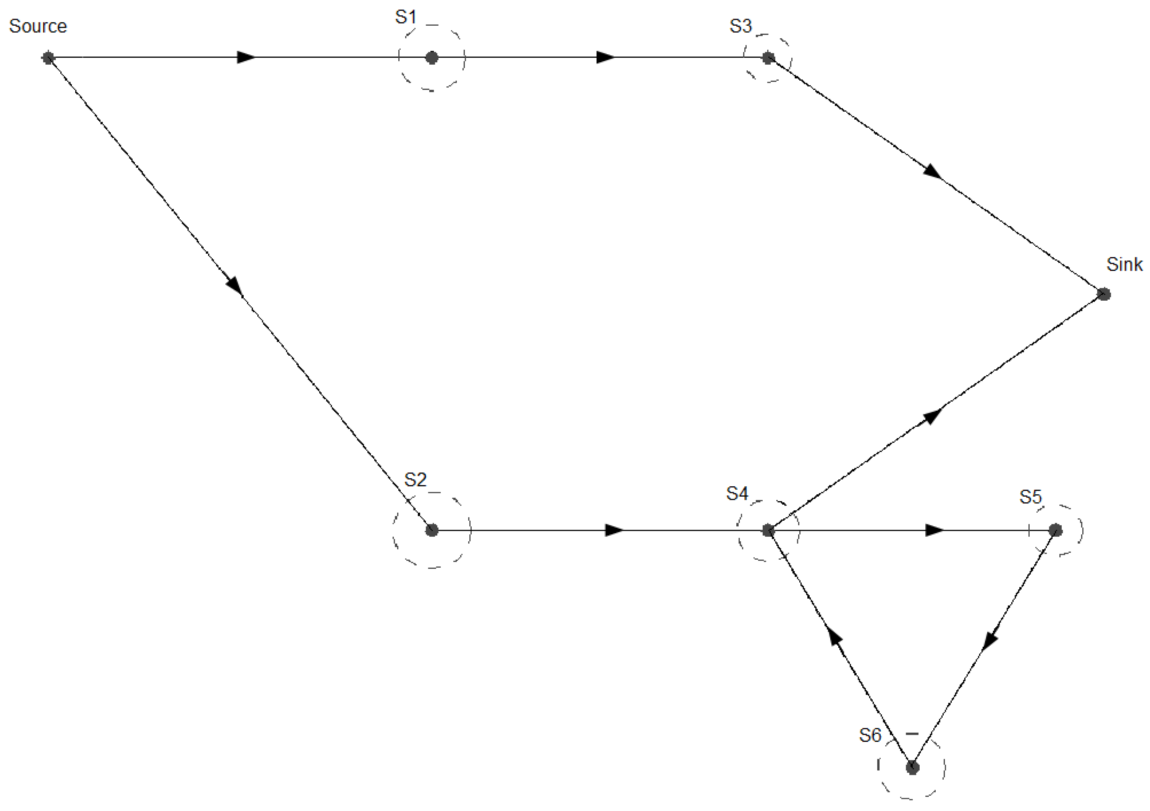


Figure 6.16: The experimental setup in terms of abstract system structure for case study 2. The arrows represent the customer flow in the system and it is independent with respect to the customer's class. The initial routing of customers to S1 and S2 occurs with probability 0.4 and 0.6 respectively, and the service cycle in the system consists of service areas S4, S5, S6 and S4.

The second case study focuses on the integration of multiple customer classes and service cycles. The system is emulated in a $25\,\text{m} \times 25\,\text{m}$ virtual environment and it supports three customer classes. Each customer class has a different speed distribution (see Table 6.3) and similarly, requires a different type of service from each service area in the system. Initially, customers who enter the system are routed to S1 or S2 with probability 0.4 and 0.6 respectively; this does not depend on their class. Service areas S4, S5 and S6 specify a service cycle with S4 being the head. Figure 6.16 shows the experimental setup for this case study.

A simplified model of an accident and emergency department is used for our third case study. This model comprises five service areas: a reception (S1), an examination room (S2), an x-ray operation room (S3), an x-ray room (S4) and a treatment room (S5). The simulation takes place

Customer Class	Mean (m/s)	Standard Deviation (m/s)
0	0.5	0.1
1	0.3	0.1
2	0.65	0.2

Table 6.3: Case study 2: the parameters of the customer speed distribution (Normal) for each customer class.

Customer Class	Mean (m/s)	Standard Deviation (m/s)
0	0.38	0.1
1	0.25	0.1
2	0.4	0.2
3	0.5	0.15

Table 6.4: Case study 3: the parameters of the customer speed distribution (Normal) for each customer class.

in a 40 m $\times$ 28 m virtual environment and the experimental setup is depicted in Figure 6.17. We assume minor and major patient classes that correspond to customer classes zero and one respectively. Furthermore, another two customer classes exist: nurses and radiologists (customer classes two and three). Patients who enter the system are routed to the reception to register and then proceed to the examination room. From there they either get discharged, or are sent to the x-ray room or to the treatment room. This routing of patients occurs with probability 0.3, 0.4 and 0.3 respectively. Patients routed to the x-ray and treatment rooms, after service completion, must revisit the reception to schedule a followup examination and then exit the system. Nurses and radiologists are immediately routed to the x-ray operation room and when their service is terminated exit the system. In order for the scanning process of patients to be initiated in the x-ray room, at least one nurse and one radiologist are required to be present in the x-ray operation room. Table 6.4 shows the speed distribution for each customer class.

The actual location and service radius of each service area, as well as its service time density, for each case study are depicted in Table 6.5.

6.4.2 Results

Results similar to those of chapters 4 and 5 are presented here. In addition, we present the computed inter-routing probabilities (where applicable) and list the service areas that make up each detected service cycle.

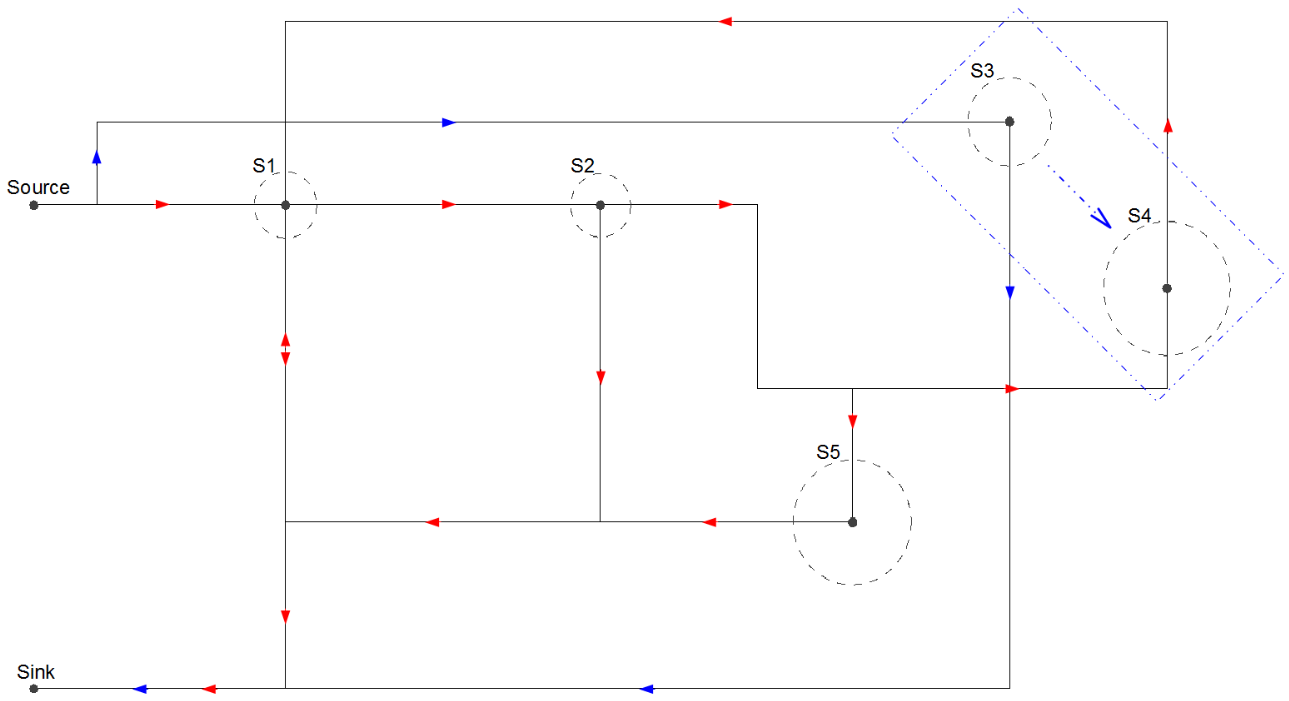


Figure 6.17: The experimental setup in terms of abstract system structure for case study 3. The red arrows represent the flow of customer class zero and one. Blue arrows are used to indicate the flow of customer class two and three. The blue dotted arrow (contained in the blue dotted rectangle) indicates the synchronisation conditions imposed by S3 on S4; one nurse (customer class 2) and one radiologist (customer class 3) must be present in the x-ray operation room (S3) so that the screening process of patients in the x-ray room (S4) can be initiated.

For the first case study, six sets of service time samples were extracted. For case studies two and three, twenty-one and twelve sets of service time samples were extracted respectively and therefore, only a selection is included in this section. In particular, the best-fit HErD results and the corresponding Kolmogorov-Smirnov tests obtained for each service area with respect to the first customer class (class 0) are presented for case study two. Similar results with respect to the second customer class (class 1) are presented for the third case study. Furthermore, for service area S4 in case study two, we include only results regarding the customers' second visit (exit from service cycle). For completeness, the remaining results can be seen in Appendix B.

As in the previous case studies the location and radii of the service areas are in general approximated well (see Table 6.6). The maximum location inference error is $0.084, 0.155$ and 0.320 metres for case studies one, two and three respectively. In the same ordering, the maximum service radius approximation error is $0.148, 0.098$ and 0.277 metres.

		Server Location	Service Radius	Service Time Density	
Case Study 1	S1	$(15.0, 15.0)$	0.5	Erlang$(2, 0.18)$	
	S2	$(30.0, 15.0)$	0.6	Exp(0.06)	
	S3	$(22.5, 22.5)$	0.9	Exp(0.08)	
	S4	$(10.0, 5.0)$	0.7	Erlang$(2, 0.08)$	
	S5	$(20.0, 5.0)$	0.8	Exp(0.12)	
Case Study 2	S1	$(10.0, 5.0)$	0.7	Customer Class 0	Erlang$(2, 0.08)$
				Customer Class 1	Exp(0.08)
				Customer Class 2	Hyper-Exp$(0.5, 0.35, 0.15; 0.05, 0.08, 0.12)$
	S2	(10.0,15.0)	0.8	Customer Class 0	Exp(0.05)
				Customer Class 1	Erlang$(5, 0.15)$
				Customer Class 2	HErD$(1, 3; 0.6, 0.4; 0.02, 0.12)$
	S3	(17.0,5.0)	0.5	Customer Class 0	Erlang$(2, 0.09)$
				Customer Class 1	Exp(0.028)
				Customer Class 2	Erlang$(3, 0.065)$
	S4	(17.0,15.0)	0.65	Customer Class 0	HErD$(1, 5, 2; 0.4, 0.1, 0.5; 0.02, 0.25, 0.12)$
				Customer Class 1	Exp(0.035)
				Customer Class 2	Erlang$(8, 0.2)$
	S5	(23.0,15.0)	0.55	Customer Class 0	Hyper-Exp$(0.5, 0.5; 0.03, 0.08)$
				Customer Class 1	Exp(0.04)
				Customer Class 2	Erlang$(3, 0.1)$
	S6	(20.0,20.0)	0.7	Customer Class 0	Exp(0.025)
				Customer Class 1	Erlang$(2, 0.085)$
				Customer Class 2	Exp(0.05)
Case Study 3	S1	$(10.0, 12.5)$	1.0	Customer Class 0	Exp(0.033)
				Customer Class 1	Exp(0.05)
	S2	(20.0,12.5)	0.95	Customer Class 0	Erlang$(2, 0.035)$
				Customer Class 1	Normal$(40, 10)$
	S3	(33.0,10.0)	1.33	Customer Class 2	Erlang$(6, 0.05)$
				Customer Class 3	Erlang$(5, 0.04)$
	S4	(38.0,15.0)	1.8	Customer Class 0	Exp(0.033)
				Customer Class 1	Exp(0.033)
	S5	(28.0,22.0)	1.5	Customer Class 0	Exp(0.013)
				Customer Class 1	Erlang$(2, 0.013)$

Table 6.5: The parameters for each service area in the system, for each case study. Server locations and their corresponding service radii are given in metres. The parameters of the HErDs represent the phase lengths, weights and rate for each branch respectively, separated by a semi-colon. Using the same notation we represent the weights and rates of the Hyper-Exponential distributions. The unit of the rate parameters for phase-type distributions is customers per second. The mean and standard deviation of the Normal distribution are given in seconds.

The quality of the extracted service time samples and the best-fit HErD are assessed by Kolmogorov-Smirnov tests which examine their compatibility (see Table 6.8). As before, we

		Server Location			Service Radius		
		Real	Inferred (3 d.p.)	Error (3 d.p.)	Real	Inferred (3 d.p.)	Absolute Error
Case Study 1	S1	(15.0, 15.0)	(15.014, 14.977)	0.027	0.5	0.580	0.080
	S2	(30.0, 15.0)	(30.000, 14.938)	0.062	0.6	0.693	0.093
	S3	(22.5, 22.5)	(22.420, 22.508)	0.080	0.9	0.982	0.082
	S4	(10.0, 5.0)	(9.970, 5.026)	0.040	0.7	0.809	0.109
	S5	(20.0, 5.0)	(20.069, 4.952)	0.084	0.8	0.948	0.148
Case Study 2	S1	(10.0, 5.0)	(9.893,5.112)	0.155	0.7	0.789	0.089
	S2	(10.0, 15.0)	(10.009, 14.985)	0.017	0.8	0.889	0.089
	S3	(17.0, 5.0)	(17.010, 5.007)	0.012	0.5	0.598	0.098
	S4	(17.0, 15.0)	(17.007, 14.991)	0.011	0.65	0.735	0.085
	S5	(23.0, 15.0)	(23.041, 14.954)	0.062	0.55	0.630	0.080
	S6	(20.0, 20.0)	(19.983, 19.881)	0.120	0.7	0.795	0.095
Case Study 3	S1	(10.0, 12.5)	(9.955, 12.511)	0.046	1.0	1.095	0.095
	S2	(20.0, 12.5)	(20.014, 12.546)	0.048	0.95	1.025	0.075
	S3	(33.0, 10.0)	(33.017, 10.014)	0.022	1.33	1.454	0.124
	S4	(38.0, 15.0)	(37.814, 14.976)	0.188	1.8	2.077	0.277
	S5	(28.0, 22.0)	(28.136, 22.290)	0.320	1.5	1.609	0.109

Table 6.6: The inferred location and service radius for each service area in the system accompanied with the absolute error, for each case study. These values are given in metres.

use the AIC to select the best-fit HErD from an enumeration of all HErDs up to a maximum number of states which depends on the coefficient of variation of the set of extracted time samples[3]. Table 6.7 shows the parameters of the best-fit HErD for each service area and customer class (for which results are included in this section), as well as the corresponding relative entropy values. Again, we observe some discrepancies between the parameters of the theoretical service time density function and the fitted HErD, e.g. S2, S3 – case study 1, S5 – case study 2, S1 – case study 3, but in general, as we can see from Figures 6.18, 6.19 and 6.20 and the

[3]The rule for selecting the maximum number of states for HErD enumeration is described in Section 4.5.2.

corresponding relative entropy values, good fits have been obtained. In some of the the cases where the best-fit HErD is not as accurate, e.g. Figures 6.19(c), 6.19(d), B.4(b), we observe a relatively small number of samples. Naturally, the number of available samples depends on the number of customers who visited that particular service area. In turn, this number depends on several factors such as the topology of the system, e.g. service areas which are the furthest away from other service areas or they are visited last will likely have fewer service time samples, the system's customer flow, e.g. probabilistic routing of customers to several service areas immediately diminishes the number of service time samples for each destination service area, the duration of the system's monitoring period, etc.

In the first case study the service cycle formed by service areas S1, S2 and S3 has been correctly identified and modelled (see Figure 6.21). In the constructed model, service areas S1, S2, S3, S4 and S5 correspond to the places labeled as Server 0, Server 2, Server 4, Server 1 and Server 3 in respective order. The computed inter-routing probabilities from S1 to destination service areas S4 and S5, represented by the weights of the immediate transitions T2 and T4, are approximated well with values 0.669 and 0.331 (3 d.p.) respectively. Furthermore, the synchronisation condition between S4 (synchronising service area) and S5 (synchronised service area) has been included in the model with the appropriate synchronisation marking, i.e. two customers are required to be present in S4 so that customers in S5 can be serviced.

The constructed PNPM for the second case study is illustrated in Figure 6.22. We observe that the structure of the model matches the abstract system structure of the model used in the simulation. Service areas S1, S2, S3, S4, S5 and S6 correspond to the places with labels Server 4, Server 0, Server 5, Server 1, Server 2 and Server 3 in the constructed model. The service areas S4, S5 and S6 which make up the service cycle have been correctly detected. Furthermore, the subnets (shown in compact transition form) – one for each customer class – which transfer the tokens back to S4 to complete the service cycle, successfully change the colour of the appropriate token class and thus, accurately represent the customer flow of the simulated system. Black (customer class 0) tokens become purple, red tokens (customer class 1) become green and blue tokens (customer class 2) become turquoise. The simulated and computed initial routing probability of the customer flow in the system is presented in Table 6.9.

		Service Time Density	Fitted HErD Parameters			Relative Entropy
			Phase Lengths	Rate (3 d.p.)	Weights (3 d.p.)	(3 d.p.)
Case Study 1	S1 - (1st visit)	Erlang$(2, 0.18)$	2	0.175	1.000	0.001
	S1 - (2nd visit)	Erlang$(2, 0.18)$	2	0.172	1.000	0.002
	S2	Exp(0.06)	$2, 2$	$0.078, 0.324$	$0.644, 0.356$	0.057
	S3	Exp(0.08)	$2, 2$	$0.121, 0.472$	$0.721, 0.279$	0.057
	S4	Erlang$(2, 0.08)$	2	0.076	1.000	0.003
	S5	Exp(0.12)	1	0.097	1.000	0.023
Case Study 2	S1	Erlang$(2, 0.08)$	2	0.085	1.000	0.003
	S2	Exp(0.05)	1	0.045	1.000	0.005
	S3	Erlang$(2, 0.09)$	1	0.036	1.000	0.142
	S4 - (2nd visit)	HErD$(1, 5, 2; 0.4, 0.1, 0.5; 0.02, 0.25, 0.12)$	$1, 3$	$0.013, 0.182$	$0.426, 0.574$	0.037
	S5	Hyper-Exp$(0.5, 0.5; 0.03, 0.08)$	2	0.085	1.000	0.324
	S6	Exp(0.025)	2	0.048	1.000	0.193
Case Study 3	S1 - (1st visit)	Exp(0.05)	$2, 2$	$0.084, 1.184$	0.899,0.101	0.054
	S1 - (2nd visit)	Exp(0.05)	$1, 2$	$0.070, 0.088$	$0.325, 0.675$	0.045
	S2	Normal$(40, 10)$	17	0.385	1.000	0.119
	S3	N/A	N/A	N/A	N/A	N/A
	S4	Exp(0.033)	1	0.027	1.000	0.016
	S5	Erlang$(2, 0.013)$	2	0.013	1.000	0.001

Table 6.7: The HErD parameters fitted by G-FIT for each service area's service time density with the relative entropy (in nat) between the theoretical and fitted probability density function, for each case study. for each case study. For case studies two and three we show the results only for the first and second customer class respectively. The parameters of the HErDs represent the phase lengths, weights and rate for each branch respectively, separated by a semi-colon.

Figure 6.23 depicts the model obtained for case study three. Customer classes 0, 1, 2 and 3 are represented with red, blue, green and black colours respectively. Places Server 0, Server 1, Server 3, Server 2 and Server 4 correspond to service areas S1 (reception), S2 (examination

		Case Study 1		Case Study 2 - Class 0		Case Study 3 - Class 1	
S1 - (1st visit)	Test Statistic	0.0398		0.0787		0.0638	
	α	0.1	0.05	0.1	0.05	0.1	0.05
	Critical Values	0.0723	0.0824	0.1544	0.1761	0.0877	0.0999
	Compatible ?	Yes	Yes	Yes	Yes	Yes	Yes
S1 - (2nd visit)	Test Statistic	0.0564		N/A		0.0422	
	α	0.1	0.05			0.1	0.05
	Critical Values	0.0733	0.0836			0.1110	0.1265
	Compatible ?	Yes	Yes			Yes	Yes
S2	Test Statistic	0.0238		0.0880		0.0455	
	α	0.1	0.05	0.1	0.05	0.1	0.05
	Critical Values	0.0720	0.0821	0.1498	0.1708	0.0901	0.1027
	Compatible ?	Yes	Yes	Yes	Yes	Yes	Yes
S3	Test Statistic	0.0257		0.1589		N/A	
	α	0.1	0.05	0.1	0.05		
	Critical Values	0.0728	0.0830	0.1578	0.1799		
	Compatible ?	Yes	Yes	No	Yes		
S4	Test Statistic	0.0636		0.0441		0.1293	
	α	0.1	0.05	0.1	0.05	0.1	0.05
	Critical Values	0.0898	0.1024	0.1578	0.1799	0.1443	0.1645
	Compatible ?	Yes	Yes	Yes	Yes	Yes	Yes
S5	Test Statistic	0.1049		0.1010		0.1011	
	α	0.1	0.05	0.1	0.05	0.1	0.05
	Critical Values	0.1317	0.1502	0.1513	0.1725	0.1671	0.1905
	Compatible ?	Yes	Yes	Yes	Yes	Yes	Yes
S6	Test Statistic	N/A		0.1325		N/A	
	α			0.1	0.05		
	Critical Values			0.1513	0.1725		
	Compatible ?			Yes	Yes		

Table 6.8: Kolmogorov-Smirnov test at significance levels 0.1 and 0.05 applied to the extracted service time samples for each service point from case studies one, two and three. The null hypothesis is that each extracted sample belongs to the corresponding best-fitted HErD.

room), S3 (x-ray operation room), S4 (x-ray room) and S5 (treatment room). The two service cycles involving service areas S1, S2, S4 and S1, S2, S5 have been correctly identified and modelled. Subnets with labels HErD21 and HErD15 connect S4 and S5 (via their travel places) back to S1 to complete the service cycles of customers of class 0 and similarly, subnets HErD19 and HErD13 complete the same service cycles of customers of class 1. We note the token colour change for the first customer class (class 0) from red to brown (by HErD21 and HErD15) and for the second customer class (class 1) from blue to yellow (by HErD19 and HErD13), which correctly models the underlying system's customer flow. The inferred routing probability from S2 to S4, S5 and sink (represented by the repository place) is shown in Table 6.10. In addition,

Destination Service Area	Customer Class	Simulated Probability	Inferred Probability (3 d.p.)
S1	0	0.4	0.485
	1	0.4	0.333
	2	0.4	0.470
S2	0	0.6	0.515
	1	0.6	0.667
	2	0.6	0.530

Table 6.9: The simulated and inferred initial routing probability of the customer flow, for each customer class, for case study two.

Destination Service Area	Customer Class	Simulated Probability	Inferred Probability (3 d.p.)
S4	0	0.4	0.429
	1	0.4	0.397
S5	0	0.3	0.280
	1	0.3	0.298
Sink	0	0.3	0.291
	1	0.3	0.305

Table 6.10: The simulated and inferred routing probability of the customer flow from service area S2 for case study three.

the synchronisation conditions between S3 (synchronising service area) and S4 (synchronised service area) have been modelled correctly; one nurse (green colour) and one radiologist (black colour) are required to be present in S3 so that patients in S4 can be scanned.

In Chapter 4 (cf. Section 4.5.2) we have discussed the possibilities of model analysis in order to assess the underlying system's performance and the modification of the constructed models to perform "what-if" analysis. Before we conclude this chapter, we must note the impact of the presented extensions on the complexity of model analysis. In particular, analysis is likely to be bedevilled by a state-space explosion caused by:

1. the use of different token types to support multiple customer classes, and

2. the exponential increase in the number of HErD subnets used to replace the non-parameterised service and travelling time transitions for each customer class.

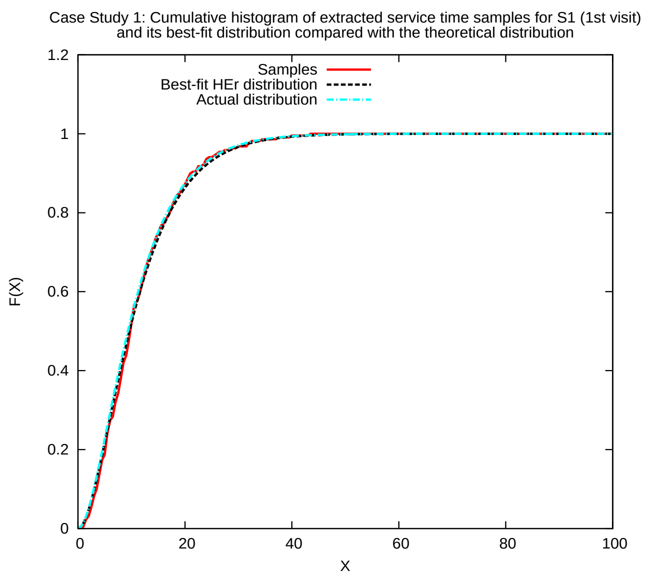
Possible techniques that could be employed to mitigate this problem are discussed in Section 8.3.

6.4.3 Conclusion

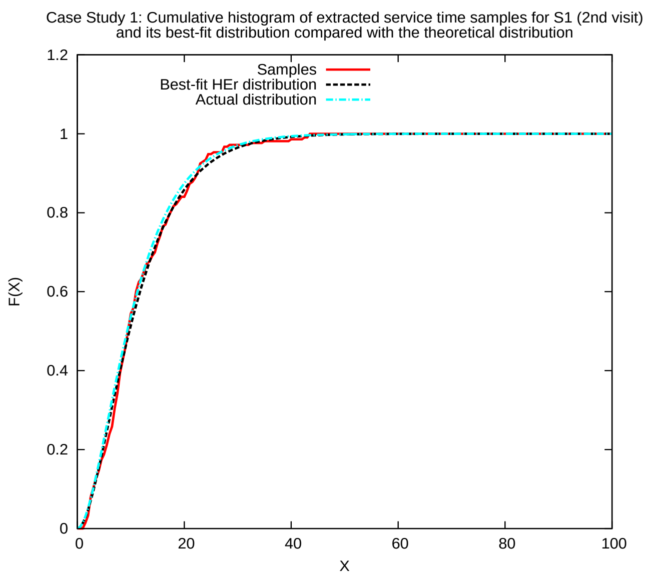
In this chapter we have presented three extensions, two of which have been made possible by the introduction of CGSPNs, which extend the domain of applicability and modelling power of our original methodology.

The first extension discards the assumption of single customer class as it enables the support of multiple customer classes. Each customer class is assigned one token colour and a distinct set of transitions to model its flow throughout the model. The second extension captures the presence of single service cycles in the system (if they exist) and models them in such a way so that the flow customers is accurately represented. The model (before parameterisation) is converted into a directed graph on which we apply Johnson's algorithm [64] which enumerates all elementary cycles contained in the graph. We then identify the appropriate transitions (see Section 6.2.1) which are connected to the place that represents the head of the service cycle and change the assigned token type of their firing mode so that any race conditions between them are eliminated. This action guarantees that the customer flow of the underlying is accurately reflected by the model. The final extension of the pipeline allows the calculation and representation of the inter-routing probabilities of the customer flow between service areas.

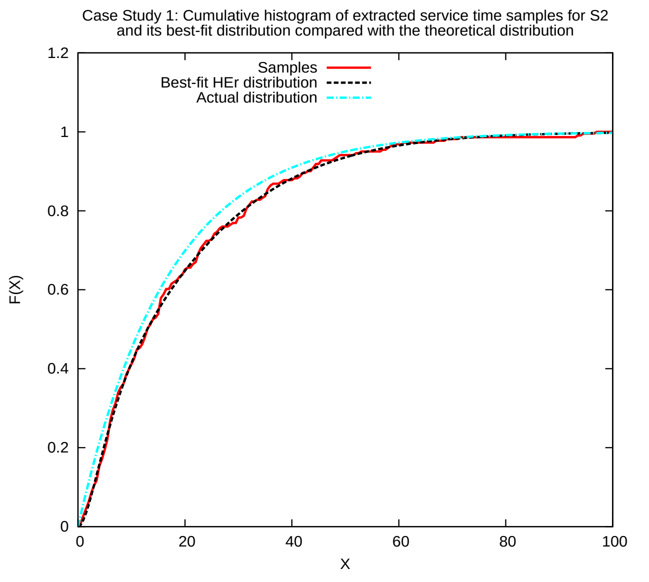
These extensions and their integration with the existing methodology presented in Chapters 4 and 5 are evaluated in case studies one, two and three. In particular, the third case study examines all supported features of our approach, i.e. the presence of synchronisation and service cycles, in parallel with multiple customer classes. The corresponding results provide evidence which reinforces the validity of earlier results, and suggests that these extensions perform well. Furthermore, the integration of these extensions with our earlier work has been successful and in fact, the integrated methodology is implemented under a unified framework by the PEPERCORN tool; this is presented in the next chapter.



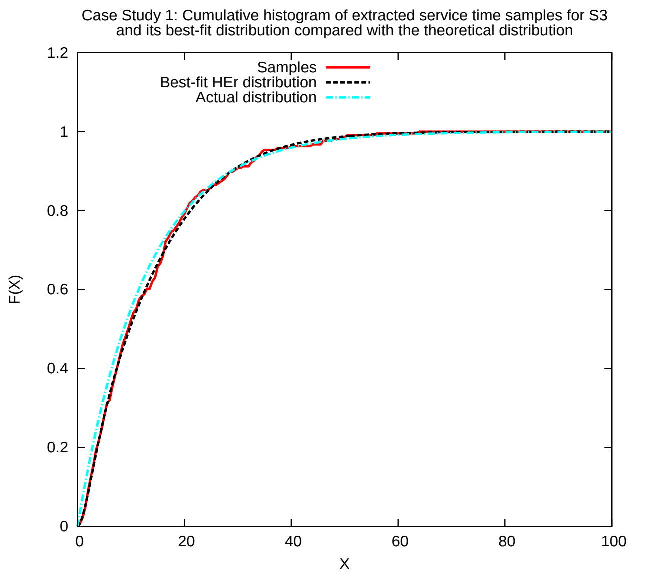
(a)



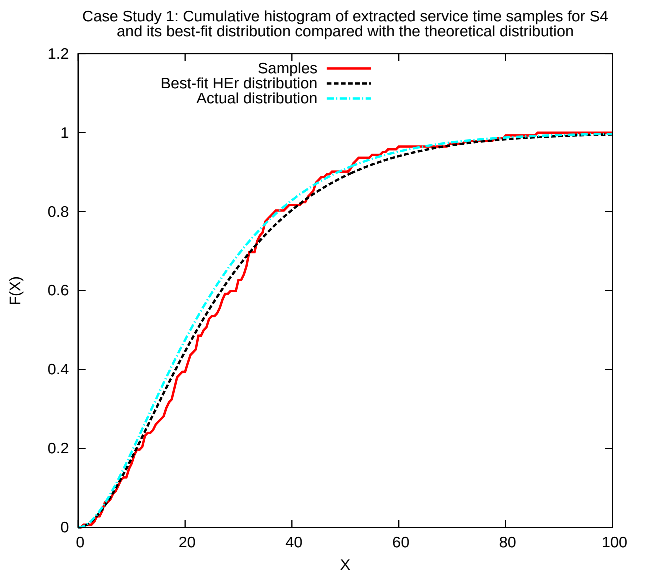
(b)



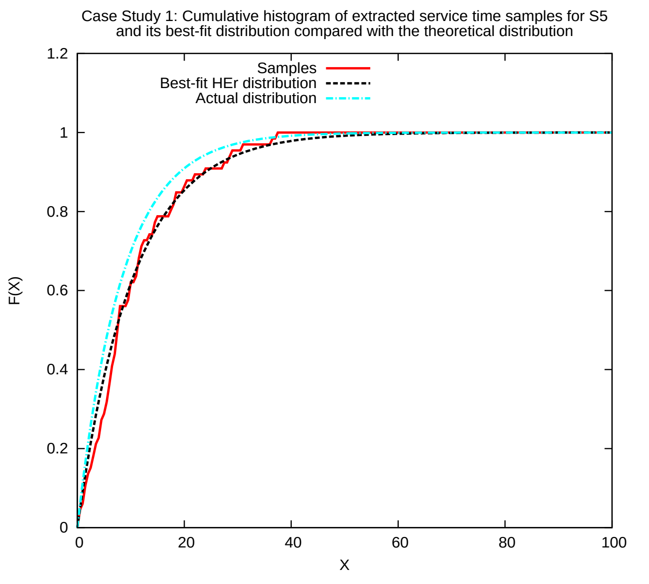
(c)



(d)

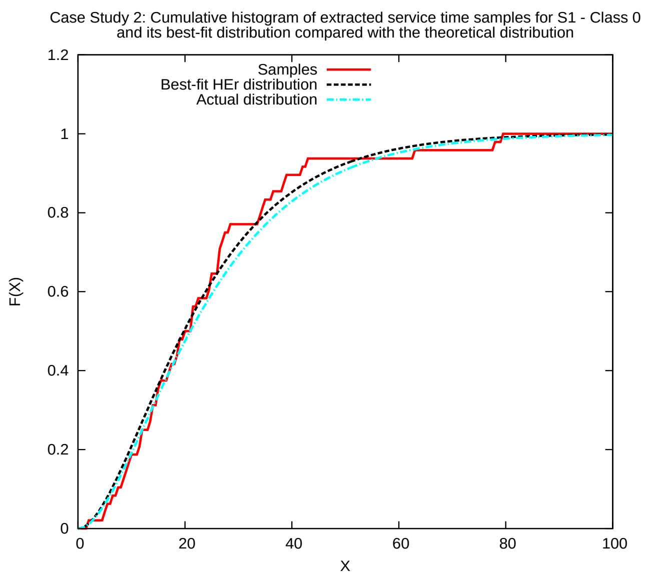


(e)

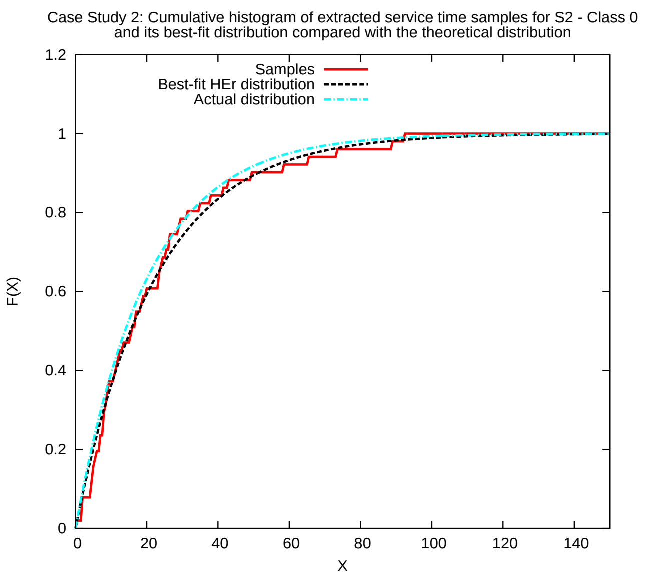


(f)

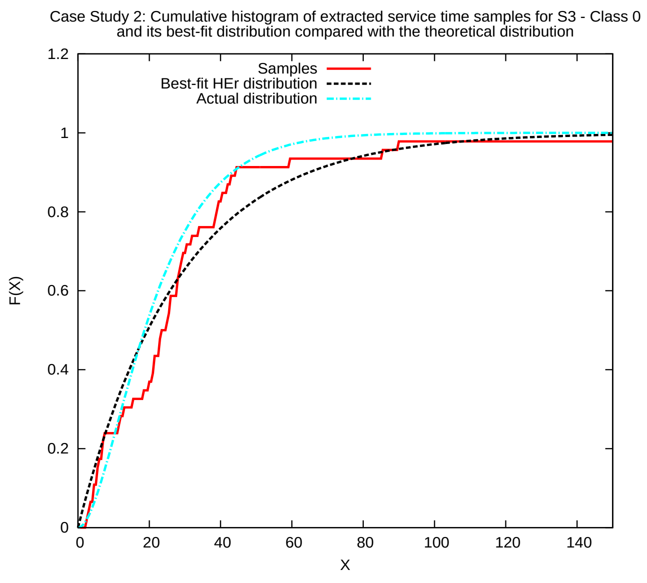
Figure 6.18: Case Study 1: Graphs 6.18(a), 6.18(b), 6.18(c), 6.18(d), 6.18(f) and 6.18(f) show the cumulative histogram of the extracted service time samples (adjusted for synchronisation in 6.18(f)) and its best-fit hyper-Erlang distribution compared with the theoretical distribution for S1 (entry to service cycle), S1 (exit from service cycle), S2, S3, S4 and S5 respectively.



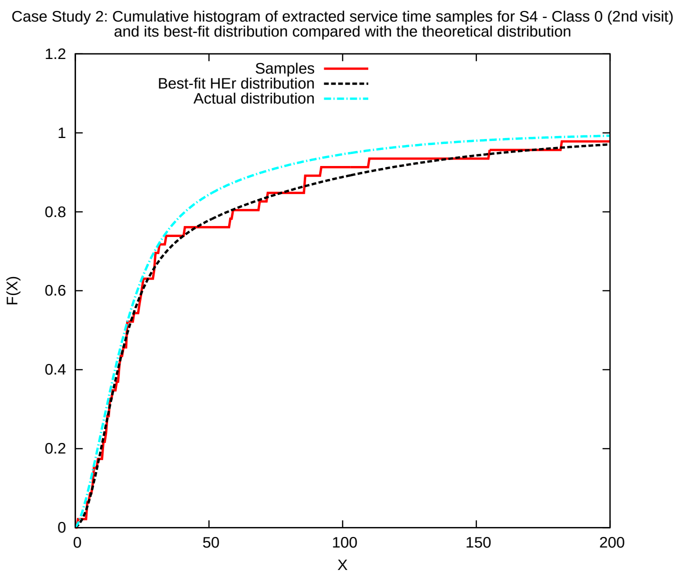
(a)



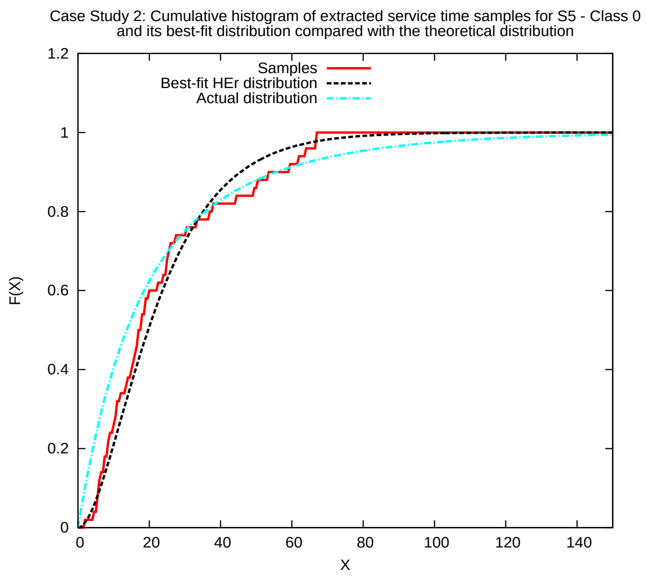
(b)



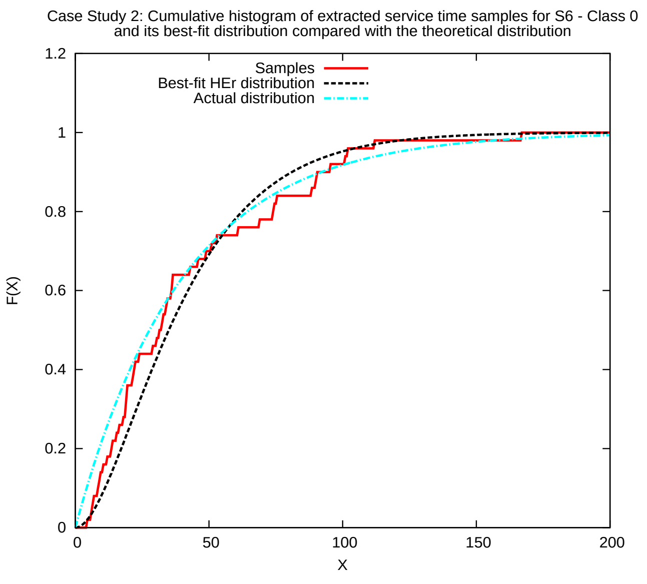
(c)



(d)

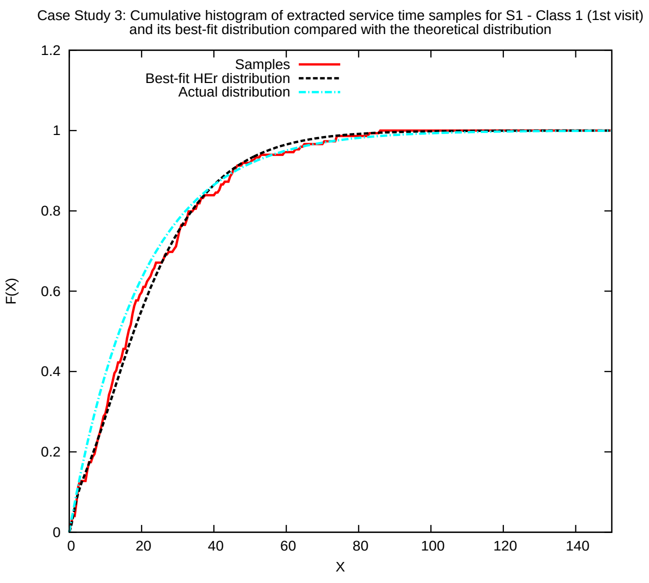


(e)

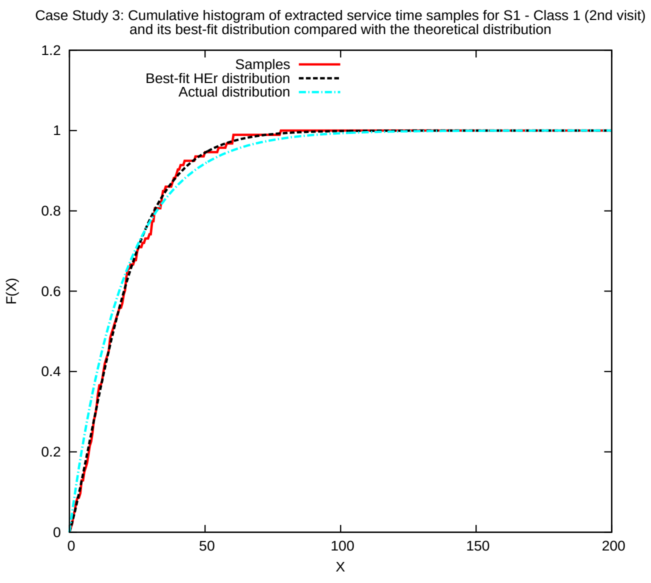


(f)

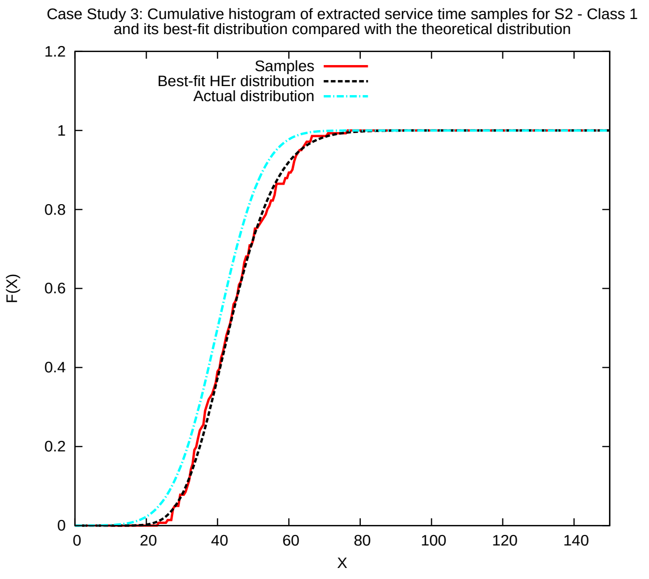
Figure 6.19: Case Study 2: Graphs 6.19(a), 6.19(b), 6.19(c), 6.19(d), 6.19(e) and 6.19(f) show the cumulative histogram of the extracted service time samples for customer class 0 and its best-fit hyper-Erlang distribution compared with the corresponding theoretical distribution for S1, S2, S3, S4 (exit from service cycle), S5 and S6 respectively.



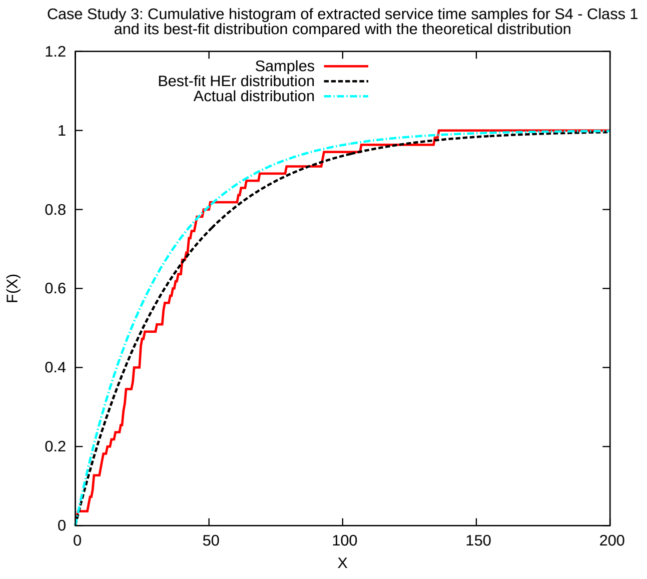
(a)



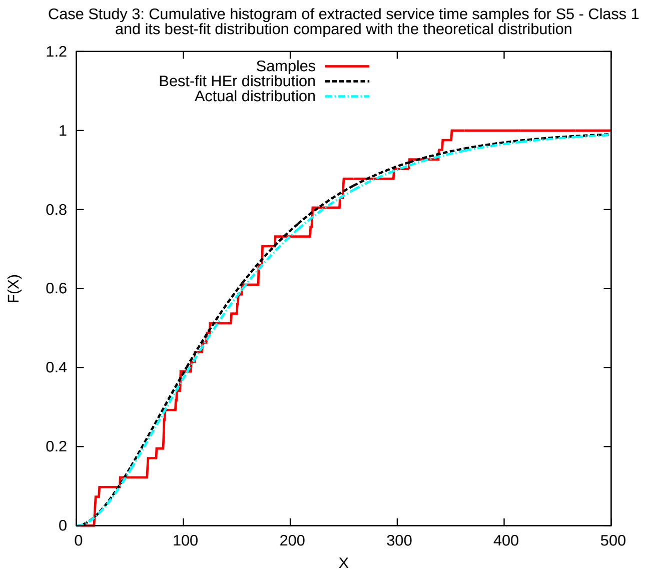
(b)



(c)



(d)



(e)

Figure 6.20: Case Study 3: Graphs 6.20(a), 6.20(b), 6.20(c), 6.20(d) and 6.20(e) show the cumulative histogram of the extracted service time samples (adjusted for synchronisation in 6.20(d)) for customer class 1 and its best-fit hyper-Erlang distribution compared with the corresponding theoretical distribution for S1 (entry to service cycle), S1 (exit from service cycle), S2, S4 and S5 respectively.

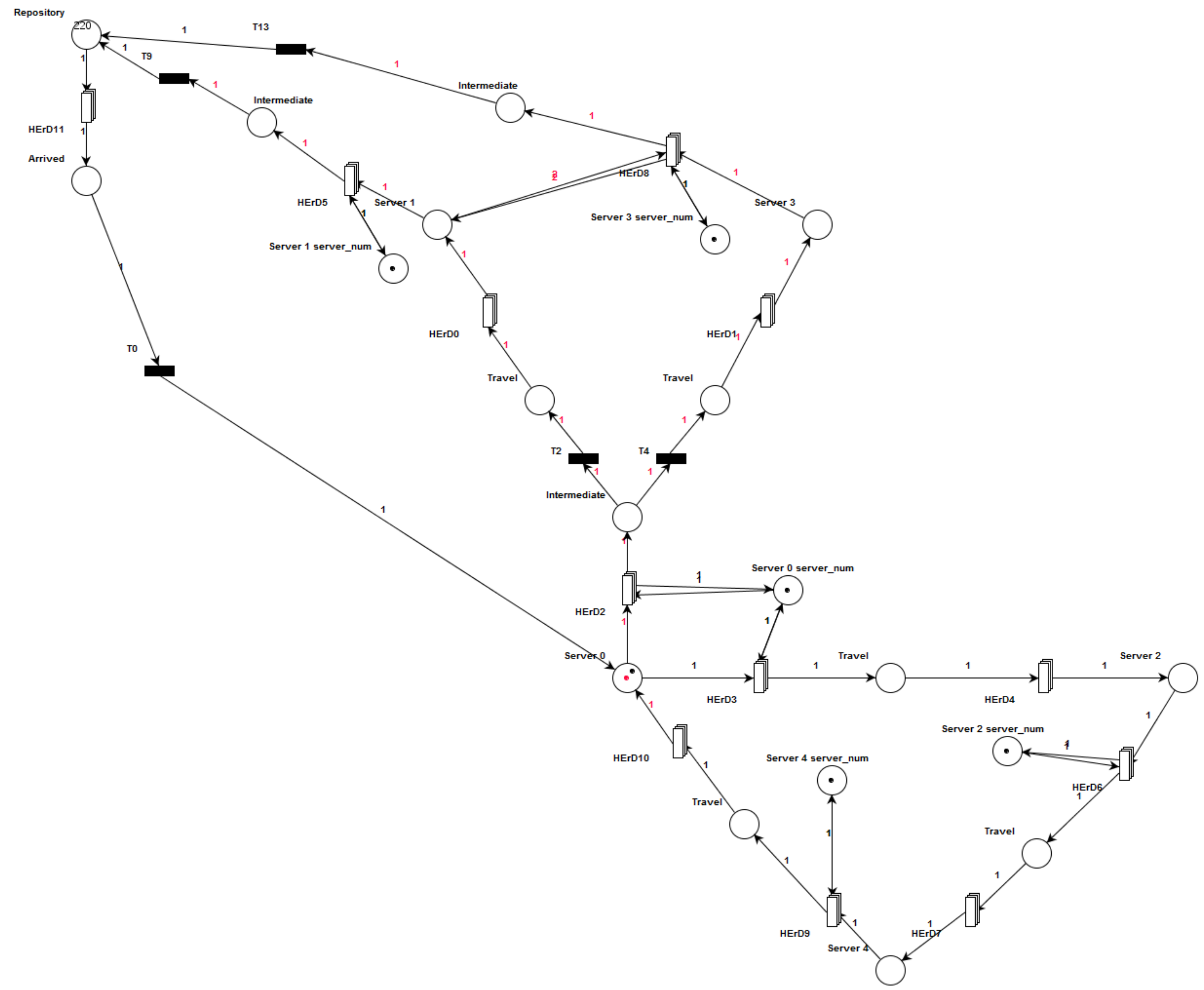


Figure 6.21: The inferred CGSPN performance model for case study one, vitualised in PIPE2 (in compact transition form).

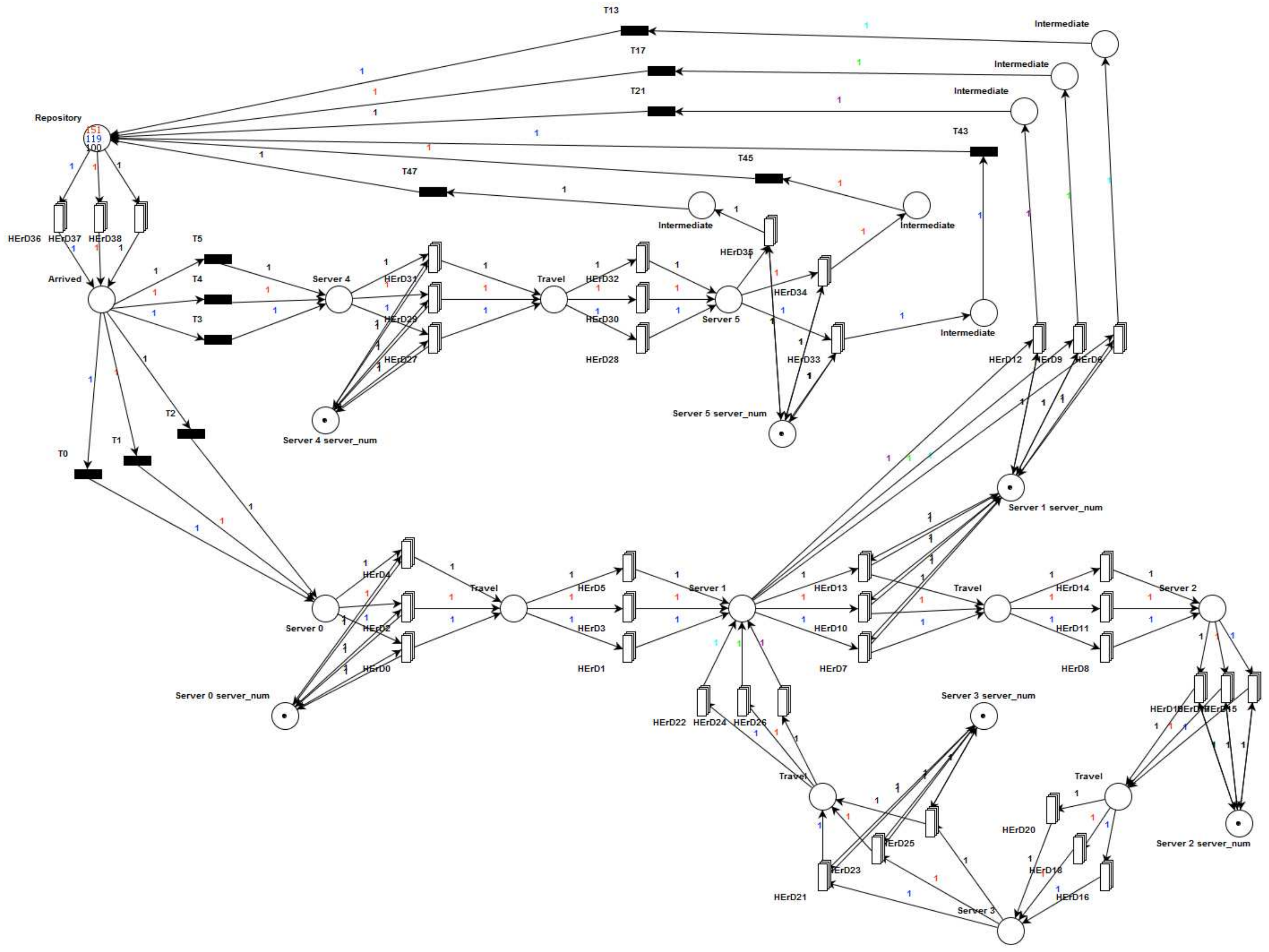


Figure 6.22: The inferred CGSPN performance model for case study two, visualised in PIPE2 (in compact transition form). Black, red and blue tokens represent the first, second and third customer class respectively.

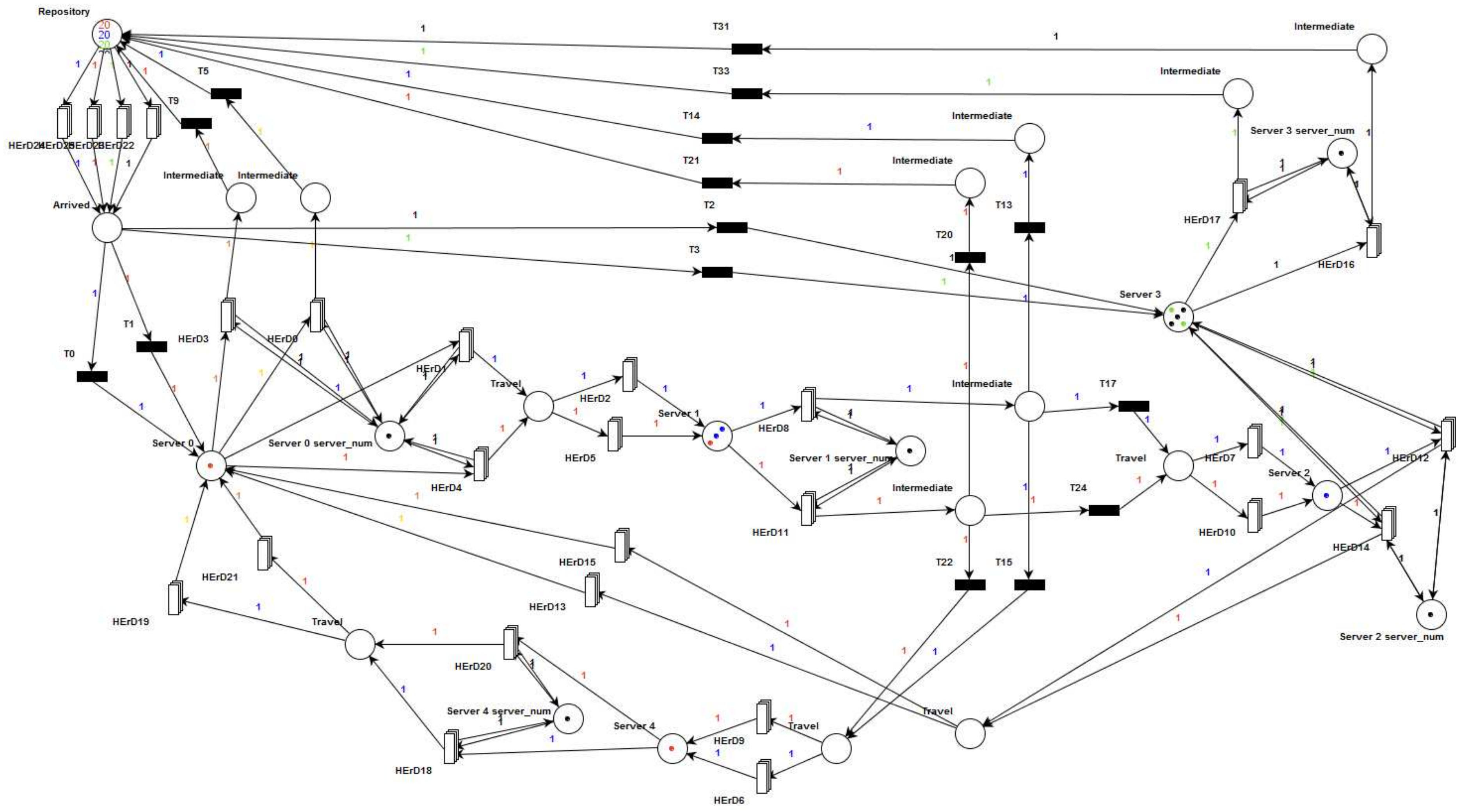


Figure 6.23: The inferred CGSPN performance model for case study three, visualised in PIPE2 (in compact transition form). Red, blue, green and black tokens represent customer class 0, 1, 2 and 3 in respective order.

Chapter 7

PEtri net PERformance model COnstRuctioN tool - PEPERCORN

This chapter presents the *PEPERCORN* tool, developed to demonstrate the feasibility of the automated methodology, presented throughout Chapters 4, 5 and 6, under a unified framework. PEPERCORN provides a user-friendly environment which allows users – who are not necessarily familiar with performance model construction – to construct PNPMs from high-precision location tracking data. This tool has been implemented in Java for two reasons: it is a platform-independent language and it provides good support for GUI development.

The chapter is organised as follows. We first provide an overview of PEPERCORN's interface and demonstrate its operation. In fact, the screen shots we include here were produced while processing the location tracking data generated by simulating the A&E department presented in the third case study of Chapter 6. We then present an outline of PEPERCORN's software architecture and conclude this chapter with a description of the mining process that extracts the data – apart from the locations and radii of the system's service areas – required to construct and parameterise the PNPM.

7.1 Overview

PEPERCORN's main interface, shown in Figure 7.1, comprises of four components: a menu bar, a toolbar, a drawing panel and a status bar. The menu bar contains actions which initialise and execute the processing pipeline, e.g. open file, process file and export PNPM (quick access to these actions is also provided by the toolbar), as well as additional options which allow the user to modify various adjustable parameters of the pipeline.

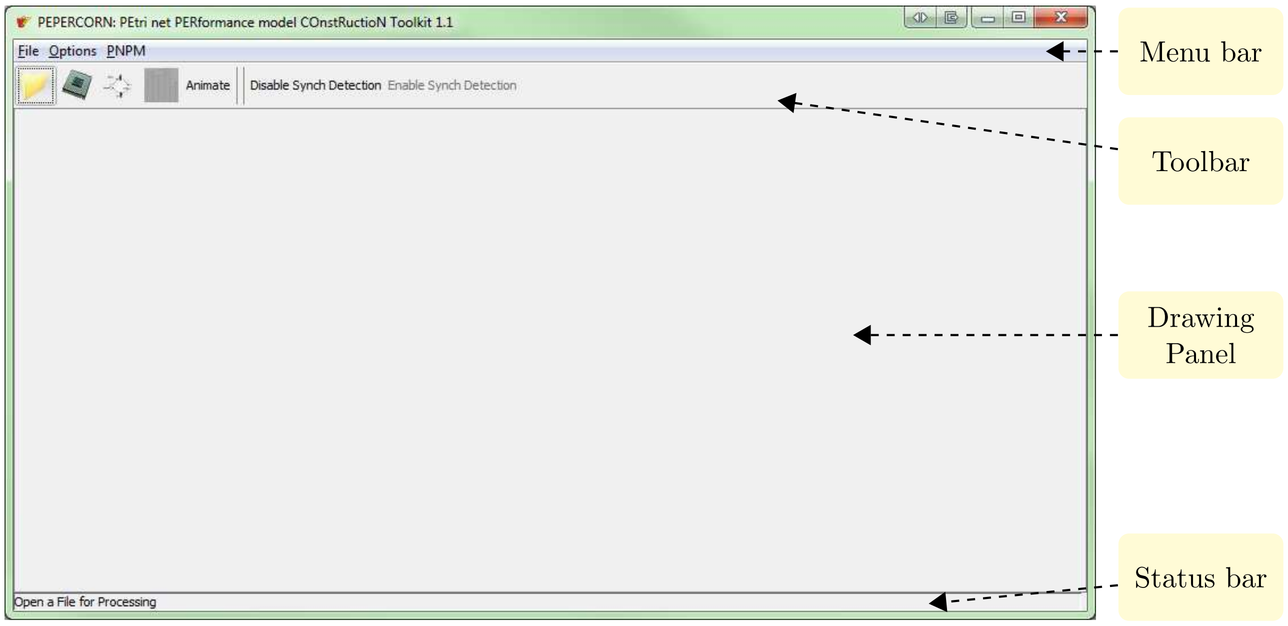


Figure 7.1: The main interface of PEPERCORN.

The status bar guides the user through the various stages of the data processing pipeline; it displays the stage and action that is currently being performed. This also includes a progress bar which shows an estimate of the overall progress of the pipeline (as a percentage) until the PNPM is constructed and ready to be exported (cf. Figure 7.2).

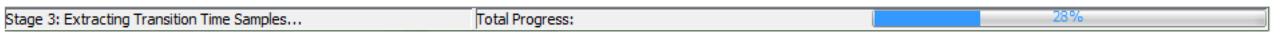
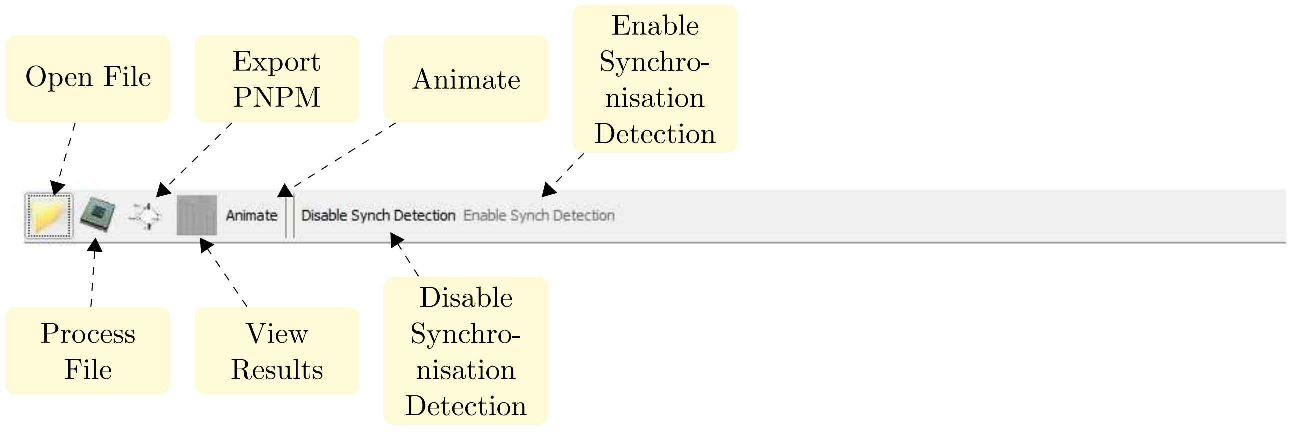
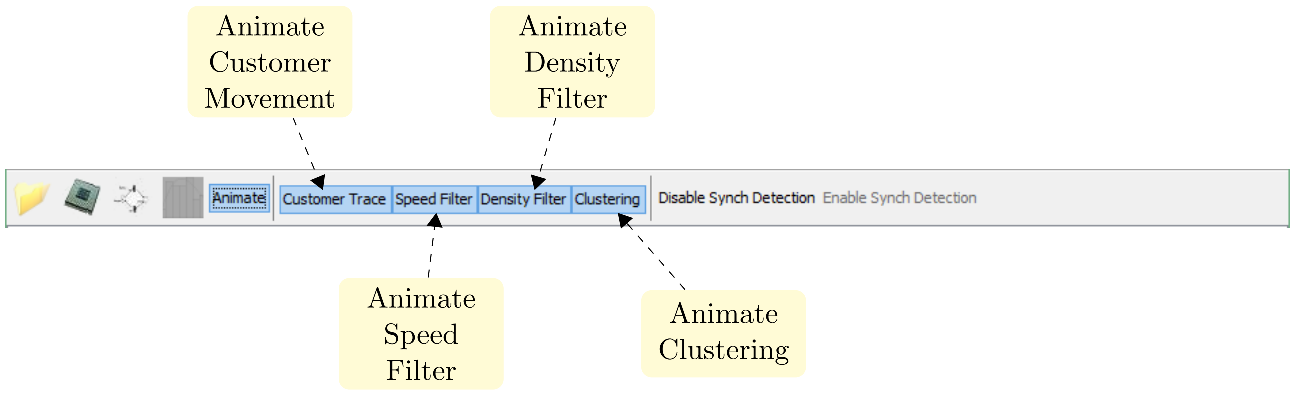


Figure 7.2: PEPERCORN's status bar.

In order to construct a PNPM using PEPERCORN, the user must first import a file which should contain the raw location tracking updates retrieved from a particular customer-processing system. As we mentioned earlier, PEPERCORN currently supports location tracking data



(a) The actions provided by the toolbar.



(b) Additional animation actions which are available once the 'Animate' button is pressed.

Figure 7.3: PEPERCORN's toolbar (above) and its expanded view, once the 'Animate' button is pressed (below).

obtained from a Ubisense UWB-based RTLS and synthetic ones, generated by LocTrack-JINQS [56]. In particular, these files should contain a stream of tuples of the form `(tagName, x, y, z, date, time, milliseconds, stderr, type)` and `(tagName, type, time, x, y, stderr)` respectively. Such files can be imported in PEPERCORN via the 'Open File' button (cf. Figure 7.3), which can be also accessed through the 'File' menu.

Once a supported file has been imported, the user can initiate the data processing pipeline by clicking the 'Process File' button which is also found on the toolbar (cf. Figure 7.3) and under the 'File' menu. However, if the user wishes to adjust some of the pipeline's default parameters or disable the synchronisation detection mechanism, the user must do so before processing commences. Specifically, through the 'Advanced Settings' menu (submenu of 'Options' – cf. Figure 7.4), one can modify the following parameters:

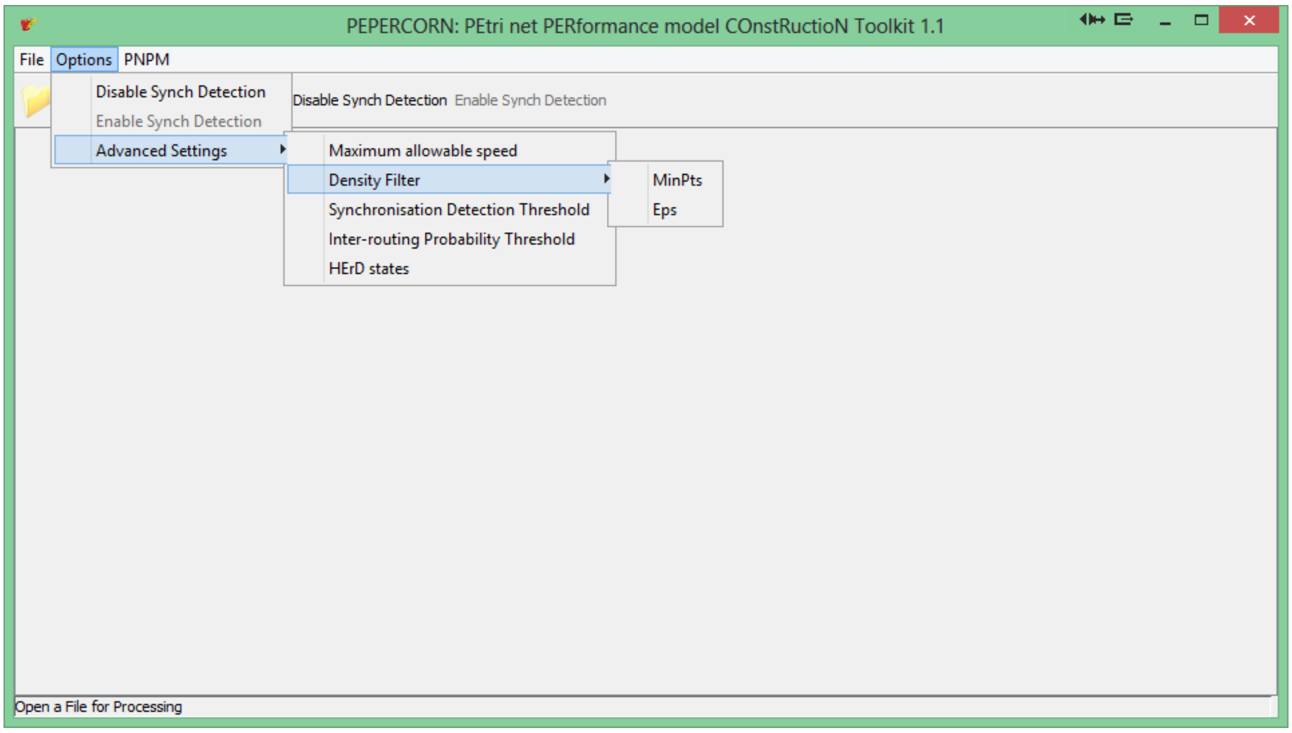


Figure 7.4: The ‘Options’ menu of PEPERCORN’s menu bar.

1. the maximum allowable speed which is used as a threshold to discard erroneous location updates during the initial data filtering process (cf. Section 4.1),

2. the values of $MinPts$ and Eps for the density filter (cf. Section 4.2.2),

3. the value of the synchronisation acceptance threshold s_{thresh} (cf. Section 5.1),

4. the threshold probability used for the calculation of the inter-routing probabilities of the customer flow (cf. Section 6.3), and

5. the maximum number of states of the HErD to be fitted, in cases where the coefficient of variation of the extracted service or travelling time samples is less than 0.4 (see Section 4.4).

The option to disable the synchronisation detection mechanism is provided because false positives may sometimes arise in systems where all customers must sequentially visit several service areas, with long service time, in the same order, e.g. a system with a structure similar to that of the first case study presented in Chapter 4.

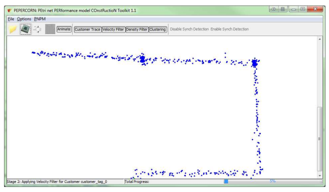
A user can select to animate the different processing phases of the second stage of the data processing pipeline by pressing the 'Animate' button (cf. Figure 7.3); this is displayed on the drawing panel. If this button is pressed before or during the initiation of the second stage of the pipeline, each of the stage's three layers is animated in "real time", i.e. the speed filter, the density filter and the execution of the DBSCAN clustering algorithm. The user can further specify which layer of this stage to animate, including the customers movement before the speed and density filters are applied (see Figure 7.3(b)). If the 'Animate' button is pressed after the second stage has been completed, only the clustering result is displayed (cf. Figure 7.5(e)). An example is shown in Figures 7.5(a), 7.5(b) and 7.5(c).

After the speed and density filters have been applied to all customer paths, the DBSCAN clustering algorithm is applied on the aggregated filtered dataset. However, before clustering initiates, a dialog is displayed to the user; this dialog shows the aggregated dataset and the *4-dist* graph, along with the automatically selected value for *Eps* (cf. Figure 7.5(d)). If the *Eps* value is not (automatically) approximated well, then the user can tune the this value manually by clicking on the desired *4-dist* value of the *4-dist* graph (see Section 4.2.3). After the *Eps* value is specified, no further user input is required.

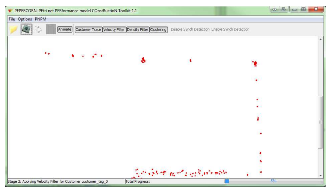
When the execution of the data processing pipeline is completed, the user can export the constructed PNPM in an XML file (a custom variation of the PNML) so it can be visualised and/or analysed in PIPE2 (cf. Figure 7.5(f)). This action is performed by clicking the 'Export PNPM' button on the toolbar or via the 'PNPM' menu.

PEPERCORN also allows users to examine key quantitative results, such as the service and travelling time distribution fits, the compatibility of extracted time samples with the fitted distribution, and service area locations, before exporting the model (see Figure 7.6). This action is performed by the 'View Results' button (located also in the 'PNPM' menu) which becomes enabled as soon as the fourth stage of the pipeline is completed.

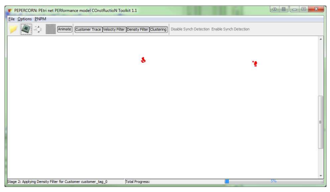
Additional actions are also provided through the application menus. The 'Clear' action, located in the 'PNPM' menu, erases all data stored in memory and deletes temporary files created during the operation of the data processing pipeline, i.e. the standardised data file, the separated



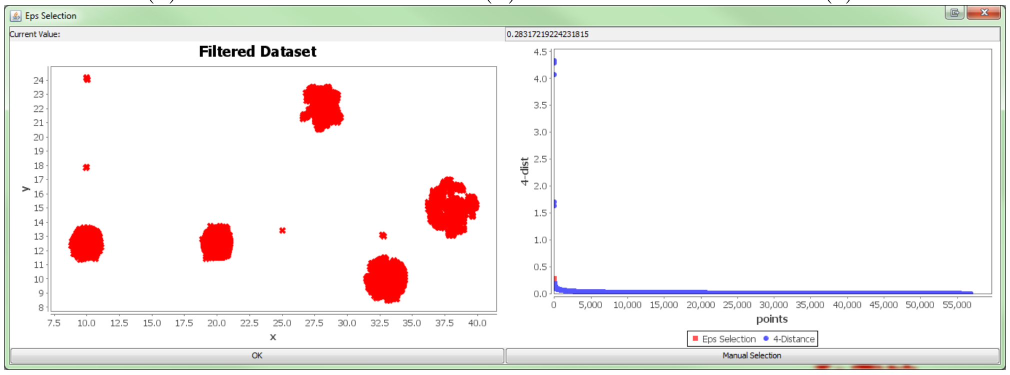
(a)



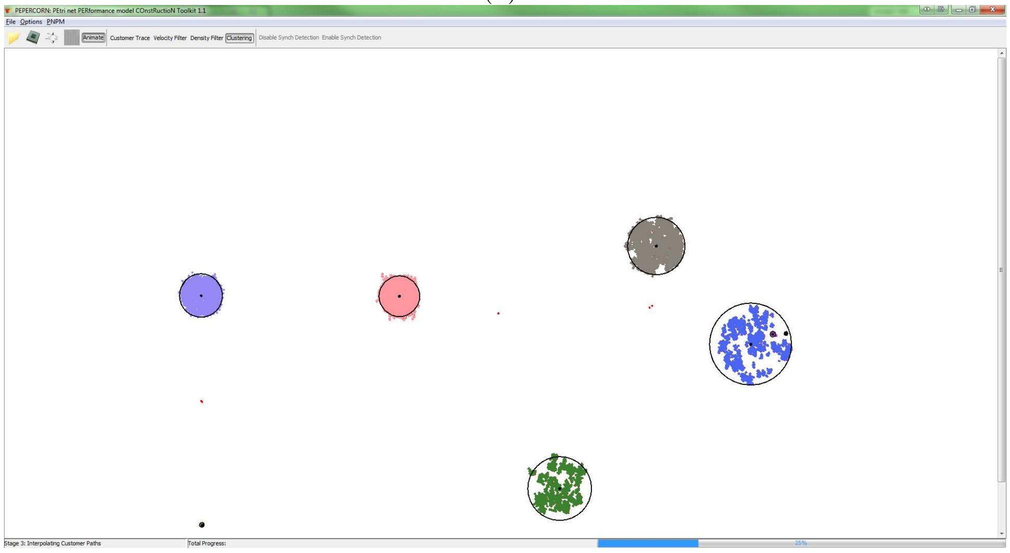
(b)



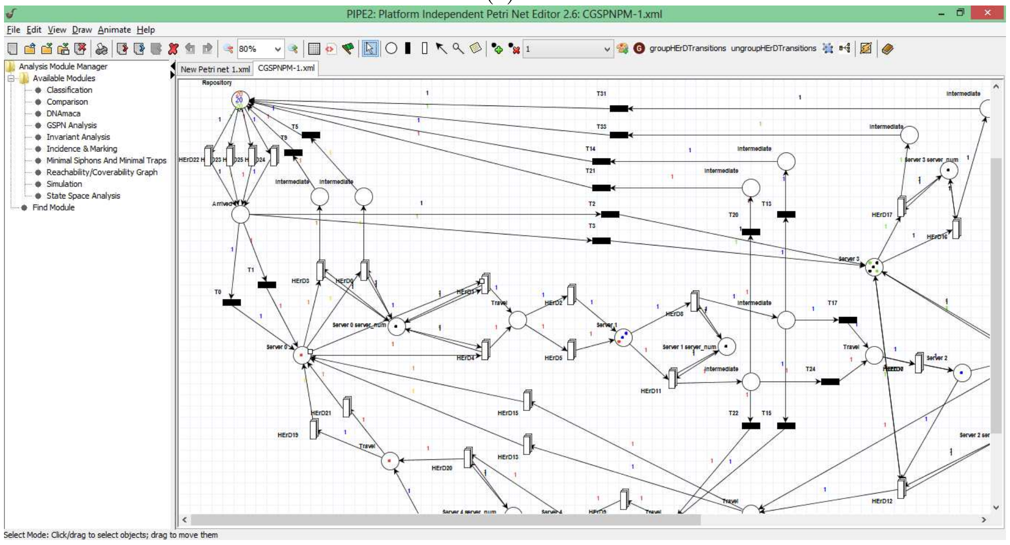
(c)



(d)

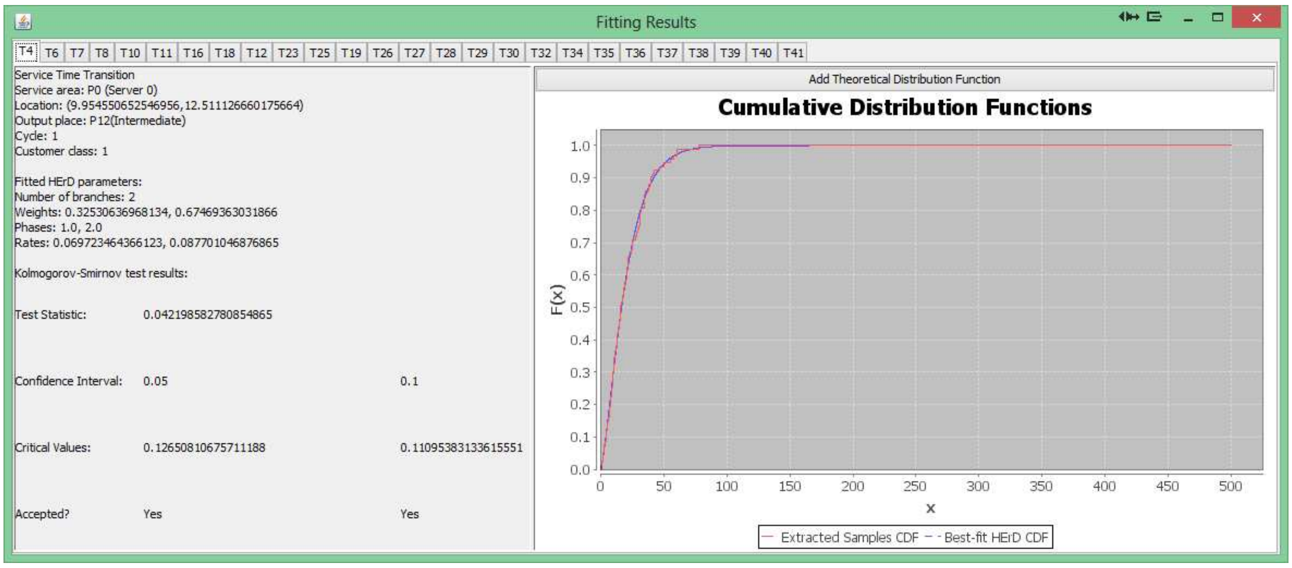

PEPERCORN: PEtri net PERformance model COnstRuctioN Toolkit 1.1
File Options PNPM
Animate Customer Trace Velocity Filter Density Filter Clustering Disable Synch Detection Enable Synch Detection
Stage 3: Interpolating Customer Paths
Total Progress:

(e)


PIPE2: Platform Independent Petri Net Editor 2.6: CGSPNPM-1.xml
File Edit View Draw Animate Help
Analysis Module Manager
Available Modules
Classification
Comparison
DNAmaca
GSPN Analysis
Invariant Analysis
Incidence & Marking
Minimal Siphons And Minimal Traps
Reachability/Coverability Graph
Simulation
State Space Analysis
Find Module
New Petri net 1.xml
CGSPNPM-1.xml
Select Mode: Click/drag to select objects; drag to move them

(f)

Figure 7.5: Figures 7.5(a), 7.5(b) and 7.5(c) show the animation of a customer's path and, the speed and density filters applied on that path respectively. Figures 7.5(d) and 7.5(e) show the *Eps* selection dialog and the clusters produced by the DBSCAN algorithm. Figure 7.5(f) shows the constructed PNPM as visualised in PIPE2 (in compact transition form).



```
Service Time Transition
Service area: P0 (Server 0)
Location: (9.954550652546956,12.511126660175664)
Output place: P12(Intermediate)
Cycle: 1
Customer class: 1

Fitted HErD parameters:
Number of branches: 2
Weights: 0.32530636968134, 0.67469363031866
Phases: 1.0, 2.0
Rates: 0.069723464366123, 0.087701046876865

Kolmogorov-Smirnov test results:

Test Statistic:        0.042198582780854865

Confidence Interval:   0.05                      0.1

Critical Values:       0.12650810675711188       0.11095383133615551

Accepted?              Yes                       Yes
```

Figure 7.6: The results window of PEPERCORN.

customer paths and the files which contain the extracted service and travelling time samples. This action should be performed when processing terminates, after the PNPM has been exported, and before processing another set of location tracking traces. Finally, the 'Export Clustering' menu, located within the 'File' menu, allows the user to export the clustering result as an image file.

7.2 Software Architecture

PEPERCORN was designed to be a stand-alone tool, initially implementing the original data processing pipeline presented in Chapter 4. However, new features were subsequently introduced to the methodology in order to facilitate the modelling of more complex customer-processing systems, i.e. presence-based synchronisation detection (cf. Chapter 5), support multiple customer classes and service cycles (cf. Chapter 6). For this reason, an evolutionary approach [100] was employed in PEPERCORN's development.

7.2.1 Resources

For the purpose of developing PEPERCORN, the following external resources were used:

1. The Java Spatial Index (JSI) RTree library [66]: this library is used to store spatial data efficiently and answer spatial queries. In PEPERCORN, this library is used during the second stage of the processing pipeline by the DBSCAN clustering algorithm; it stores the 2D points of the aggregated filtered dataset. This library was modified to support two more queries: retrieve the points that lie in the *Eps*-Neighbourhood of a point p and retrieve the Nth nearest neighbour of p (required to compute the *4-dist* graph).

2. G-FIT [105]: this tool is used to fit a HErD to a set of samples, using the EM algorithm. PEPERCORN uses G-FIT to fit several HErDs to each set of extracted service and travelling time samples. Furthermore, in order to select the best-fit HErD using the AIC, we have slightly modified G-FIT so that it returns the log-likelihood value for each fit.

3. JFreeChart Java library [63]: this library facilitates the creation of various types of charts which can be displayed as, including other formats, `swing`[1] components. In PEPERCORN, JFreeChart is used to produce and display the filtered dataset and *4-dist* graphs (see Figure 7.5(d)), as well as the service and travelling time cumulative density functions shown in the results window (see Figure 7.6).

7.2.2 Outline

The current version of PEPERCORN contains a total of sixty-five classes divided into six main packages (see Figure 7.7). The **filtering** package contains three classes which implement the three filters described in Sections 4.1, 4.2.1 and 4.2.2. Package **fitting** contains four classes that are used to obtain and evaluate[2] the best-fit HErD for a given set of service or travelling

[1] `swing` is a core Java package which contains important classes for adding a GUI to an application.

[2] This automated evaluation performs a Kolmogorov-Smirnov test in order to test the hypothesis that the extracted time samples match the best-fit HErD. The relative entropy values between the best-fit HEr and theoretical distributions (presented in the previous chapters) were computed manually using Wolfram Mathematica 8.

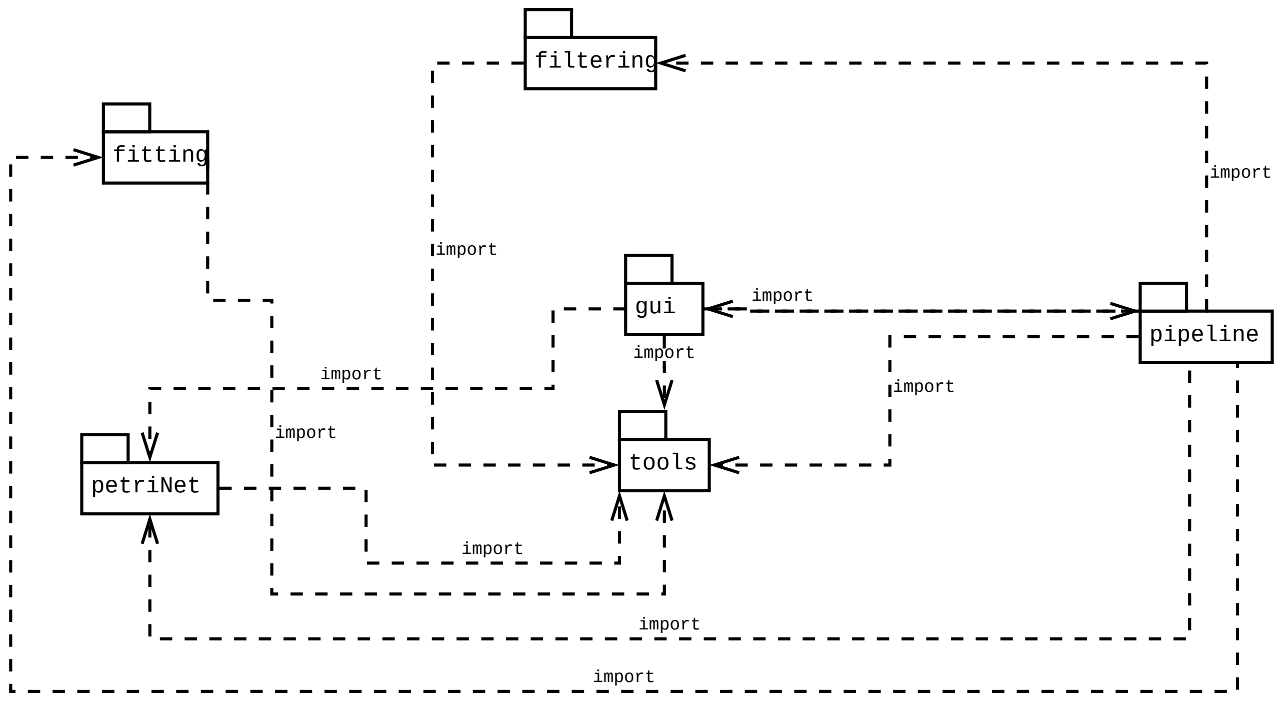


Figure 7.7: UML package diagram for the main packages of PEPERCORN.

time samples. **gui** contains a set of classes which construct PEPERCORN's user interface and handle user interaction.

Package **petriNet** consists of two subpackages: **netComponents** and **modelData**. Classes which represent the elements of the PNPM such as arcs, places, transitions, etc., and the net itself, are located within **netComponents**. The UML class diagram which depicts the relations between the basic classes of this package is shown in Figure 7.8. The **netComponents** package contains four additional classes which are not included in Figure 7.8: `ServerPlace`, `TravelPlace`, `HErDSubnet` and `PetriNet`. `ServerPlace` and `TravelPlace` are specialisations of the `Place` class, and are used to represent *service areas* and *customer movements* (between pairs of service areas) respectively. An instance of `ServerPlace` stores and process the data associated with the service area which represents. Similarly, an instance of `TravelPlace` stores and process the data associated with the movement of customers between a pair of two service areas. Auxiliary classes which are used to hold data associated with customers and the model, e.g. the number of customer classes, the inferred synchronisation conditions and the colour associated with each token type that is employed in the PNPM, are contained in **modelData**.

The **pipeline** package contains the classes which implement the stages of the integrated data processing pipeline, one class for each stage. **tools** includes various utility classes, as well as the **dbscan** package whose classes implement the DBSCAN clustering algorithm. The utility classes provide methods to perform low-level tasks, such as customer path separation, data standardisation, or define data structures which are used by other classes. We note that **dbscan** includes the **rtree** package which contains the Java Spatial Index (JSI) RTree library [66].

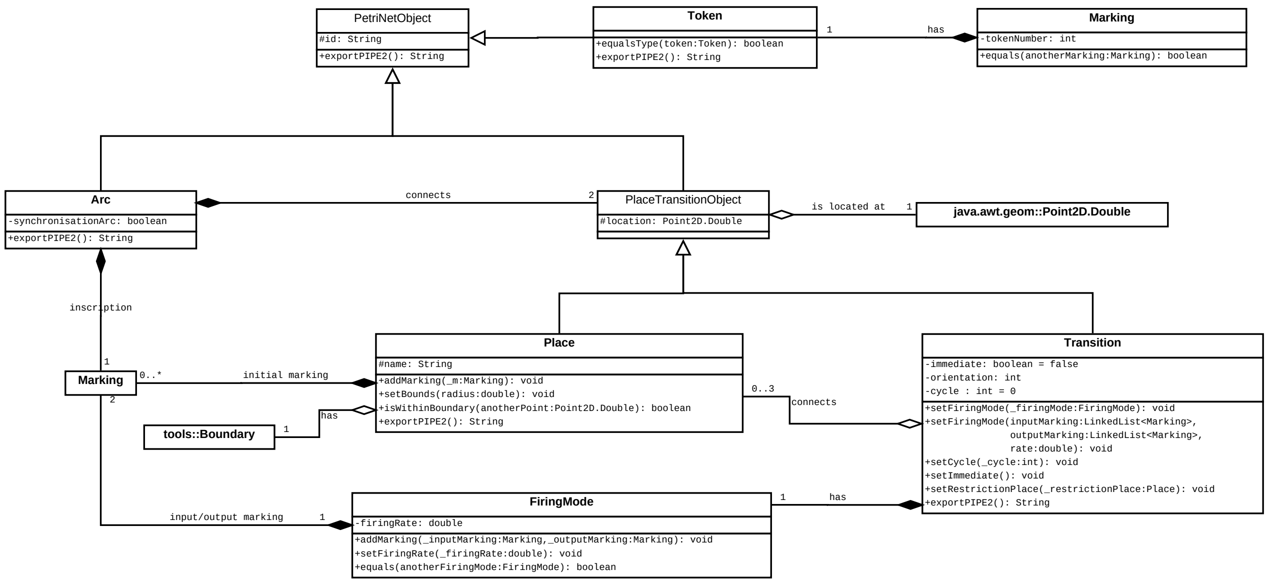

PetriNetObject
#id: String
+exportPIPE2(): String
Token
+equalsType(token:Token): boolean
+exportPIPE2(): String
1
has
Marking
-tokenNumber: int
+equals(anotherMarking:Marking): boolean
Arc
-synchronisationArc: boolean
+exportPIPE2(): String
connects
2
PlaceTransitionObject
#location: Point2D.Double
is located at
1
java.awt.geom::Point2D.Double
inscription
1
Marking
2
0..*
initial marking
Place
#name: String
+addMarking(_m:Marking): void
+setBounds(radius:double): void
+isWithinBoundary(anotherPoint:Point2D.Double): boolean
+exportPIPE2(): String
has
1
tools::Boundary
0..3
connects
Transition
-immediate: boolean = false
-orientation: int
-cycle : int = 0
+setFiringMode(_firingMode:FiringMode): void
+setFiringMode(inputMarking:LinkedList<Marking>, outputMarking:LinkedList<Marking>, rate:double): void
+setCycle(_cycle:int): void
+setImmediate(): void
+setRestrictionPlace(_restrictionPlace:Place): void
+exportPIPE2(): String
1
has
FiringMode
-firingRate: double
+addMarking(_inputMarking:Marking,_outputMarking:Marking): void
+setFiringRate(_firingRate:double): void
+equals(anotherFiringMode:FiringMode): boolean
input/output marking
1

Figure 7.8: UML class diagram for the basic classes in **netComponents**.

7.2.3 Data Mining Process

One of the main challenges we encountered while developing PEPERCORN was to extract and aggregate the data from each customer path so that it could be processed to yield the PNPM. In particular, apart from the location and service radius of each service area in the system, the following data is required in order to construct and parameterise the PNPM:

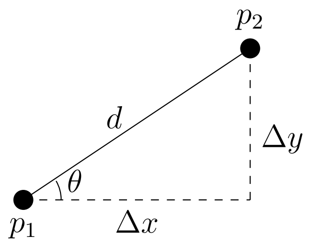
1. the timestamps required to compute the sojourn (response) time sample(s) of each customer at each service area,
2. the timestamps required to compute the travelling time sample(s) of each customer between pairs of service areas and from the system's entry point until the first service area is reached,

3. the number of customer classes supported by the system and the total number of customers of each class that were processed by the system,
4. the number of customers of each class who arrived at a service area after their entry in the system – required to approximate the initial routing probability of the customer flow,
5. the destination(s) of each customer after completing its service at a particular service area – required to approximate the inter-routing probabilities of the customer flow, and
6. the last two recorded locations of each customer processed by the system. These are used to determine if a customer has reached the system's exit point.

Most of this data is associated with the processing of customers at the system's service areas. For this reason, we implemented the `CustomerData` class. An instance of this class is used to represent each customer processed at each service area and it also stores data related to the customer's service experience at the service area. This data includes the customer's tag id, its class, the timestamps required to compute the customer's service time and the ordered list[3] of destinations after the customer's departure from the service area. `CustomerData` instances are stored in `ServerPlace`s (in particular, they are stored in the `ServerPlace` instance that corresponds to the service area the customers were processed by) and accessed via a hash map which uses the customer's tag id as key.

The `CustomerData` instances of each `ServerPlace` instance are populated during the data mining process which is performed by the `mine` method of the `DataMiner` class. However, before `mine()` is executed, each customer path must be interpolated so that the time interval Δt between consecutive location updates is no greater than 0.25 seconds. This action is taken in order to account for the "lost" readings that were filtered out during the first stage of the data processing pipeline and thus, obtain better estimates for the response and travelling time samples. To demonstrate how an interpolated location update is generated consider the following diagram where p_1 and p_2 denote the customer's position at time t_1 and t_2 respectively ($t_2 > t_1 + 0.25$), i.e. the position of the customer as indicated from location updates l_1 and l_2.

[3]Multiple entries in this list denote multiple visits by the customer to the service area, i.e. existence of cyclic services. In this scenario, the ith entry of the list, holds the id of the destination that the customer was observed at, after its ith visit to the service area.



The Cartesian coordinates p'_x, p'_y of the kth interpolated point p'_k are given by

$$\begin{aligned} p'_x &= p^1_x + \frac{d}{\Delta t} t \cos\theta \\ p'_y &= p^1_y + \frac{d}{\Delta t} t \sin\theta \end{aligned}$$

where p^1_x and p^1_y denote the x and y coordinate of p_1, $\Delta t = t_2 - t_1$ and t is the kth cumulative time increment, i.e. $t = 0.25 \cdot k$, $k \in \{i \,|\, i \in \mathbb{N}, 0.25 \cdot i < \Delta t\}$. The kth interpolated location update is then `(tagName,` p'_x`,` p'_y`,` t`, stderr, type)` where `stderr` and `type` are equal to the values of the corresponding fields in l_2. The latter process is performed by the `PathInterpolation` class contained within **tools**.

The key idea behind the implementation of `mine()` is that the state of a customer during its stay in the system can be characterised by three events: system entry, arrival at a service area, departure from a service area. Thus, a customer path[4] can be decomposed into a sequence of states, say A, B and C, where state A initiates upon the customer's entry in the system, state B initiates upon the customer's arrival at a service area and state C initiates upon the customer's departure from a service area. This is shown graphically in Figure 7.9.

Transitions between states are identified by examining the values of two `Place` instances, `last` and `next`. `last` holds a reference to the last service area where the customer was observed to be present and `next` to the current one. These objects are initialised as `null`. Each state, as well as a transition between two states, is identified by considering the values of these two objects at each location update:

[4]Recall that a customer path is a collection of all (time-ordered) location updates associated with a particular customer.

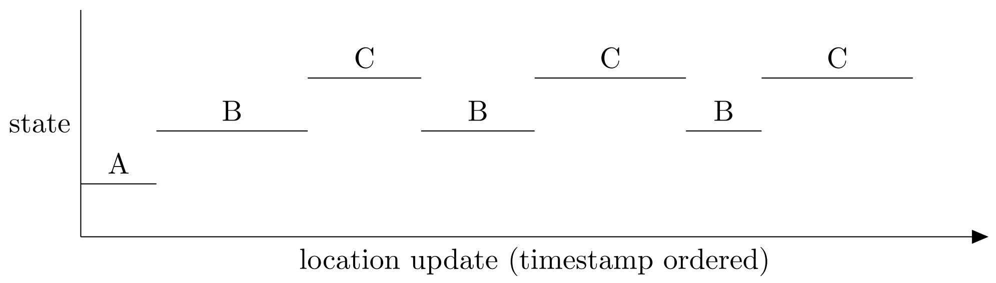


Figure 7.9: The states of a typical customer path. State A means that the customer has entered the system and that is moving towards the first service area. State B denotes that the customer is inside a service area waiting or being serviced, and state C that the customer is moving between service areas or that is headed towards the system's exit point.

- If `last == null` and `next == null`, then the customer has entered the system and is moving towards the first service area, i.e. the customer is in state A.
- If `last == null` and `next != null`, then the customer has reached the first service area, i.e. transition A→B occurs.
- If `last != null` and `next == null`, then the customer has completed its service at the service area which corresponds to `last` and is moving towards the next service area or towards the system's exit point, i.e. transition B→C occurs.
- If `last != null` and `next != null`, then the customer is either in state B, i.e. waiting or being serviced at the service area that corresponds to `last`, or has left the service area that corresponds to `last` and has reached some other service area, i.e. transition C→B. We distinguish between these two cases by comparing the ids of `last` and `next`.

Whenever a value not equal to `null` is observed for `next`, we set `last` equal to `next` and `next` equal to `null` so a transition to another state can be identified.

When A→B occurs the method increases the number of initial customers (of the appropriate customer class) of the service area which corresponds to the place (`ServerPlace`) `next`. This information is used later to compute the initial routing probabilities of the customer flow. The timestamps required to compute the entry time of the customer at `next` are also extracted here. In particular, the timestamp of the last location update in state A corresponds to the last disappearance timestamp and the timestamp of the first update in state B corresponds to the first disappearance timestamp (cf. Figure 4.11). These location updates are passed as arguments to the `addLastDisappearanceTime` and `addFirstAppearanceTime` methods of `next`.

When C→B occurs we store the id of `next` in a list (one for each customer) within the place (`ServerPlace`) `last`. These lists are processed to yield the routing probabilities of customers processed by the service area that corresponds to `last`. The same procedure as above, i.e. transition A→B, is employed to store the timestamps that are required to compute the entry time of the customer at the service area that corresponds to `next`.

The last appearance and first disappearance timestamps (required to compute the exit time – cf. Figure 4.11) of the customer at a service area are identified when the transition B→C is observed. The methods `addLastAppearanceTime` and `addFirstDisappearanceTime` of `last` are now invoked with arguments the last location update in state B and the first location update in state C respectively.

Instances of `TravelPlace` are dynamically created during the runtime of this method as follows: when a customer transits from a service area to another, i.e. transition C→B occurs, a new `TravelPlace` object associated with the customer movement between the service areas that correspond to `last` and `next` is created. That is, only if no other instance of `TravelPlace` associated with the two service areas already exists; otherwise the existing instance is retrieved. Within that object we store the timestamps required to compute the travelling time samples of customers (via the `addTimestamp` method): the last appearance and first disappearance times of the customer at the service area which corresponds to `last` and, the last disappearance and first appearance times of the same customer at the service corresponding to `next`. For the computation of travelling time samples from the system's entry point until the first service area, we consider three timestamps: the system entry time, i.e. the timestamp of the first reading of the customer's path[5], and, the last disappearance and first appearance times of the customer at the service area which corresponds to `next`.

[5]This assumes that customers are tagged as soon as they enter the system.

7.3 Conclusion

This chapter has presented an overview of the capabilities and software architecture of the PEPERCORN tool. This is a Java-based implementation of the automated Petri Net performance model inference pipeline that was presented in Chapters 4, 5 and 6.

PEPERCORN has been evaluated through six case studies. These case studies were conducted using synthetic location tracking data and assessed PEPERCORN under several types of customer-processing systems, including systems with synchronisation, multiple customer classes and service cycles. The obtained results suggest that PEPERCORN is capable of inferring the abstract structure, stochastic features and high-level customer flow of complex systems (subject to the modelling assumptions presented in Section 1.2), at least when synthetic location tracking data is used.

The synthetic location traces are generated using an extended version of the location-aware Queueing Network simulator LocTrackJINQS which was presented in Chapter 3.

Chapter 8

Conclusion

8.1 Summary of Achievements

This thesis has explored the automated extraction and construction of Petri Net performance models from high-precision location tracking data. The availability of such automated techniques can increase the applicability domain of performance modelling since the modelling process will be faster, more affordable – compared with traditional techniques – and will provide more accurate results. Previous work in this area focused on developing application-specific automated extraction and/or construction techniques, e.g. [114, 71], whose application is limited only to a narrow spectrum of customer-processing systems.

The key contribution of this thesis is the methodology presented in Chapter 4 and its extensions presented in Chapters 5 and 6. The incrementally developed methodology provides a unified framework under which performance models can be inferred from the location tracking traces of the system's customer flow. Its coarse-grained approach is based on a four-stage data processing pipeline and aims to provide high-level information regarding the customer and resource flow of the underlying system. The system's performance can be examined through the computation of end-to-end response time distributions and quantiles for the constructed model (cf. Section 4.5.2). The processing pipeline can be applied to general-type customer-processing systems given that they satisfy certain assumptions (cf. Section 1.2).

The first stage of the pipeline prepares the raw location tracking data for processing by the subsequent stages. Initially, the input data is converted into a standardised format and separated into customer paths. A speed-based filter is then applied on each customer path in order to remove possibly erroneous location updates. The second stage infers the locations and radii of service areas in the system. It consists of a three-layer technique and operates under the assumption that customers stop or slow down while receiving service. The three layers are: speed filtering, density filtering and the application of the DBSCAN clustering algorithm [41] on the aggregated filtered data. The centroids of the produced clusters are used to approximate the locations of the system's service areas. The radius of a service area is conservatively approximated as the 110% of the 95th percentile of the distance between the corresponding cluster's centroid and each of its contained points. Stage three constructs the basic structure of the derived PNPM, beginning with places and transitions required to represent the flow of customers in the system. During this stage timed transitions are not parameterised; they are replaced in the next stage by GSPN subnets that accurately reflect the distributions of the relevant time delays. In preparation for this, the response time samples – for each customer processed by a service area – are computed and broken down into waiting and service time. Samples of the time required by each customer to transit between two service areas are also computed (travelling time samples). In the fourth stage of the processing pipeline the timed transitions are replaced by GSPN subnets that reflect the distributions of the corresponding service and travelling time samples computed during stage three. For each set of these time samples we fit several hyper-Erlang distributions (HErDs) using the G-FIT tool [106]. Each GSPN subnet is constructed in a way it reflects the best-fit HErD selected using the Akaike Information Criterion (AIC) [3].

To address the lack of support for synchronisation, we have developed a mechanism which can automatically detect presence-based synchronisation between two or more service areas and calculate the corresponding synchronisation conditions. This mechanism employs two algorithms which gather evidence that indicates whether the processing of customers at a service area depends on the presence of customers at other service areas. This evidence consists of two components: the first is the maximum number of customers which were present at other service areas during the customer's service time interval; the second is the number of customers present at other service areas upon the customer's service termination. A third algorithm aggregates this evidence for all customers processed at each service area and assesses whether the evidence suffices to infer synchronisation. When synchronisation is inferred, the synchronisation conditions are calculated. The presence-based synchronisation detection mechanism and its incorporation into the existing methodology is presented in Chapter 5 and is evaluated through three case studies (cf. Chapters 5 and 6).

To provide support for multiple customer classes, we introduce CGSPNs into the methodology. Each class of customers in the underlying system is modelled by a distinct token type (colour). This is achieved by extrapolating the earlier methodology, and in particular stages three and four of the data processing pipeline, to each customer class. In stage three, distinct transitions – each with a different firing mode – are created in order to model the movements of customers of different classes. Furthermore, the extracted service time samples for each service area and the travelling time samples for each pair of service areas are now separated into subsets, one subset for each customer class that was processed at the particular service area. During the fourth stage, several HErDs are fitted to each of the latter subsets of time samples and for each subset the best-fit HErD is selected as before; these HErDs are now modelled by CGSPN subnets. CGSPNs are also employed to enable the accurate representation of the flow of customers in systems which contain service cycles. To achieve this, we examine the system for the presence cyclic services using Johnson's algorithm [64]. This algorithm is applied to a directed graph – obtained by mapping the places of the non-parameterised PNPM to vertices and similarly, its transitions, along with their incident arcs, to directed edges – and it enumerates all elementary cycles that are contained in that graph. Each detected cycle in the directed graph corresponds to a service cycle in the underlying system. The service areas that make up each service cycle are identified by reversing the mapping that was used to obtain the directed graph. Customers are then accurately routed through the service cycle by changing the token colour that corresponds to their class upon their second visit to the first service area of the service cycle, i.e. the head of the service cycle. This modelling approach is applied to each service cycle and it allows the distinction between customers that have completed the service cycle and those that are about to enter it. Another extension enables the calculation and representation of the inter-routing probability of the customer flow between the system's service areas. These extensions are presented and evaluated (via three case studies) in Chapter 6.

The design and execution of physical experiments to support and validate location-based research is costly, both in terms of time and resources. Furthermore, the success of such experiments depends – in addition to precise design – on the behaviour of the participating individuals and on the degree of accuracy that the RTLS is calibrated. Thus, inaccurate and/or non-representative data is likely to be obtained which may encumber the validation process. To this end, we have developed a tool which can build and simulate Queueing Networks, namely LocTrackJINQS [56]. This tool facilitates a well defined and controlled environment for running location-based experiments and is able to generate location tracking data similar to those collected from actual RTLSs. LocTrackJINQS is based on the extensible library JINQS [43] which allows the high-level specification and event-driven simulation of multiclass Queueing Networks. LocTrackJINQS preserves the high-level features provided by JINQS and in addition, allows the specification of low-level spatial information such as location of servers, service radii and user-defined customer paths. Finally, its inherited extensibility from JINQS permits users to specify specialised entities, service and queueing policies, making it suitable for simulating a variety of physical customer-processing systems.

The work presented in this thesis has demonstrated that the automated construction of performance models is a viable and feasible option to be considered in performance modelling. In particular, the results obtained from the case studies suggest that the developed methodology has the ability to construct accurate Petri Net performance models of physical customer-processing systems (which satisfy the set of modelling assumptions stated in Section 1.2) by analysing the location tracking traces of the system's customer flow, at least when synthetic location tracking data is used. Naturally, there is a dependency between the collected location tracking data and the inferred model. This implies that the inferred model will accurately reflect the underlying system's structure and behaviour during the period that the system was monitored. For example, if the system is monitored during a period of heavier customer traffic load, additional synchronisation conditions, bottlenecks and/or service areas may be discovered compared to those detected from location tracking data collected (from the same system) during periods of lower workloads. Our work is supplemented with two tools: PEPERCORN and LocTrackJINQS. The former is a Java-based implementation of the developed methodology and the latter, also implemented in Java, can be used as a stand-alone tool or as a library in order to formulate and simulate physical customer-processing systems as Queueing Networks.

8.2 Applications

The generic nature of the research presented allows the developed data processing pipeline to be applied to several types of physical systems which process customers or goods and which satisfy the modelling assumptions stated in Section 1.2. Evidently, our methodology presupposes the availability of high-precision location tracking data describing the (low-level) flow of customers in the system to be modelled. RTLSs, especially UWB-based ones, can provide accurate and continuous monitoring of indoor environments and are mainly employed in healthcare, manufacturing and supply chain management. This currently restricts the immediate domain of applicability of our methodology to such applications. However, if the service areas in a system are relatively large, and thus interactions between customers and resources take place over large areas, then less accurate tracking can be used, e.g. GPS. Furthermore, state of the art GPS systems, namely GPS III, are to be deployed in the near future and can achieve an accuracy within one metre [38]. The use of data provided by such widely adopted tracking technologies can lead to further applications.

This methodology is particularly suitable for assessing the performance of small-scale indoor customer-processing systems characterised by complex processes which are difficult to capture via manually collected data. For example, the developed data processing pipeline may be easily applied to A&E departments in order to locate system bottlenecks and assess the QoS received by patients; if QoS is not acceptable, the obtained PNPM can also be used as a virtual laboratory whose purpose is to examine whether the addition of resources is a viable solution to this issue. Other types of customer-processing systems which fall into the applicability domain of our methodology include banks, post offices and airports. However, the widespread adoption of RTLSs by such organisations is currently limited since it requires a heavy upfront investment. Nevertheless, it is clear that such data will become increasingly available as the ever-progressing wireless technology enables the realisation of the Internet of Things [5], whereby numerous everyday objects – especially mobile phones equipped with location tracking sensors [69] – are networked. Currently, the scalability of our methodology mainly depends on three components: the DBSCAN clustering algorithm, the presence-based synchronisation detection mechanism and the G-FIT tool. The worst-case computational complexity of these components is $O(n \cdot\ logn)$, where n is the total number of points to be clustered, $O(N^2 \cdot n^3)$, where N is the number of service areas in the system and n is the number of customers processed by each service area[1], and $O(M \cdot K)$ per iteration of the EM algorithm, where M is the number of Erlang branches and K is the sample size. Therefore, the main bottleneck that (currently) prohibits the application of our methodology to large-scale systems is the presence-based synchronisation detection mechanism.

Real time modelling of physical customer-processing systems is another possible application, given that the model is not constructed from a "blank" state. This would require the availability of a considerable amount of historical data. For example, if a model for the underlying system has been already inferred from historical data, then a stream of new location tracking data can be used to fine-tune the model's parameters, e.g. initial and inter-routing probabilities of the customer flow, service and travelling time distributions. Another possible real time application of our methodology, assuming that the model of a system has already been inferred, is system security. That is, the real time movements of customers can be compared against the expected behaviour specified by the inferred model. If the movement of a customer is not compliant with the general movement of its class, then the behaviour of that customer can be flagged as irregular. For example, if we consider a hospital, this application can be used to generate alerts if customers classified as visitors are observed in surgery rooms.

[1]Assuming that all service areas in the system process the same number of customers.

8.3 Future Work

There are numerous possible extensions to the work presented in this thesis. The current methodology assumes a unique tag identifier for each customer processed by the system, i.e. a one-to-one mapping between tags and customers. Sometimes in real systems, such as hospitals, tags are assigned to customers, when they enter the system, and are recycled upon the customers' exit. By detecting the departure of customers from the system, we can flag the initiation of a new customer path when an already existing tag identifier re-appears. This will enable us to discard the latter assumption and increase the scalability of our methodology. Furthermore, in the scenario of multiple customer classes, we currently assume that the class of each customer remains constant throughout the customer's presence in the system. In the future, we wish to support the dynamic allocation of customer classes. That is, customers in the system can be assigned to a different class after the termination of their service at a service area. This feature will enable the modelling of more realistic customer-processing systems since in reality, it is often the case that customers are classified after their arrival in a system, e.g. hospitals, and, also, they may be assigned to a different class later on, as they pass through the various processing stages, e.g. factory assembly lines. This feature can be supported by examining the class of a customer before and after service completion at a service area. If the class of a customer is changed, we can use service time transitions which "modify" the colour of the token that is associated with that particular customer class, one for each observed combination of customer class change.

The structure of the constructed Petri Net models and, in particular, that of the subnets used to simulate the service delays of the system (cf. Figures 4.14 and 6.4) provides support for multiple servers. To be precise, the number of tokens contained in the complementary place of these subnets – which controls the number of tokens (customers) that can be in the subnet simultaneously – can be adjusted accordingly so that multiple servers are modelled[2]. Currently, our methodology assumes single-server semantics. It would be interesting to investigate for mechanisms which would allow the inference of the number of servers that consist a particular

[2]For timed transitions inside these subnets we assume infinite-server semantics.

service area from location tracking data.

In real life systems it is often the case that certain classes of customers are serviced with higher priority. As an example, consider an A&E department where patients are categorised and treated based on the criticality of their condition. Although our methodology considers multiple customer classes, priority is not explicitly supported. Therefore, another possible extension would be the support for customer class priority. This feature can be implemented either by developing and incorporating subnets in the PNPM which will use immediate transitions to give priority to certain customer classes over others, or by employing Petri Nets with priority [77]. This subclass of Petri Nets is defined as GSPNs but introduces an additional function, which assigns to each transition an integer which corresponds to the transition's priority. The implementation of FIFO queueing discipline (which is often the case in physical customer-processing systems) is certainly another possible extension. However, this is not inherently supported in Petri Nets since tokens are indistinguishable and therefore, subnets which model the selection algorithm need to be developed. Another possible way to tackle this issue would be to employ Queueing Petri Nets [13].

As we discussed earlier (cf. Chapters 4, 5 and Section 8.1), the constructed PNPM is highly dependent on the set of location tracking data collected during the monitoring period of the underlying customer-processing system. It would be very interesting to investigate for ways to support model refinement based on several iterations of the PNPM for different sets of location tracking data. An initial direction to tackle this would be to characterise the system's workload that corresponds to each dataset as low, medium or heavy. One possible way this can be achieved is by computing (for each dataset) the ratio between the total number of customers processed by the system and the duration of the system's monitoring period. Then, an aggregate model may be build by considering the properties of each PNPM that are likely to be best captured under each workload type. For example, synchronisation between service areas and their corresponding synchronisation conditions are (most likely) correctly inferred during low or medium customer traffic loads. Also, assuming that the system's structure and resource allocation is static, bottlenecks, e.g. regions of high-customer traffic, can be easily and automatically detected by identifying any discrepancies between the number and locations of the inferred service areas of a low (or medium workload model) and a heavy workload model.

We would also like to characterise the robustness and precision of our methodology in the face of increasing levels of noise. Initial experiments, focused on the second case study of Chapter 4, indicate that the methodology is robust for normally distributed errors of up to mean 0.5 m and standard deviation 0.4 m. Nevertheless, we expect that after a certain level of noise, our methodology will break down. This is because the filtering techniques employed to clean the data and to infer the locations and radii of the system's service areas will likely fail due to the apparently erratic movement of customers.

Ultimately, the goal of our research is to utilise the automatically constructed Petri Net performance models in order to assess the underlying system's performance and to identify bottlenecks that may elude the manual model construction. Therefore, an important part of our future work is to compute the end-to-end response time distributions for some of the case studies that were presented in this thesis. This will also provide additional evidence that supports the accuracy of the constructed models. However, we must take into consideration the increasing model complexity (proportional to the underlying system's complexity) which may lead to a state-space explosion (cf. Section 6.4.2). A possible technique that could mitigate this problem would be adopting a Semi-Markov Stochastic Petri Net (SM-SPN) [25, 24] representation the travelling and service delays delays of each customer class at each service area; SM-SPNs allow transitions to fire with generally-distributed time and thus the use of subnets to reflect each best-fitted HErD will no longer be required. In extreme cases (where the state-space remains unmanageably large) the latter approach can be complemented with aggregation techniques for Semi-Markov processes [49].

There are several extensions which can be implemented in LocTrackJINQS so that more realistic simulations can be constructed. Extending customer paths into two dimensions, i.e. the addition of path width, is a primary objective since this could be used to restrain the, currently unrestrained, flow of customers. Furthermore, customer speed can be adjusted according to customer density in a path so that customers move more realistically. The incorporation of more complex-shaped service areas, possibly user-defined, is also another possible extension.

Finally, the exportation of the constructed Queueing Networks to other tools via the PMIF2 [98] would be a very useful feature to include. However, it is likely that location-based information will be lost as PMIF2 does not (currently) provide support for location information regarding the network's entities.

Bibliography

[1] S. Abramsky. Petri Nets, Discrete Physics, and Distributed Quantum Computation. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 527–543. 2008.

[2] R. Agrawal, D. Gunopulos, and F. Leymann. Mining Process Models from Workflow Logs. In *Proc. 6th International Conference on Extending Database Technology (EDBT 1998)*, pages 469–483, Valencia, Spain, March 1998.

[3] H. Akaike. A New Look at the Statistical Model Identification. *Automatic Control, IEEE Transactions on*, 19(6):716 – 723, December 1974.

[4] F.A. Graybill A.M. Mood and D.C. Boes. *Introduction to the Theory of Statistics.* McGraw-Hill, 1974.

[5] Scientific American. The Internet Of Things. `http://www.scientificamerican.com/article.cfm?id=the-internet-of-things`.

[6] N. Anastasiou, T.-C. Horng, and W. Knottenbelt. Deriving Generalised Stochastic Petri Net Performance Models from High-Precision Location Tracking Data. In *Proc. 5th International ICST Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2011)*, pages 91–100, Paris, France, May 2011.

[7] N. Anastasiou and W. Knottenbelt. Deriving Coloured Generalised Stochastic Petri Net Performance Models from High-Precision Location Tracking Data. In *Proc. 4th ACM/SPEC International Conference on Performance Engineering (ICPE 2013)*, Prague, Czech Republic, April 2013. To appear.

[8] N. Anastasiou, W. Knottenbelt, and A. Marin. Automatic Synchronisation Detection in Petri Net Performance Models Derived from Location Tracking Data. In *Proc. 8th European Conference on Computer Performance Engineering (EPEW 2011)*, pages 29–41, Borrowdale, UK, October 2011.

[9] R. Angeles. RFID Technologies: Supply-chain applications and implementation issues. *Information Systems Management*, (22):51–65, 2005.

[10] M. Ankerst, M.M. Breunig, H.-P. Kriegel, and J. Sander. OPTICS: Ordering Points To Identify the Clustering Structure. *SIGMOD Rec.*, 28:49–60, June 1999.

[11] S.W.M. Au-Yeung. Response Times in Healthcare Systems, 2008.

[12] S.W.M. Au-Yeung, U. Harder, E.J. McCoy, and W. Knottenbelt. Predicting Patient Arrivals to an Accident and Emergency Department. *Emergency Medicine Journal*, (26):241–244, 2009.

[13] F. Bause. Queueing Petri Nets-A formalism for the combined qualitative and quantitative analysis of systems. In *Proc. 5th International Workshop on Petri Nets and Performance Models*, pages 14–23, Toulouse, France, October 1993.

[14] F. Bause and P.S. Kritzinger. *Stochastic Petri Nets*. Friedrich Vieweg & Sohn Verlag, 2002.

[15] R. Becker, R. Cáceres, K. Hanson, S. Isaacman, J.M. Loh, M. Martonosi, J. Rowland, S. Urbanek, A. Varshavsky, and C. Volinsky. Human Mobility Characterization from Cellular Network Data. *Communications ACM*, 56(1):74–82, January 2013.

[16] R.A. Becker, R. Caceres, K. Hanson, J.M. Loh, S. Urbanek, A. Varshavsky, and C. Volinsky. A Tale of One City: Using Cellular Network Data for Urban Planning. *Pervasive Computing, IEEE*, 10(4):18 –26, April 2011.

[17] C. Biernacki, G. Celeux, and G. Govaert. Choosing starting values for the EM algorithm for getting the highest likelihood in multivariate Gaussian mixture models. *Computional Statistics & Data Analysis*, 41(3-4):561–575, January 2003.

[18] J. Billington, S. Christensen, K.V. Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In *Proc. 24th International Conference of Applications and Theory of Petri Nets (ICATPN 2003)*, pages 1023–1024, Eindhoven, The Netherlands, June 2003.

[19] J. Bilmes. A Gentle Tutorial of the EM Algorithm and its Application to Parameter Estimation for Gaussian Mixture and Hidden Markov Models. Technical report, 1998.

[20] P. Bonet, C.M. Llado, R. Puijaner, and W. Knottenbelt. PIPE v2.5: A Petri Net Tool for Performance Modelling. In *Proc. 23rd Latin American Conference on Informatics (CLEI 2007)*, San Jose, Costa Rica, October 2007.

[21] S. Borman. The Expectation Maximization Algorithm: A Short Tutorial. Technical report, 2004.

[22] S. Boschi, M. Di lanni, P. Crescenzi, G. Rossi, and P. Vocca. MOMOSE: a mobility model simulation environment for mobile wireless ad-hoc networks. In *Proc. 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems Workshops (SIMUTOOLS 2008)*, pages 1–10, 2008.

[23] H. Bozdogan. Model selection and Akaike's Information Criterion (AIC): The general theory and its analytical extensions. *Psychometrika*, 52:345–370, 1987. 10.1007/BF02294361.

[24] J.T. Bradley, N.J. Dingle, P.G. Harrison, and W.J. Knottenbelt. Distributed Computation of Passage Time Quantiles and Transient State Distributions in Large Semi-Markov Models. In *Proc. International Workshop on Performance Modelling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO-PDS 2003)*, Nice, France, April 2003.

[25] J.T. Bradley, N.J. Dingle, P.G. Harrison, and W.J. Knottenbelt. Performance Queries on Semi-Markov Stochastic Petri Nets with an Extended Continuous Stochastic Logic. In *Proc. 10th International Workshop on Petri Nets and Performance Models (PNPM 2003)*, pages 62–71, Urbana, Illinois, September 2003.

[26] P.S. Bradley and U. Fayyad. Refining Initial Points for K-Means Clustering. In *Proc. 15th International Conference on Machine Learning (ICML 1998)*, page 9199, Madison, Wisconsin, USA, July 1998.

[27] F. Brosig, N. Huber, and S. Kounev. Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems. In *Proc. 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 183–192, Lawrence, Kansas, USA, November 2011.

[28] F. Calabrese, F.C. Pereira, G. Di Lorenzo, L. Liu, and C. Ratti. The geography of taste: analyzing cell-phone mobility and social events. In *Proc. 8th International Conference on Pervasive Computing (Pervasive 2010)*, pages 22–37, Helsinki, Finland, May 2010. Springer-Verlag.

[29] Cambridge Systematics Inc., PB Consult, and System Metric Group Inc. Analytical tools for asset management. Technical Report 545, National Cooperative Highway Research Program, 2005.

[30] B. Carré. *Graphs and networks.* Oxford applied mathematics and computing science series. Clarendon Press, 1979.

[31] A. Charalambous. Extension of PIPE2 to Support Coloured Generalised Stochastic Petri Nets. Final Year Project, Imperial College London, 2010.

[32] P. Chen, S.C. Bruell, and G. Balbo. Alternative Methods for Incorporating Non-exponential Distributions into Stochastic Timed Petri Nets. In *Proc. 3rd International Workshop on Petri Nets and Performance Models (PNPM 1989)*, pages 187–197, Kyoto, Japan, December 1989.

[33] G. Ciardo. Petri nets with marking-dependent arc cardinality. In *Properties and Analysis, Advances in Petri Nets, LNCS*, pages 179–198. Springer-Verlag, 1994.

[34] G. Ciardo, J. Muppala, and K. Trivedi. SPNP: Stochastic Petri Net Package. In *Proc. 3rd International Workshop on Petri Nets and Performance Models (PNPM 1989)*, pages 142–151, Kyoto, Japan, December 1989.

[35] ComputerWeekly.com. British Airways reveals what went wrong with Terminal 5. `http://www.computerweekly.com/news/2240086013/British-Airways-reveals-what-went-wrong-with-Terminal-5`.

[36] V. Cortellessa and R. Mirandola. Deriving a Queueing Network based Performance Model from UML Diagrams. In *Proc. 2nd International Workshop on Software and Performance WOSP 2000*, pages 58–70, Ottawa, Canada, September 2000.

[37] A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[38] digitaltrends.com. GPS III explained: Everything you need to know about the next generation of GPS. `http://www.digitaltrends.com/mobile/gps-iii-explained-everything-you-need-to-know-about-the-next-generation-of-gps/`.

[39] N.J. Dingle, P.G. Harrison, and W.J. Knottenbelt. Response Time Densities in Generalised Stochastic Petri Net Models. In *Proc. 3rd International Workshop on Software and Performance (WOSP 2002)*, pages 46–54, Rome, Italy, July 2002.

[40] N.J. Dingle, W.J. Knottenbelt, and T. Suto. PIPE2: A Tool for the Performance Evaluation of Generalised Stochastic Petri Nets. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):34–39, March 2009.

[41] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proc. 2nd International Conference in Knowledge Discovery and Data Mining (KDD 1996)*, pages 226–231, Portland, Oregon, USA, August 1996.

[42] Y. Fang. Hyper-Erlang Distribution Model and its Application in Wireless Mobile Networks. *Wireless Networks*, 7:211–219, 2001.

[43] Tony Field. JINQS: An Extensible Library for Simulating Multiclass Queueing Networks V1.0. User Guide, 2006.

[44] J. Gambon. Ultra-Wideband RFID Tracks Nuclear Power Plant Workers. RFID journal. `http://www.ubisense.net/en/resources/case-studies/ultra-wideband-rfid-tracks-nuclear-power-plant-workers.html`.

[45] H. Gonzalez, J. Han, and X. Li. FlowCube: Constructing RFID FlowCubes for Multi-Dimensional Analysis of Commodity Flows. In *Proc. 32nd International Conference on Very Large Databases (VLDB 2006)*, pages 834–845, Seoul, Korea, September 2006.

[46] H. Gonzalez, J. Han, and X. Li. Mining Compressed Commodity Workflows From Massive RFID Data Sets. In *Proc. 15th ACM International Conference on Information and Knowledge Management (CIKM 2006)*, pages 162–171, Arlington, Virginia, USA, November 2006. ACM.

[47] M.C. González, C.A. Hidalgo, and A.-L. Barbási. Understanding individual human mobility patterns. *Nature*, 453(5):779–782, June 2008.

[48] The Guardian. Number of A&E patients waiting more than four hours is highest since 2004. `http://www.guardian.co.uk/society/2012/may/31/patients-waiting-four-hours-2004`.

[49] M.C. Guenther, N.J. Dingle, J.T. Bradley, and W.J. Knottenbelt. Passage-time Computation and Aggregation Strategies for Large Semi-Markov Processes. *Performance Evaluation*, 68(3):221–236, 2011.

[50] G. Hamerly and C. Elkan. Learning the k in k-means. Technical report, University of California, San Diego, 2002.

[51] T.R. Hansen, J.E. Bardram, and M. Soegaard. Moving Out of the Lab: Deploying Pervasive Technologies in a Hospital. *IEEE Pervasive Computing*, 5(3):24 –31, 2006.

[52] P.G. Harrison and W.J. Knottenbelt. Passage Time Distributions in Large Markov Chains. In *Proc. 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2002)*, pages 77–85, Marina Del Rey, California, June 2002.

[53] P.G. Harrison and N.M. Patel. *Performance Modelling of Communication Networks and Computer Architectures (International Computer S)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1992.

[54] M. Heiner, D. Gilbert, and R. Donaldson. Petri Nets for Systems and Synthetic Biology. In *Proc. Formal Methods for the Design of Computer, Communication, and Software Systems 8th International Conference on Formal Methods for Computational Systems Biology (SFM 2008)*, pages 215–264, Bertinoro, Italy, June 2008.

[55] L. M. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Tréves. A Primer on the Petri Net Markup Language and ISO/IEC 15909-2. *Petri Net Newsletter*, 76:9–28, October 2009.

[56] T.-C. Horng, N. Anastasiou, and W. Knottenbelt. LocTrackJINQS: An Extensible Location-aware Simulation Tool for Multiclass Queueing Networks. In *Proc. 5th International Workshop on Practical Applications of Stochastic Modelling (PASM 2011)*, pages 59–71, Karlsruhe, Germany, March 2011.

[57] T.-C. Horng, N. Dingle, A. Jackson, and W. Knottenbelt. Towards the Automated Inference of Queueing Network Models from High-Precision Location Tracking Data. In *Proc. 23rd European Conference on Modelling and Simulation (ECMS 2009)*, pages 664–674, Madrid, Spain, June 2009.

[58] J. Horsky, L. Gutnik, and V.L. Patel. Technology for Emergency Care: Cognitive and Workflow Considerations. In *Proc. AMIA Annual Fall Symposium 2006*, pages 344–348, Washington, DC, USA, November 2006.

[59] A. Horváth and M. Telek. Phfit: A General Phase-Type fitting tool. In *Proc. 12th International Conference on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation (TOOLS 2002)*, pages 82–91, London, UK, April 2002.

[60] C.M. Hurvich and C.-L.Tsai. Regression and time series model selection in small samples. *Biometrika*, 76(2):297–307, 1989.

[61] A.K. Jain, M.N. Murty, and P.J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31:264–323, September 1999.

[62] K. Jensen. Coloured Petri Nets and the Invariant Method. *Theoritical Computer Science*, 14:317–336, 1981.

[63] JFreeChart. A free Java chart library. `http://www.jfree.org/jfreechart/`.

[64] D.B. Johnson. Finding All the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[65] M.A. Johnson and M.R. Taaffe. The denseness of phase distributions. *Purdue School of Industrial Engineering Research Memoranda*, (88-20), 1988.

[66] JSI Rtree library. The Java Spatial Index RTree library. `http://jsi.sourceforge.net/`.

[67] KidSpotter. A Child Tracking System. `http://kidspotter.com`.

[68] S.J. Kim, S.K. Yoo, S.K. Yoo, H.O. Kim, H.S. Bae, J.J. Park, K.J. Seo, and B.C. Chang. Smart Blood Bag Management System in a Hospital Environment. In *Proc. 11th International Conference on Personal Wireless Communications (PWC 2006)*, pages 506–517, Albacete, Spain, September 2006.

[69] N. Kiukkonen, J. Blom, O. Dousse, D. Gatica-Perez, and J. Laurila. Towards Rich Mobile Phone Datasets: Lausanne Data Collection Campaign. In *Proc. 7th International Conference on Pervasive Services (ICPS 2010)*, Berlin, Germany, July 2010.

[70] W.J. Knottenbelt. Parallel Performance Analysis of Large Markov Models, 2000.

[71] S. Kounev, K. Bender, F. Brosig, N. Huber, and R. Okamoto. Automated Simulation-Based Capacity Planning for Enterprise Data Fabrics. In *Proc. 4th International ICST Conference on Simulation Tools and Techniques (SIMUTOOLS 2011)*, pages 27–36, Barcelona, Spain, March 2011.

[72] S. Kullback and R. A. Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):pp. 79–86, 1951.

[73] L. Li and H. Yokota. Application of Petri Nets in Bone Remodeling. *Gene Regulation and Systems Biology*, pages 105–114, 2009.

[74] M. Li and N.D. Georganas. Colored Generalized Stochastic Petri Nets for Integrated System Protocol Performance Modelling. In *Proc. IEEE Global Telecommunications Conference (GLOBECOM 1988)*, volume 3, pages 1798–1802, Hollywood, Florida, USA, November 1988.

[75] MailOnline. Queue here for Britain: Borders chaos returns to heathrow as passengers face 'horrendous' delays at immigration - and the Olympics still haven't even started. `http://www.dailymail.co.uk/news/article-2167600/Queue-chaos-Heathrow-passengers-face-horrendous-delays-immigration-run-London-2012-Olympics.html`.

[76] P.L. Markt and M.H. Mayer. WITNESS simulation software: a flexible suite of simulation tools. In *Proc. 29th Winter Simulation Conference (WSC 1997)*, pages 711–717, Atlanda, USA, December 1997.

[77] M.A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* John Wiley and Sons, 1995.

[78] M.A. Marsan, G. Conte, and G. Balbo. A Class of Generalized Stochastic Petri Nets for Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, 1984.

[79] M.K. Molloy. *On the Integration of Delay and Throughput Measures in Distributed Processing Models.* PhD thesis, University of California, 1981.

[80] S. Natkin. *Les Reseaux de Petri Stochastiques et leur Application a l' Evaluation des Systemes Informatiques.* PhD thesis, CNAM, Paris, 1980.

[81] BBC News. Olympic football delay at St. James' Park 'unacceptable'. `http://www.bbc.co.uk/news/uk-england-tyne-19013655`.

[82] Manchester Evening News. Tagging system to protect new born babies at Wythenshawe Hospital. http://menmedia.co.uk/manchestereveningnews/news/health/s/1409356_tagging_system_to_protect_new_born_babies_at_wythenshawe_hospital.

[83] C.H. Ng and B.H. Soong. *Queueing Modelling Fundamentals with Applications in Communication Networks*. John Wiley and Sons, 2008.

[84] J.R. Norris. *Markov Chains*. Cambridge University Press, 1st edition, 1998.

[85] M.C. O'Connor. RTLS Combines Infrared and RFID. RFID journal. http://www.rfidjournal.com/article/view/3519.

[86] M.C. O'Connor. U.S. Army Uses UWB to Track Trainees. RFID journal. http://www.ubisense.net/en/resources/case-studies/us-army-uses-uwb-to-track-trainees.html.

[87] S. Ong. Using RFID to Enhance Supply Chain Visibility - Airbus Case Study. http://www.tegoinc.com/pdfs/news/airbuscasestudy0310.pdf.

[88] D.J. Patterson, L. Liao, D. Fox, and H. Kautz. Inferring High-Level Behavior from Low-Level Sensors. In *Proc. 5th International Conference on Ubiquitous Computing (UbiComp 2003)*, pages 73–89, Seattle, Washington, USA, October 2003.

[89] C.A. Petri. Communication with automata. Technical report, Applied Data Research Inc., January 1966.

[90] D.C. Petriu and H. Shen. Applying the UML Performance Profile: Graph Grammar-based Derivation of LQN Models from UML Specifications. In *Proc. 12th International Conference on Modelling tools and techniques for Computer and Communication System Performance Evaluation (TOOLS 2002)*, pages 159–177, London, UK, April 2002.

[91] L. Popova-Zeugmann, M. Heiner, and I. Koch. Time Petri Nets for Modelling and Analysis of Biochemical Networks. *Fundamenta Informaticae*, (67):149–162, 2005.

[92] P. Reinecke, T. Krau, and K. Wolter. Cluster-Based Fitting of Phase-Type Distributions to Empirical Data. *Computers & Mathematics with Applications*, 64(12):3840–3851, 2012.

[93] F. Robinson, A. Apon, D. Brewer, L. Dowdy, D. Hoffman, and B. Lu. Initial Starting Point Analysis for K-Means Clustering: A Case Study. In *Proc. 2006 Conference on Applied Research in Information Technology*, Conway, Arkansas, USA, 2006.

[94] C. E. Shannon. A Mathematical Theory of Communication. *Bell system technical journal*, 27, 1948.

[95] G. Sheikholeslami, S. Chatterjee, and A. Zhang. WaveCluster: A Multi-Resolution Clustering Approach for Very Large Spatial Databases. In *Proc. 24th International Conference on Very Large Databases (VLDB 1998)*, pages 428–439, New York, USA, August 1998.

[96] S.G. Shirinivas, S. Vetrivel, and N.M. Erlango. Applications of Graph Theory in Computer Science An Overview. *International Journal of Engineering Science and Technology*, 2(9):4610–4621, 2010.

[97] Simul8. `http://www.simul8.com/`.

[98] C.U. Smith, C.M. Llado, and R. Puigjaner. Performance Model Interchange Format (PMIF 2): A comprehensive approach to Queueing Network Model interoperability. *Performance Evaluation*, 67(7):548 – 568, 2010.

[99] C.U. Smith and L.G. Williams. A performance model interchange format. *Journal of Systems and Software*, 581:67–85, 1994.

[100] I. Sommerville. *Software Engineering 8*. Pearson, 2007.

[101] C. Swedberg. SmartRoom Brings Information to Patients, Caregivers. RFID journal. `http://www.rfidjournal.com/article/view/8786`.

[102] P.G. Harrison S.W.M. Au-Yeung and W. Knottenbelt. A Queueing Network Model of Patient Flow in an Accident and Emergency Department. In *Proc. 20th Annual European Simulation and Modelling Conference (ESM 2006)*, pages 60–67, Toulouse, France, October 2006.

[103] R.E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[104] R.E. Tarjan. Enumeration of the Elementary Circuits of a Directed Graph. Technical report, Ithaca, New York, USA, 1972.

[105] A. Thümmler, P. Buchholz, and M. Telek. G-FIT. `http://ls4-www.cs.tu-dortmund.de/home/thummler/pubs.html`.

[106] A. Thümmler, P. Buchholz, and M. Telek. A Novel Approach for Phase-Type Fitting with the EM Algorithm. *IEEE Transactions on Dependable and Secure Computing*, 3:245–258, 2005.

[107] J.C. Tiernan. An Efficient Search Algorithm to find the Elementary Circuits of a Graph. *Communincations ACM*, 13(12):722–726, December 1970.

[108] Ubisense. Ubisense Customers. `http://www.ubisense.net/en/customers`.

[109] W.M.P. van der Aalst. Putting high-level Petri nets to work in industry. *Computers in Industry*, 25(1):45–54, 1994.

[110] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[111] M. Vankipuram, K. Kahol, T. Cohen, and V.L. Patel. Toward Automated Workflow Analysis and Visualization in Clinical Environments. *Journal of Biomedical Informatics*, 44(3):432 – 440, 2011.

[112] F. Wang and P. Liu. Temporal Management of RFID Data. In *Proc. 31st International Conference on Very Large Databases (VLDB 2005)*, pages 1128–1139, Trondheim, Norway, August 2005.

[113] F. Wu, F. Kuo, and L.-W. Liu. The Application of RFID on Drug Safety of Inpatient Nursing Healthcare. In *Proc. 7th International Conference on Electronic Commerce (ICEC 2005)*, pages 85–92, Xi'an, China, August 2005. ACM.

[114] Y. Xue, R.M. Kieckhafer, and F.F. Choobineh. Automated construction of GSPN models for flexible manufacturing systems. *Computers in Industry*, 37:17–25, 1998.

[115] A. Zenie. Colored Stochastic Petri Nets. In *Proc. International Workshop on Timed Petri Nets (PNPM 1985)*, pages 262–271, Torino, Italy, July 1985. IEEE Computer Society.

[116] A. Zimmermann, H. Westphal, and S. Gramlich. Colored Petri Nets for the Performance Evaluation of a Semiconductor Fabrication Facility. In *Proc. 7th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 1999)*, volume 2, pages 1071–1080, Barcelona, Spain, October 1999.

Appendix A

LocTrackJINQS Manual and XML Node Definitions

A.1 LocTrackJINQS Manual

As mentioned in Section 3.1, simulations in LocTrackJINQS can be specified either programmatically or through the GUI. Here, we demonstrate the simulation construction process for both approaches.

In order to construct a simulation programmatically, a new class must be implemented. This must contain the simulation components and parameters, as well as instantiate three objects: `GUI_Sim`, `DrawingPanel` and `JFrame`[1]. The simulation components and parameters must be defined within the `buildNetwork()` method. An example of a class used to instantiate a new simulation environment is shown in Code A.1.

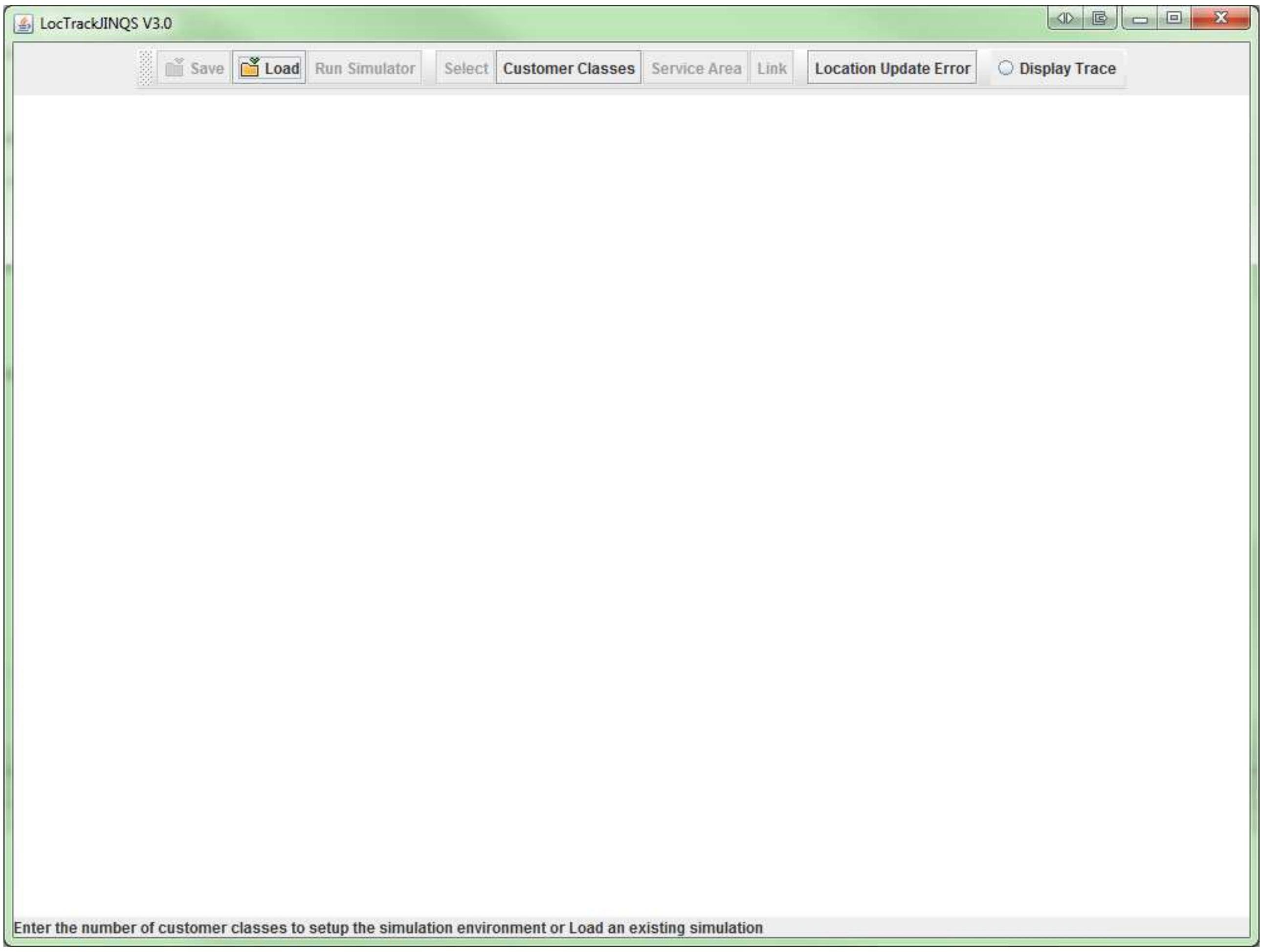
```
public class SampleSimulation{

  GUI_Sim simulator;
  JFrame mainFrame;
  DrawingPanel ownerPanel;

  public SampleSimulator(double stopTime){

     mainFrame = new JFrame("Example Simulation");
     ownerPanel = new DrawingPanel(mainFrame);

     mainFrame.add(ownerPanel);
     mainFrame.setSize(1000,800); // Frame dimensions - adjustable
     mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     mainFrame.setVisible(true);

     simulator = new GUI_Sim(ownerPanel);
     simulator.setTerminateTime(stopTime);
     buildNetwork(); // this method defines the structure and parameters of the simulation

   }

   public void simulate(){

     simulator.run();

   }


}
```

Code A.1: An example class used to instantiate a simulation environment.

If the user wishes to setup the simulation using the GUI, the user must do so through LocTrackJINQS's main window. This is presented to the user when LocTrackJINQS is run as a stand-alone application, as it is shown in Figure A.1. A status bar, placed on the bottom left corner of the main window, displays messages which guide new users through the simulation

[1]The `JFrame` instance is used as a container for the `DrawingPanel`.

construction process.



Enter the number of customer classes to setup the simulation environment or Load an existing simulation

Figure A.1: LocTrackJINQS GUI - main window.

A.1.1 Setting-up the Simulation Environment

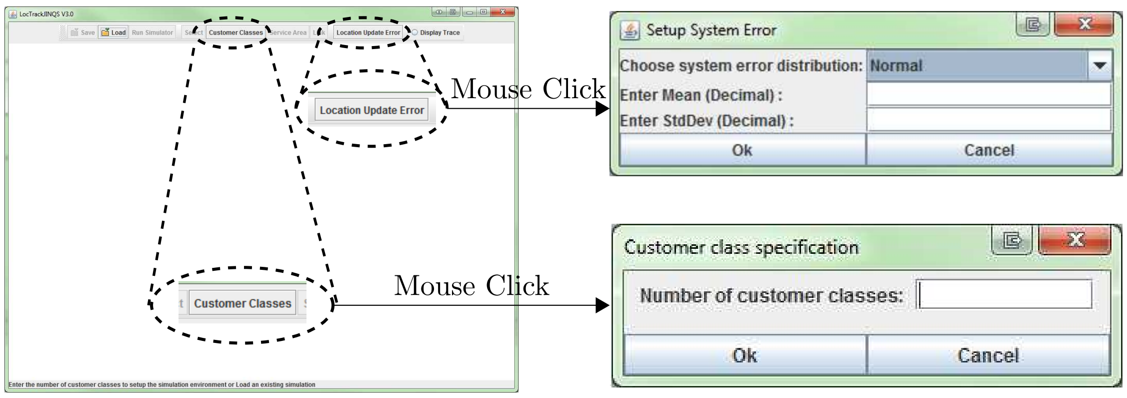
A.1.1.1 Initial Setup

Initially, users must specify the number of different customer classes they wish to include in the simulation. This is required so that LocTrackJINQS can determine the number of parameters that should be expected for each node, e.g. customer speed and service time distribution(s), class priorities, etc.

In the programmatic specification of the simulation, i.e. in the `buildNetwork()` method, the user must first explicitly initialise the `Network` and `NetworkMonitor` classes by invoking their respective initialisation methods, i.e.

```
Network.initialise();
NetworkMonitor.initialiseMonitor();
```

The number of customer classes must then be specified; if the programmatic specification approach is employed, this is done through `setCustomerClass(int)` method of the `Network` class. Alternatively, through the GUI, the user must press the 'Customer Classes' button which will display an input dialog to set-up this value. Similarly, via another dialog, the location update error distribution can specified. These dialogs are shown in Figure A.2. Subsequently, users can create various nodes, specify their parameters and connect them via various types of links.



Choose system error distribution: Normal

Enter the number of customer classes to setup the simulation environment or Load an existing simulation

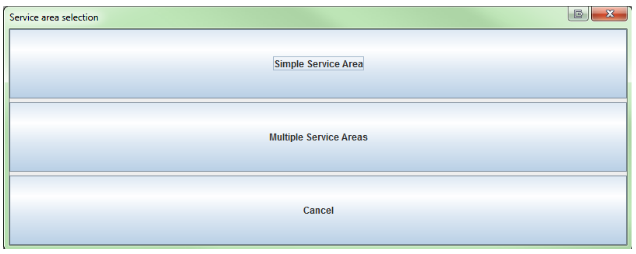
Figure A.2: LocTrackJINQS GUI - customer class and location update error dialog.

A.1.1.2 Creating Nodes

In LocTrackJINQS, service areas[2] (including source and sink) can be differentiated into two main categories: simple and multiple. A simple service area represents a conventional service point consisting of one or more servers which share the same service area and queue. A multiple service area consists of many servers – each with an associated service radius and service time distribution – that share only one queue.

Users that follow the GUI specification approach, can select the category of the service area they wish to create by pressing the 'Service Area' button (see Figure A.3(a)); however, this

[2]A service area (excluding its spatial information) refers to what is known as a node in Queueing Networks.



(a)

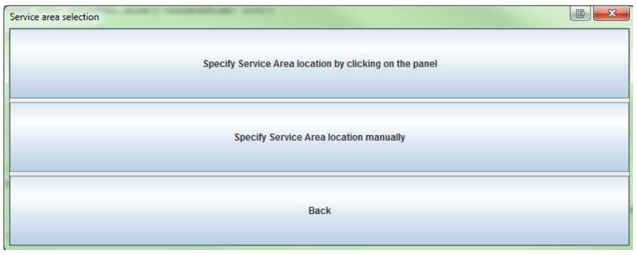

Service area selection
Specify Service Area location by clicking on the panel
Specify Service Area location manually
Back
(b)

Figure A.3: Menu for selecting the category of the service area to be added (left) and input location selection method menu for simple service areas (right).

button becomes enabled once the number of customer classes participating in the simulation is specified[3].

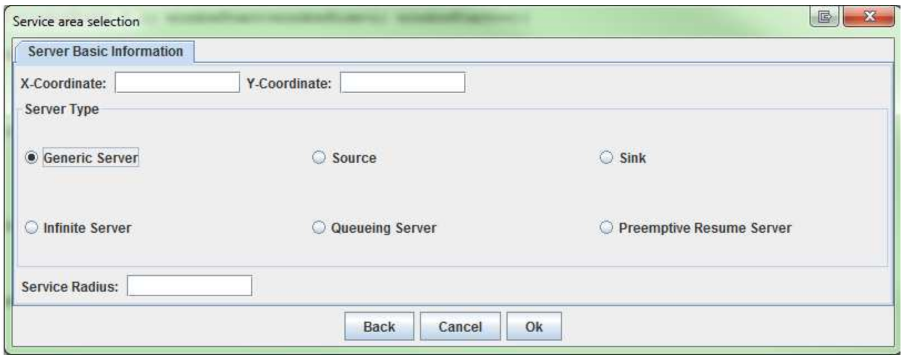
Simple Service Areas

If the simple service area category is selected (cf. Figure A.3(a)), another menu is displayed which allows the user to specify the service area's location by either clicking at some location within the simulation environment (the drawing panel), or by manually entering its coordinates. This menu is shown in Figure A.3(b). Depending on the user's selection, another dialog is created; this dialog, examples of which are shown in Figures A.4, A.5, A.6, A.7 and A.8, enables users to select the type of service area they wish to create. The types of simple service areas that are currently supported include the Generic Server, Source, Sink, Infinite Server, Queueing Server and Preemptive Server. Furthermore, the latter dialog asks for the input required to parameterise the selected type of service area. For each type of parameter, the following metrics are assumed:

1. spatial parameters: metres (m)
2. service time distribution parameters: customers per second (c/s)
3. speed distribution parameters: metres per second (m/s).

The Generic Server provides no real service and may be used as a routing node. Therefore, it only requires the specification of its location and service radius (see Figure A.4). Identical

[3]This is a precaution taken in order to reduce the possibility of error during the simulation specification process.



Server Type: [✓ Generic Server] Source Sink Infinite Server Queueing Server Preemptive Resume Server

Figure A.4: Input dialog for Generic Server.

input is required for the creation of a Sink. Programmatically, these two types of service areas are created by instantiating new objects of their respective classes and setting their location and service radius as follows:

```
Server genericServer = new Server();
genericServer.setCurrentLocation(2.0,2.0);
genericServer.setServiceRadius(0.5);
Network.addNode(genericServer);
```

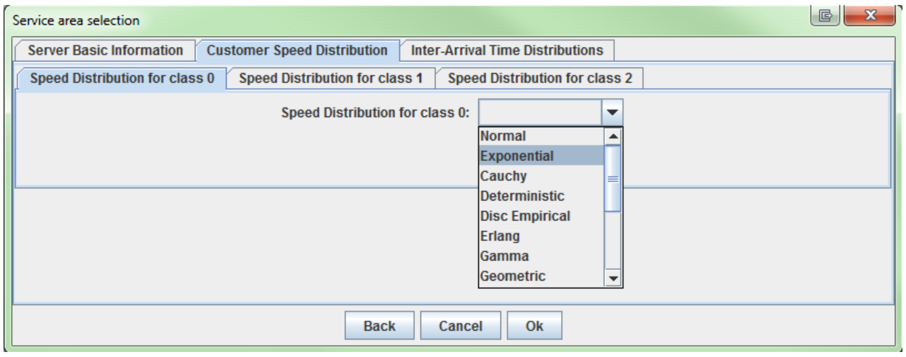

Service area selection
Server Basic Information
Customer Speed Distribution
Inter-Arrival Time Distributions
Speed Distribution for class 0
Speed Distribution for class 1
Speed Distribution for class 2
Speed Distribution for class 0:
Normal
Exponential
Cauchy
Deterministic
Disc Empirical
Erlang
Gamma
Geometric
Back
Cancel
Ok

Figure A.5: Input dialog for Source.

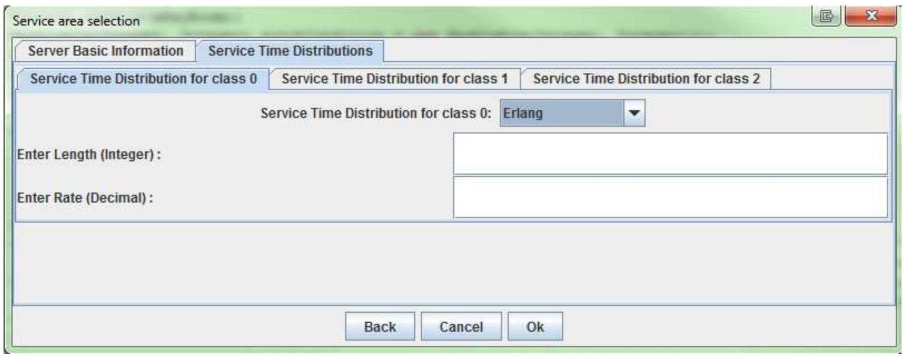
The `addNode( · )` method must be called whenever a new service area is created so that this can be added to the Queueing Network.

The Source requires no service radius. However, in addition to the location specification, it requires users to specify – for each customer class – the type and parameters of the customers' speed distribution, and similarly, the customers' inter-arrival time distribution. The type of the distribution is selected through a drop-down menu which contains all available distributions (see Figure A.5). The programmatic specification of source nodes differs for single and multiple customer classes. If there is only one class of customers, then an instance of the `Source` class should be created; otherwise, a `SourceWithClassDist` object must be instantiated. A `Source` is instantiated as follows:

```
DistributionSampler arrivalSampler = new Exp(0.05);
Source source = new Source(arrivalSampler);
DistributionSampler speedSampler = new Normal(0.5,0.1);
source.setCustomerSpeedDistribution(speedSampler);
source.setCurrentLocation(5.0,5.0);
Network.addNode(source);
```

The `SourceWithClassDist` instantiation requires an array of inter-arrival and speed distributions, one for each customer class. An example implementation, involving two customer classes, can be seen below. We note that the first argument in the constructor of `SourceWithClassDist` corresponds to the number of customer classes that are supported and is of type `int`.

```
DistributionSampler[] arrivalSampler = new DistributionSampler[2];
arrivalSampler[0] = new Exp(0.05);
arrivalSampler[1] = new Erlang(2,0.08);
SourceWithClassDist source = new SourceWithClassDist(2,arrivalSampler);
DistributionSampler[2] speedSampler = new DistributionSampler[2];
speedSampler[0] = new Normal(0.5,0.1);
speedSampler[1] = new Normal(0.3,0.1);
source.setCustomerSpeedDistribution(speedSampler);
source.setCurrentLocation(5.0,5.0);
Network.addNode(source);
```


Service area selection
Server Basic Information
Service Time Distributions
Service Time Distribution for class 0
Service Time Distribution for class 1
Service Time Distribution for class 2
Back
Cancel
Ok

Service Time Distribution for class 0: Erlang

Enter Length (Integer) :

Enter Rate (Decimal) :

Figure A.6: Input dialog for Infinite Server.

The Infinite Server requires the same spatial information as the Generic Server, i.e. location and service radius, as well as the specification of a service time distribution for each supported customer class. Once the distribution type is selected from the drop-down list (similar to

Figure A.5), the appropriate parameters, along with type of input they require, are displayed to the user (see Figure A.6). In order to define service time distribution(s) in the programmatic implementation of Infinite Server, the `Delay` or `DelayWithCustClass` classes must be used. For example, if one class of customers is supported, then this should be implemented as:

```
Delay serviceTimeDelay = new Delay(new Exp(0.05));
InfiniteServer infServer = new InfiniteServer(serviceTimeDelay);
```

and for two customer classes as:

```
DistributionSampler[] distributions = new DistributionSampler[2];
distributions[0] = new Exp(0.05);
distributions[1] = new Erlang(5,0.1);
DelayWithCustClass serviceTimeDelay = new DelayWithCustClass(distributions);
InfiniteServer infServer = new InfiniteServer(serviceTimeDelay);
```

The Queueing Server is similar to the Infinite Server, but it requires some additional information: the queueing discipline, the queue capacity and the number of servers contained in the service area (see Figure A.7). The number of servers is specified in the service time distribution tab. Programmatically, for a Queueing Server, the service time distributions are specified in the same way as for the Infinite Server. The constructor of a `QueueingServer` takes three arguments: the service time, number of servers and queueing discipline, i.e.

```
QueueingServer queueServer = new QueueingServer(serviceTimeDelay,1,new FIFOQueue());
```

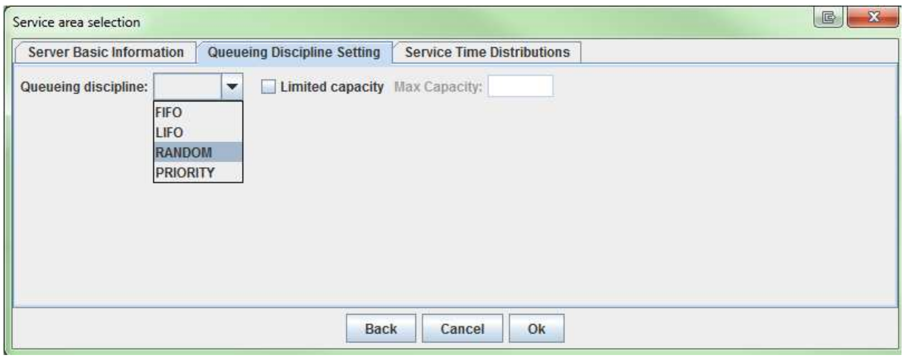
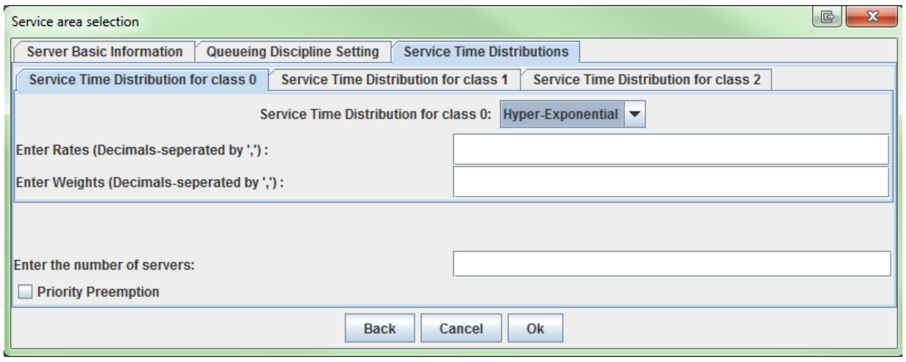


Figure A.7: Input dialog for Queueing Server.

The last type of service area that can currently be created through LocTrackJINQS's GUI is the Preemptive Resume Server; this type supports service preemption. Optionally, the user can specify priority service preemption which uses as criterion each customer's class. Under this service policy, an on-going service of a low priority customer, will be interrupted so that customers of higher priority can be serviced (see Figure A.8). The interrupted service of customers is resumed immediately after the service of the higher priority customers is completed. `PreemptiveResumeServer` objects are constructed in the same way as `QueueingServer` ones. However, if the user wishes to implement the priority preemption service policy, the priority of


Service area selection

Server Basic Information | Queueing Discipline Setting | Service Time Distributions

Service Time Distribution for class 0 | Service Time Distribution for class 1 | Service Time Distribution for class 2

Service Time Distribution for class 0: Hyper-Exponential

Enter Rates (Decimals-seperated by ','):

Enter Weights (Decimals-seperated by ','):

Enter the number of servers:

☐ Priority Preemption

Back | Cancel | Ok

Figure A.8: Input dialog for Preemptive Server.

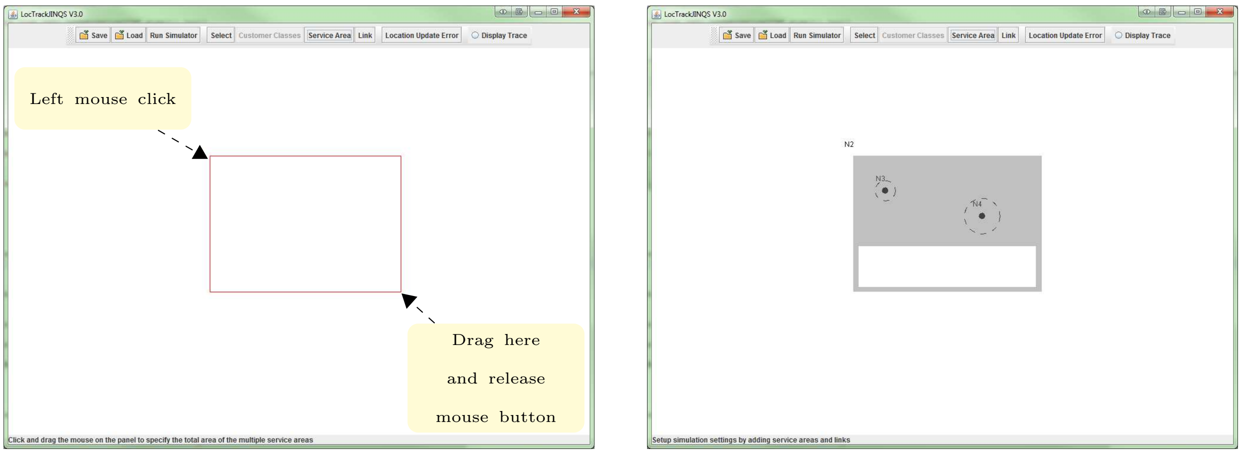
each customer class must be explicitly defined. An example, for two customer classes is shown below:

```
int[] priorityPolicy = new int[2];
priorityPolicy[0] = 1;
priorityPolicy[1] = 0;
preemptiveResumeServer.setPriorityAssignPolicy(priorityPolicy);
```

This example specifies that customers of class 0 have higher priority than customers of class 1. We note that if a priority queueing discipline is used, then the assigned service preemption priorities must be the same as the queueing ones.

Multiple Service Areas

The creation of multiple service areas is quite similar to that of simple service areas, especially when they are created though LocTrackJINQS's interface. First, the user must define the (rectangular) area in which the queue and simple service areas are to be included. This can be easily done by pressing (and holding) the left mouse button and dragging the mouse in the drawing panel (see Figure A.9(a)). As soon as the mouse button is released, the area is drawn on the panel and a dialog is displayed to the user which requests the number of simple service areas to be added. The location of each such service area must be specified by clicking on a location within the grey area. The dialog which allows the user to specify the parameters of the service area is then displayed. This is similar to the corresponding dialog used for simple service areas, e.g. see Figure A.7. We note that the only type of service area that is currently supported by multiple service areas is the Queueing Server. The location and parameter specification process, presented above, is repeated for each Queueing Server that is to be included in the multiple service area. An example of multiple service area, consisting of two Queueing Servers is depicted in Figure A.9(b).



Click and drag the mouse on the panel to specify the total area of the multiple service areas

Setup simulation settings by adding service areas and links

(a) (b)

Figure A.9: Defining the area for multiple service area (left) and the result after adding two service areas (right). The white filled rectangle within the multiple service area (grey area) represents the shared queueing area.

Programmatically, a multiple service area is created by instantiating a `MultiQueueingServers` class, i.e.

```
MultiQueueingServers multiServer = new MultiQueueingServers(10,5,5,5);
```

The constructor arguments are of type `double`. The first two numbers specify the location of multiple service area (the location of the rectangle's top left corner). The third number specifies the service area's width and the fourth one its height. As mentioned above, the only type of simple service area that may currently be added to a multiple service area is the Queueing Server. However, in this case, the Queueing Server is not identical to the one presented earlier;

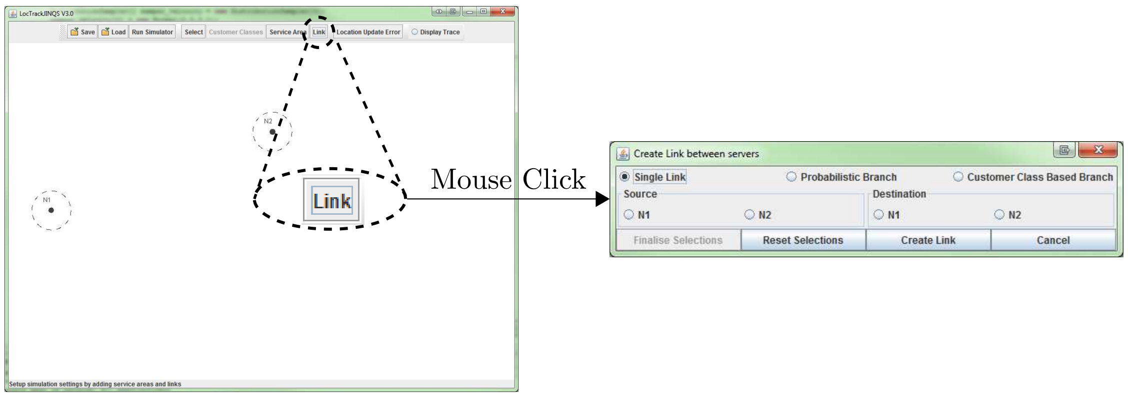
this is a special type which uses the multiple service area's queueing area as its queue, and it is implemented by the `QueueingServerExternQueue` class. Furthermore, the location of such service areas must be within the multiple service area. `QueueingServerExternQueue` objects are instantiated and added to a multiple service area in the following way:

```
QueueingServerExternQueue s1 = new QueueingServerExternQueue(new Delay(new Exp(0.5)),
1, multiServ.getQueueArea());
s1.setCurrentLocation(11, 6);
s1.setServiceRadius(0.3);
multiServ.addServer(s1);
```

As it happens with simple service areas, when a multiple service are is parameterised and its Queueing Servers are added, it must also be added to the network via the `addNode( · )` method.

A.1.1.3 Creating Links

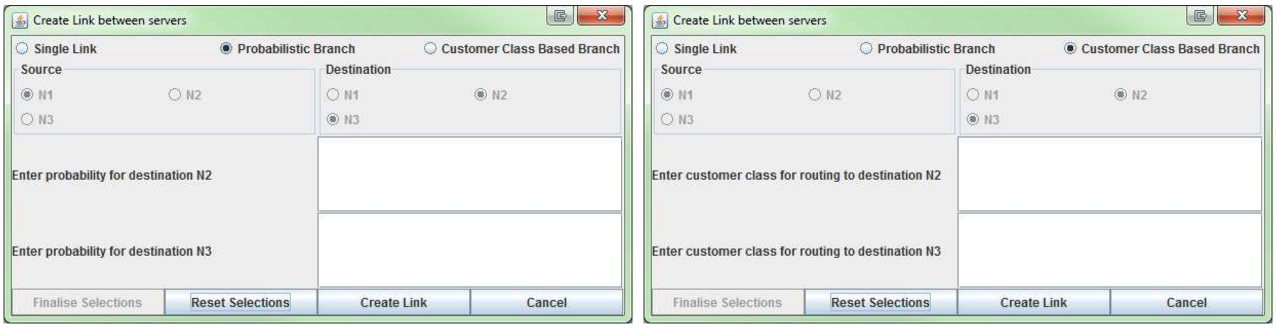
In LocTrackJINQS, there are two major categories of links: single and routing. A single link provides a one-to-one connection between two service areas. A routing link is used to connect one service area (source) to two or more service areas (destinations). Links can be added through the GUI by clicking the 'Link' button. When this button is pressed, the link dialog is displayed to the user (see Figure A.11).



Setup simulation settings by adding service areas and links

Figure A.10: LocTrackJINQS GUI - link dialog.

If the user wishes to create a single link, it is sufficient to select the source node and the destination node; the nodes that are available to be used as a source and as a destination are


Create Link between servers
Single Link
Probabilistic Branch
Customer Class Based Branch
Source
Destination
N1
N2
N3
Enter probability for destination N2
Enter probability for destination N3
Finalise Selections
Reset Selections
Create Link
Cancel
Create Link between servers
Single Link
Probabilistic Branch
Customer Class Based Branch
Source
Destination
N1
N2
N3
Enter customer class for routing to destination N2
Enter customer class for routing to destination N3
Finalise Selections
Reset Selections
Create Link
Cancel

(a) (b)

Figure A.11: Link dialog for specifying the parameters of the probabilistic link (left) and customer class based link (right).

listed in the dialog. The link is created once the 'Create Link' button is pressed, provided that the source and destination nodes have been selected.

Routing links consist of two types: probabilistic and customer class based. In order to create a probabilistic link, the user must first select the 'Probabilistic Branch' option (cf. Figure A.11). Next, the source and destinations must be selected. Once these are selected, the user must specify the routing probability for each destination node. This is done by pressing the 'Finalise Selection' button; when this button is pressed, the routing probability specification panel is displayed (see Figure A.11(a)). The link is created by pressing the 'Create Link' button. The customer class based link may only be used when multiple customer classes are supported in the current simulation. Via this type of routing link, the user can create a routing rule based on each customer's class. To create a customer class based link, the user must first select the 'Customer Class Based Branch' option. Subsequently, the source and destination nodes must be chosen and finalised; this process is performed in the same way as in the probabilistic link case. However, in this case, when the 'Finalise Selection' button is pressed, another panel is displayed that allows the user to specify which customer class is routed from the source node to each destination node (see Figure A.11(b)).

When a link is created, it is drawn as a straight line segment(s) connecting the source and the destination(s) nodes. However, the path of customers forwarded by that link can be customised via the addition of break points. If one presses the right mouse button on a link contained in the drawing panel of LocTrackJINQS (the 'Select' button must be pressed first), a pop-up menu will be displayed. This menu allows the user to add a break points or delete the selected link (see Figure A.12). If a break point is added, it can be then dragged to another location and thus customise the representation of the link which ultimately defines the customer path.

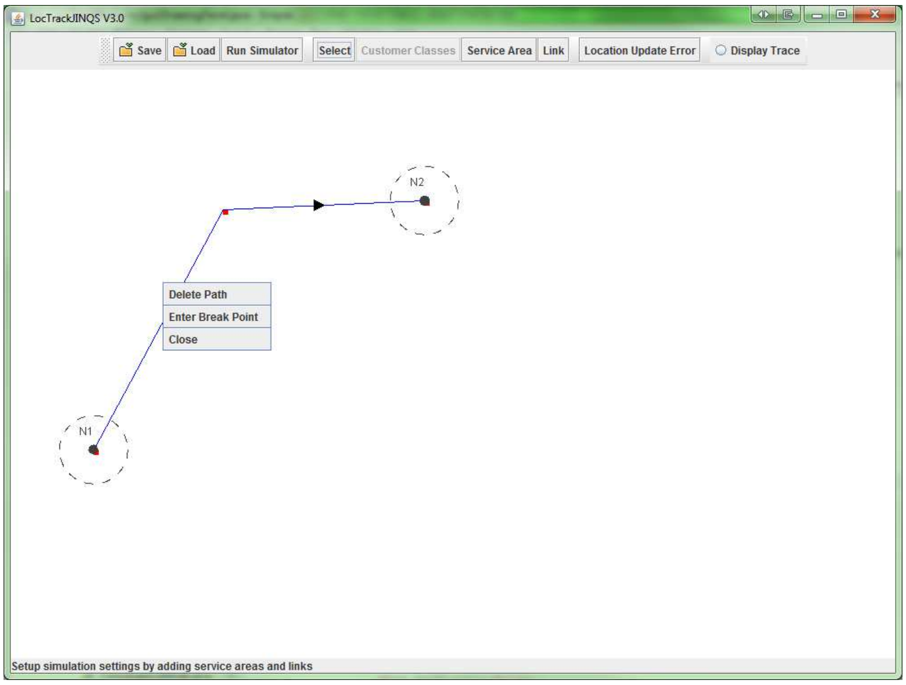


Figure A.12: Pop-up menu for deleting links and entering new break points.

In order to create links programmatically, the link's representation, i.e. the collection of straight line segments, must be first specified. For example, if we consider a single link connecting two nodes, S0 and S1, a possible representation, which includes three break points, may be defined as:

```
ArrayList<Line2D.Double> path = new ArrayList<Line2D.Double>();
path.add(new Line2D.Double(S0.giveLocation(), new Point2D.Double(25,12.5)));
path.add(new Line2D.Double(new Point2D.Double(25,12.5), new Point2D.Double(25,18)));
path.add(new Line2D.Double(new Point2D.Double(25,18), new Point2D.Double(38,18)));
path.add(new Line2D.Double(new Point2D.Double(38,18),S1.getLocation()));
```

Then, the link can be created in the following way:

```
TransportLink l1 = new TransportLink(S1);
l1.setPathSegments(path);
S0.setLink(l1);
Network.addLink(l1);
```

A similar approach applies for the creation of routing links. An example, containing both probabilistic and customer class-based routing links, can be seen in the following section.

A.1.2 An Example

Here, we demonstrate the programmatic simulation construction process through an example. The complete implementation of the `buildNetwork()` method can be seen in Code A.2. This implementation is associated with Code A.1, which creates the frame and initiates the simulation, and it produces the network depicted in Figure A.13.

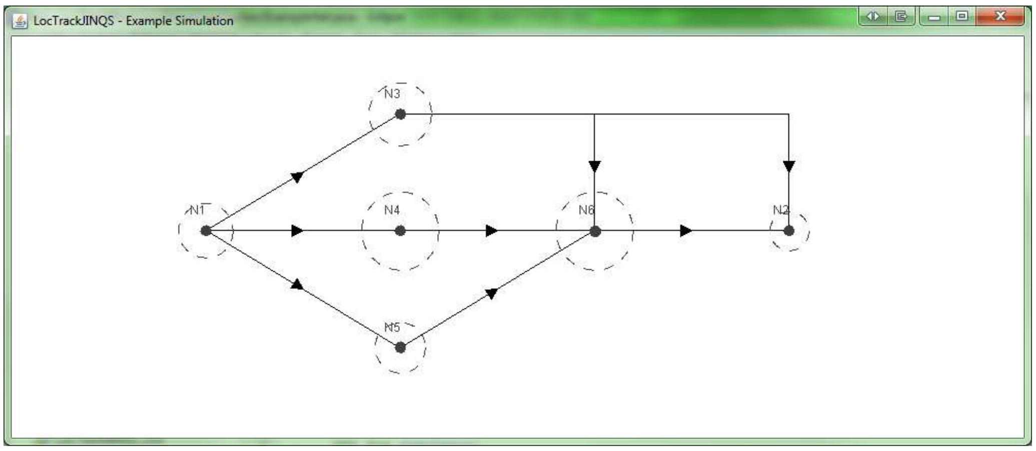


Figure A.13: An example of a simulation environment constructed via the `buildNetwork()` method shown in Code A.2.

```
public void buildNetwork(){

   Network.initialise(); // initialise network
   NetworkMonitor.initialiseMonitor(); // initialise network monitor
   Network.setCustomerClass(3); // set the number of customer classes

   DistributionSampler errorDistribution = new Normal(0.15, 0.2); // specify error
       distribution
   Network.systemError = errorDistribution; // assign error distribution

   /*Creating source*/

   DistributionSampler[] sampler_arrival = new DistributionSampler[Network.getCustomerClass
       ()];
   sampler_arrival[0] = new Exp(0.01); // arrival distribution for customer class 0
   sampler_arrival[1] = new Exp(0.008); // arrival distribution for customer class 1
   sampler_arrival[2] = new Exp(0.004); // arrival distribution for customer class 2

   DistributionSampler[] samper_velocity = new DistributionSampler[Network.getCustomerClass
       ()];
   samper_velocity[0] = new Normal(0.38,0.1); // speed distribution for class 0
   samper_velocity[1] = new Normal(0.25, 0.1); // speed distribution for class 1
   samper_velocity[2] = new Normal(0.4,0.2); // speed distribution for class 2

   SourceWithClassDist source = new SourceWithClassDist(null, Network.getCustomerClass(),
       sampler_arrival);
   source.setCurrentLocation(5,5);
   source.setCustomerSpeedDistribution(samper_velocity);
   source.setTagged(false);
   Network.addNode(source);

   /*End of source*/

   /*Creating sink*/

```

```java
    Sink sink = new Sink();
    sink.setCurrentLocation(20, 5);
    sink.setServiceRadius(0.5);
    sink.setTagged(false);
    Network.addNode(sink);

    /*End of sink*/

    /*Creating Queueing Server S1*/

    // for customer class 0 - no other customer classes are routed here

    Delay s1ServiceTime = new Delay(new Erlang2(3,0.065)); // service time delay - follows
        an Erlang distribution
    QueueingServer s1 = new  QueueingServer(s1ServiceTime, 1, new FIFOQueue());
    s1.setCurrentLocation(10,2);
    s1.setServiceRadius(0.8);
    s1.setTagged(false);
    Network.addNode(s1);

    /*End of S1*/


    /*Creating Queueing Server S2*/

    // for customer class 1 - no other customer classes are routed here

    // define the parameters of hyper-Erlang distribution
    double[] herdWeights = {0.5,0.5};
    double[] herdRates = {0.05,0.48};
    int[] herdLengths = {2,2};

    Delay s2ServiceTime = new Delay(new HyperErlang(herdRates, herdLengths, herdWeights));
        // service time delay - follows a hyper-Erlang distribution
    QueueingServer s2 = new  QueueingServer(s2ServiceTime, 1, new FIFOQueue());
    s2.setCurrentLocation(10,5);
    s2.setServiceRadius(1);
    s2.setTagged(false);
    Network.addNode(s2);

    /*End of S2*/


    /*Creating Queueing Server S3*/

    // for customer class 2 - no other customer classes are routed here

    Delay s3ServiceTime = new Delay(new Exp(0.005)); // service time delay - follows an
        Exponential distribution
    QueueingServer s3 = new  QueueingServer(s3ServiceTime, 1, new FIFOQueue());
    s3.setCurrentLocation(10,8);
    s3.setServiceRadius(0.65);
    s3.setTagged(false);
    Network.addNode(s3);

    /*End of S3*/




    /*Creating Queueing Server S4*/

    // all customer classes are routed here

    DistributionSampler[] s4ServiceTime = new DistributionSampler[Network.getCustomerClass()
        ];
    s4ServiceTime[0] = new Exp(0.05); // service time distribution for class 0
    s4ServiceTime[1] = new Erlang2(3,0.08); // service time distribution for class 1
    s4ServiceTime[2] = new Exp(0.004); // service time distribution for class 2
```

```java

QueueingServer s4 = new QueueingServer(new DelayWithCustClass(s4ServiceTime), 1,new
    RandomQueue());
s4.setCurrentLocation(15,5);
s4.setServiceRadius(1);
s4.setTagged(false);
Network.addNode(s4);

/*End of S4*/


/*Creating class based link L1 connecting source to S1, S2 and S3*/

// path1 connects source to S1
ArrayList<Line2D.Double> path1 = new ArrayList<Line2D.Double>();
path1.add(new Line2D.Double(source.getLocation(), s1.getLocation()));
this.ownerPanel.addCustomPath(path1); // add path in drawing panel

// path2 connects source to S2
ArrayList<Line2D.Double> path2 = new ArrayList<Line2D.Double>();
path2.add(new Line2D.Double(source.getLocation(), s2.getLocation()));
this.ownerPanel.addCustomPath(path2); // add path in drawing panel

// path3 connects source to S3
ArrayList<Line2D.Double> path3 = new ArrayList<Line2D.Double>();
path3.add(new Line2D.Double(source.getLocation(), s3.getLocation()));
this.ownerPanel.addCustomPath(path3); // add path in drawing panel

INode[] destinationsForl1 = {s1,s2,s3};
int[] l1RoutingRule = {0,1,2}; // each entry corresponds to a customer class and
    contains the index of the destination for that customer class
ClassBasedRouting l1 = new ClassBasedRouting(destinationsForl1, Link.TRANSPORTLINK,
    l1RoutingRule);
((TransportLink)l1.getLink(0)).setPathSegments(path1);
((TransportLink)l1.getLink(0)).setOwner(source);
((TransportLink)l1.getLink(1)).setPathSegments(path2);
((TransportLink)l1.getLink(1)).setOwner(source);
((TransportLink)l1.getLink(2)).setPathSegments(path3);
((TransportLink)l1.getLink(2)).setOwner(source);
source.setLink(l1);
Network.addLink(l1);

/*End of L1*/


/*Creating probabilistic link L2 connecting S1 to S4 and sink*/

// path4 connects S1 to S4
ArrayList<Line2D.Double> path4 = new ArrayList<Line2D.Double>();
path4.add(new Line2D.Double(s1.getLocation(), new Point2D.Double(15,2)));
path4.add(new Line2D.Double(new Point2D.Double(15,2), s4.getLocation()));
this.ownerPanel.addCustomPath(path4); // add path in drawing panel

// path5 connects S1 to sink
ArrayList<Line2D.Double> path5 = new ArrayList<Line2D.Double>();
path5.add(new Line2D.Double(s1.getLocation(), new Point2D.Double(20,2)));
path5.add(new Line2D.Double(new Point2D.Double(20,2), sink.getLocation()));
this.ownerPanel.addCustomPath(path5); // add path in drawing panel

INode[] destinationsForl2 = {s4,sink};
double[] l2Probabilities = {0.60,0.40}; // routing probability for each destination
    respectively
ProbabilisticRouting l2 = new ProbabilisticRouting(destinationsForl2, Link.
    TRANSPORTLINK, l2Probabilities);
((TransportLink)l2.getLink(0)).setPathSegments(path4);
((TransportLink)l2.getLink(0)).setOwner(s1);
((TransportLink)l2.getLink(1)).setPathSegments(path5);
((TransportLink)l2.getLink(1)).setOwner(s1);

```

```
    s1.setLink(l2);
    Network.addLink(l2);

    /*End of L2*/

    /*Creating single link L3 connecting S2 to S4*/

    // path6 connects S2 to S4
    ArrayList<Line2D.Double> path6 = new ArrayList<Line2D.Double>();
    path6.add(new Line2D.Double(S2.getLocation(), s4.getLocation()));
    this.ownerPanel.addCustomPath(path6); // add path in drawing panel

    TransportLink l3 = new TransportLink(s4);
    l3.setPathSegments(path6);
    s2.setLink(l3);
    Network.addLink(l3);

    /*End of L3*/


    /*Creating single link L4 connecting S3 to S4*/

    // path7 connects S3 to S4
    ArrayList<Line2D.Double> path7 = new ArrayList<Line2D.Double>();
    path7.add(new Line2D.Double(S3.getLocation(), s4.getLocation()));
    this.ownerPanel.addCustomPath(path7); // add path in drawing panel

    TransportLink l4 = new TransportLink(s4);
    l4.setPathSegments(path7);
    s3.setLink(l4);
    Network.addLink(l4);

    /*End of L4*/

    /*Creating single link L5 connecting S4 to Sink*/

    // path8 connects S4 to sink
    ArrayList<Line2D.Double> path8 = new ArrayList<Line2D.Double>();
    path8.add(new Line2D.Double(S4.getLocation(), sink.getLocation()));
    this.ownerPanel.addCustomPath(path8); // add path in drawing panel

    TransportLink l5 = new TransportLink(sink);
    l5.setPathSegments(path8);
    s4.setLink(l5);
    Network.addLink(l5);

    /*End of L5*/
}
```

Code A.2: An example of a programmatic network setup.

A.2 XML Node Definitions

Distribution

```
<distribution>
<type>"Distribution Type"</type>
<parameter name = "ParameterName">"Parameter Value"</parameter>
                                .
                                .
                                .
<parameter name = "ParameterName">"Parameter Value"</parameter>
</distribution>
```

System Error

```
<locationUpdateError>
"Distribution Definition"
</locationUpdateError>
```

Queue

```
<type>"Queue type"</type>
<capacity>"Queue Capacity"</capacity>
<supportedCustomerClasses>"Customer Class Values"</supportedCustomerClasses>
<subQueueDiscipline>"Sub-queue Type"</subQueueDiscipline>
```

Delay

```
<serviceTimeDistribution>
"Distribution Definition"
</serviceTimeDistribution>
```

Queueing Area (for multiple servers sharing a common queue)

```
<queueingArea id = "Queueing Area ID">
<type>abstract</type>
<location>
<Point2D id = "pt1">
<x>"Upper Left X-Coordinate"</x>
<y>"Upper Left Y-Coordinate"</y>
</Point2D>
</location>
<width>"Width Value"</width>
<height>"Height Value"</height>
<queueingDiscipline>
"Queue Definition"
</queueingDiscipline>
</queueingArea>
```

Link

```
<link id = "Link Source ID" to "Link Destination 1 ID,...,Link Destination N ID">
<type>"Link Type"</type>
<owner>"Source ID"</owner>
<target id = "1">"Destination 1 ID"</target>
                        .
                        .
                        .
<target id = "N">"Destination N ID"</target>
<probabilities>
<probability id = "Destination 1 ID">"Probability Value"</probability>
                                        .
                                        .
                                        .
<probability id = "Destination N ID">"Probability Value"</probability>
</probabilities>
<customerClassDestinations>
<classRouting id ="Destination 1 ID">"Customer Class Value"</classRouting>
                                        .
                                        .
                                        .
<classRouting id ="Destination N ID">"Customer Class Value"</classRouting>
</customerClassDestinations>
<pathsAndPoints>
<path id ="Source ID" to "Destination 1 ID">
<Point2D id = "pt 1">
<x>"Point 1 X-Coordinate"</x>
<y>"Point 1 Y-Coordinate"</y>
</Point2D>
              .
              .
              .
<Point2D id = "pt K">
<x>"Point K X-Coordinate"</x>
<y>"Point K Y-Coordinate"</y>
</Point2D>
</path>
                        .
                        .
                        .
<path id ="Source ID" to "Destination N ID">
<Point2D id = "pt 1">
<x>"Point 1 X-Coordinate"</x>
<y>"Point 1 Y-Coordinate"</y>
</Point2D>
              .
              .
              .
<Point2D id = "pt K">
<x>"Point K X-Coordinate"</x>
<y>"Point K Y-Coordinate"</y>
</Point2D>
</path>
</pathsAndPoints>
</link>
```

Node (Simple Service Area)

```
<node id = "Node ID">
<tagType>Server</tagType>
<tagged>"True or False"</tagged>
<updateRate>"Tag Update Rate Value"</updateRate>
<location>
<Point2D id = "pt1">
<x>"X-Coordinate Value"</x>
<y>"Y-Coordinate Value"</y>
</Point2D>
</location>
<type>"Node Type"</type>
<serviceRadius>"Service Radius Value"</serviceRadius>
<numberOfServers>"Server Value"</numberOfServers>
<servicePriorityPolicy id = "Customer Class">"Priority Value"</servicePriorityPolicy>
"Delay Definition"
<interarrivalTimeDistribution>
"Distribution Definition"
</interarrivalTimeDistribution>
<velocityDistribution>
"Distribution Definition"
</velocityDistribution>
<queueingDiscipline>
"Queue Definition"
</queueingDiscipline>
</node>
```

Multi-Queueing Server (Multiple service areas sharing a common queue)

```
<multiQueueingServer id = "ID">
<x>"Upper Left X-Coordinate"</x>
<y>"Upper Left Y-Coordinate"</y>
<width>"Width Value"</width>
<height>"Height Value"</height>
<serverCluster>
<server id = "1">"Service Area 1 ID"</server>
                         .
                         .
                         .
<server id = "N">"Service Area N ID"</server>
</serverCluster>
"Queueing Area Definition"
"Internal Link Definition"
</multiQueueingServer>
```

Appendix B

Additional Results for Chapter 6

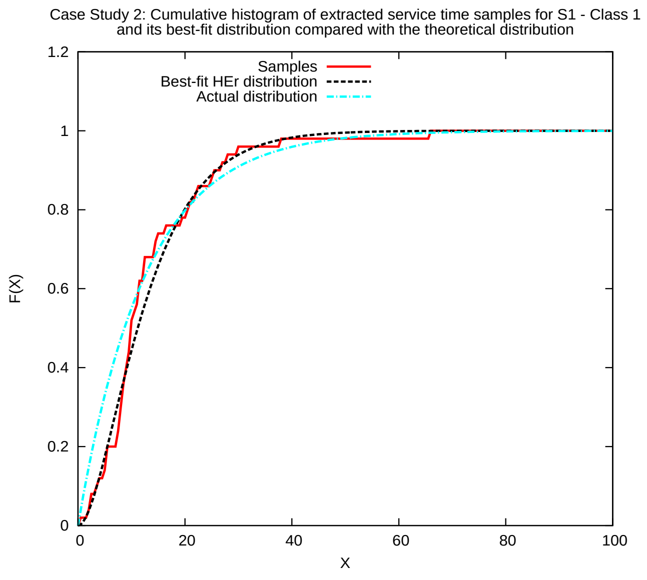
B.1 Case Study 2

		Service Time Density	Fitted HErD Parameters			Relative Entropy
			Phase Lengths	Rate (3 d.p.)	Weights (3 d.p.)	(3 d.p.)
Customer Class 1	S1	Exp(0.08)	2	0.150	1.0	0.195
	S2	$\text{Erlang}(5, 0.15)$	6	0.158	1.0	0.060
	S3	Exp(0.028)	1	0.023	1.0	0.015
	S4 - (1st visit)	Exp(0.035)	1	0.028	1.0	0.022
	S4 - (2nd visit)	Exp(0.035)	1	0.028	1.0	0.021
	S5	Exp(0.04)	1	0.038	1.0	0.001
	S6	$\text{Erlang}(2, 0.085)$	3,6	0.091,0.373	0.560,0.440	0.105
Customer Class 2	S1	Hyper-Exp(0.5,0.35,0.15;0.05,0.08,0.12)	2	0.097	1.0	0.311
	S2	$\text{HErD}(1, 3; 0.6, 0.4; 0.02, 0.12)$	1	0.025	1.0	0.036
	S3	$\text{Erlang}(3, 0.065)$	3	0.058	1.0	0.021
	S4 - (1st visit)	$\text{Erlang}(8, 0.2)$	7	0.169	1.0	0.009
	S4 - (2nd visit)	$\text{Erlang}(8, 0.2)$	1,8	0.899,0.211	0.038,0.962	0.050
	S5	$\text{Erlang}(3, 0.1)$	3	0.107	1.0	0.007
	S6	Exp(0.05)	2	0.098	1.0	0.192

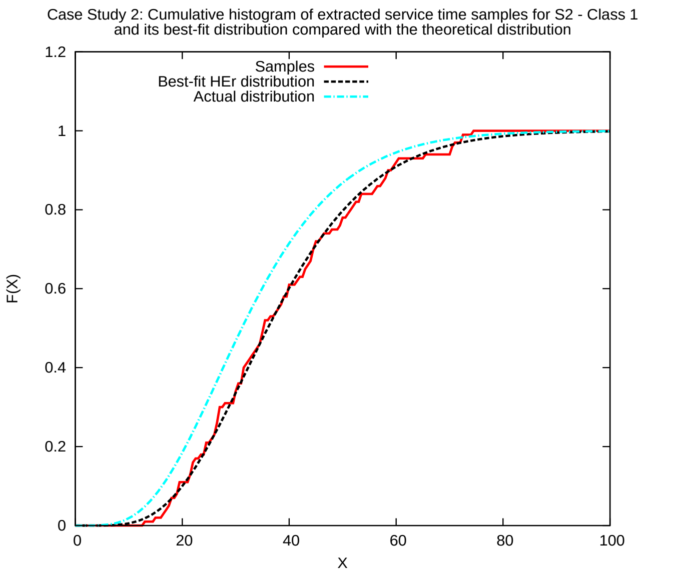
Table B.1: The HErD parameters fitted by G-FIT for each service area's service time density with the relative entropy (in nat) between the theoretical and fitted probability density function for case study two of Chapter 6. Results are shown for the second (class 1) and third (class 2) customer class. The parameters of the HErDs represent the phase lengths, weights and rate for each branch respectively, separated by a semi-colon.

		Customer Class 1		Customer Class 2	
S1	Test Statistic	0.1394		0.1461	
	α	0.1	0.05	0.1	0.05
	Critical Values	0.1513	0.1725	0.1443	0.1645
	Compatible ?	Yes	Yes	No	Yes
S2	Test Statistic	0.0456		0.1002	
	α	0.1	0.05	0.1	0.05
	Critical Values	0.1070	0.1220	0.1359	0.1549
	Compatible ?	Yes	Yes	Yes	Yes
S3	Test Statistic	0.0971		0.0758	
	α	0.1	0.05	0.1	0.05
	Critical Values	0.1513	0.1725	0.1544	0.1761
	Compatible ?	Yes	Yes	Yes	Yes
S4 - 1st Visit	Test Statistic	0.0895		0.0809	
	α	0.1	0.05	0.1	0.05
	Critical Values	0.1081	0.1232	0.1393	0.1588
	Compatible ?	Yes	Yes	Yes	Yes
S4 - 2nd Visit	Test Statistic	0.0800		0.1109	
	α	0.1	0.05	0.1	0.05
	Critical Values	0.1227	0.1399	0.1470	0.1676
	Compatible ?	Yes	Yes	Yes	Yes
S5	Test Statistic	0.0900		0.0742	
	α	0.1	0.05	0.1	0.05
	Critical Values	0.1081	0.1232	0.1359	0.1549
	Compatible ?	Yes	Yes	Yes	Yes
S6	Test Statistic	0.0431		0.0980	
	α	0.1	0.05	0.1	0.05
	Critical Values	0.1081	0.1232	0.1359	0.1549
	Compatible ?	Yes	Yes	Yes	Yes

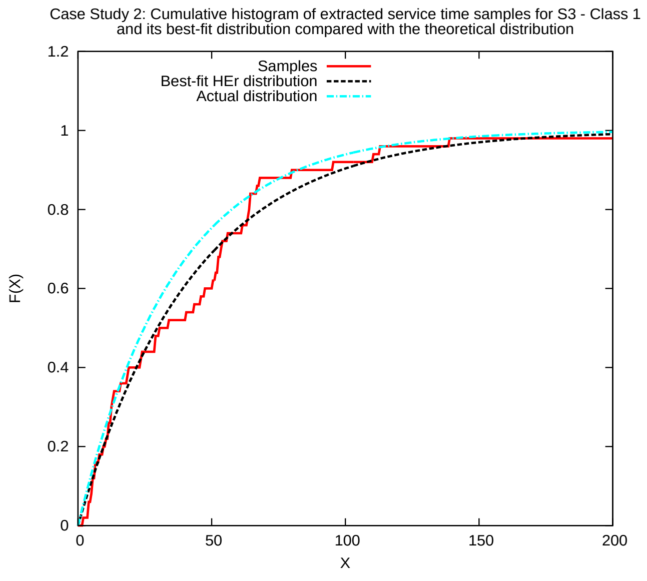
Table B.2: Kolmogorov-Smirnov test at significance levels 0.1 and 0.05 applied to the extracted service time samples for each service point from the second case study of Chapter 6. The null hypothesis is that each extracted sample belongs to the corresponding best-fitted HErD.



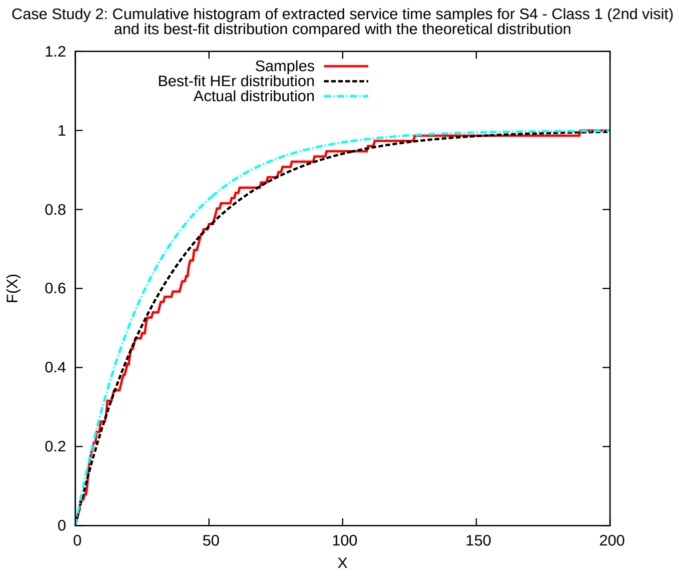
(a)



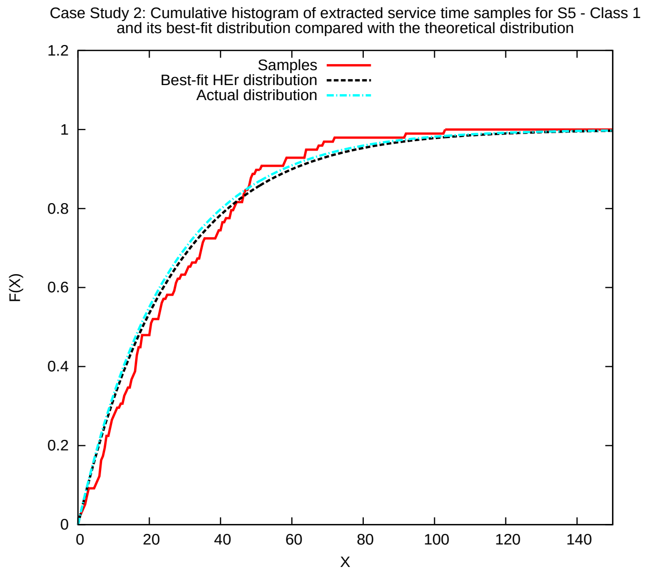
(b)



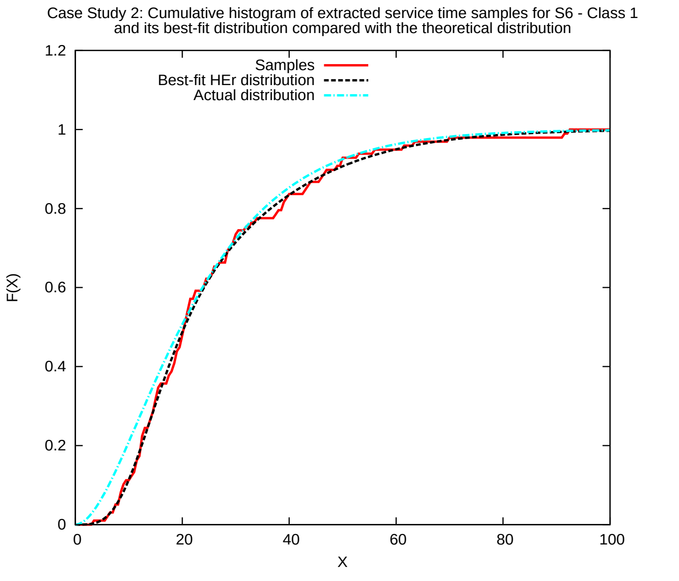
(c)



(d)

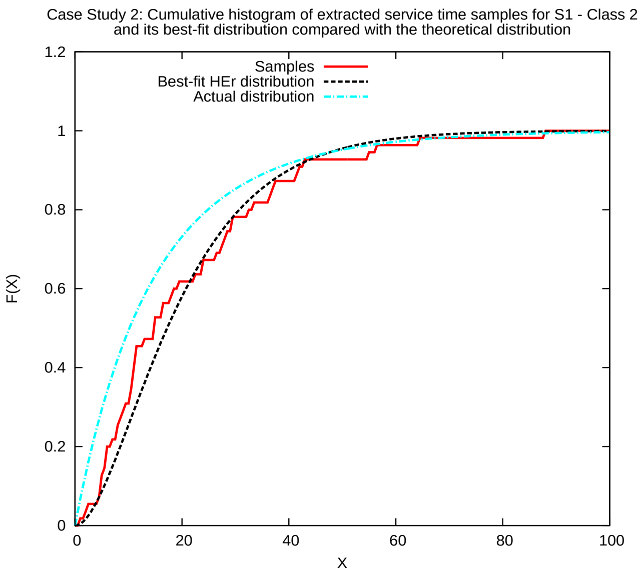


(e)

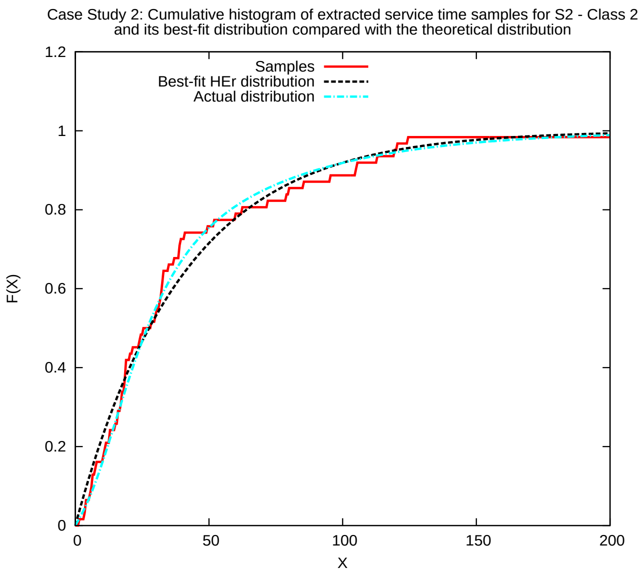


(f)

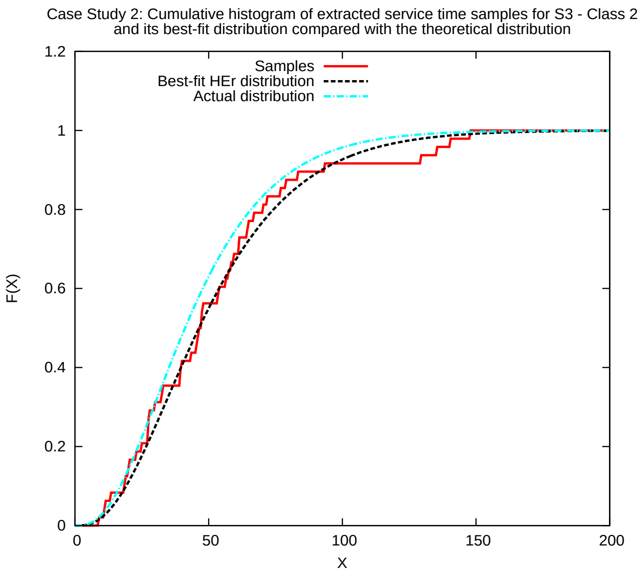
Figure B.1: Chapter 6, Case Study 2: Graphs B.1(a), B.1(b), B.1(c), B.1(d), B.1(e) and B.1(f) show the cumulative histogram of the extracted service time samples for customer class 1 and its best-fit hyper-Erlang distribution compared with the corresponding theoretical distribution for S1, S2, S3, S4 (exit from service cycle), S5 and S6 respectively.



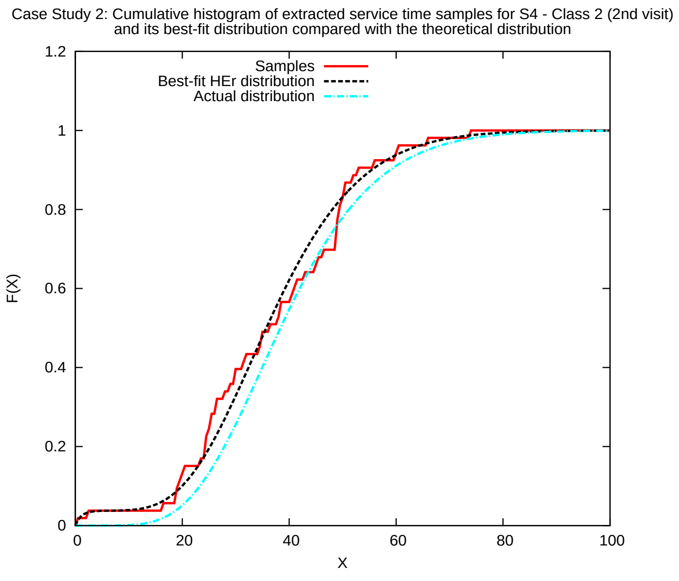
(a)



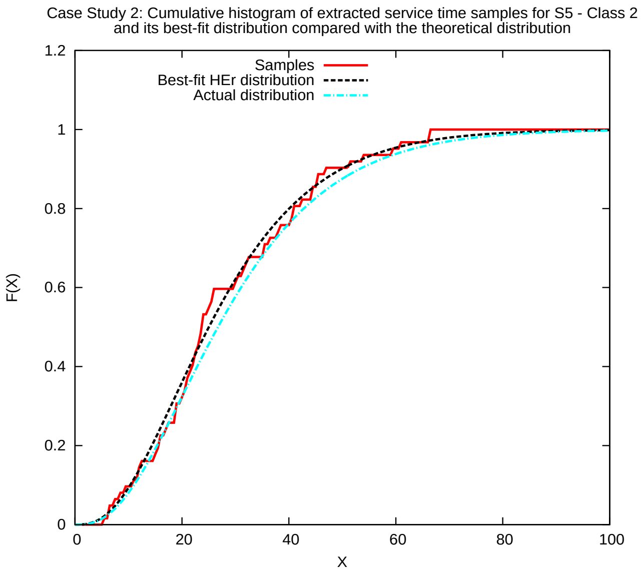
(b)



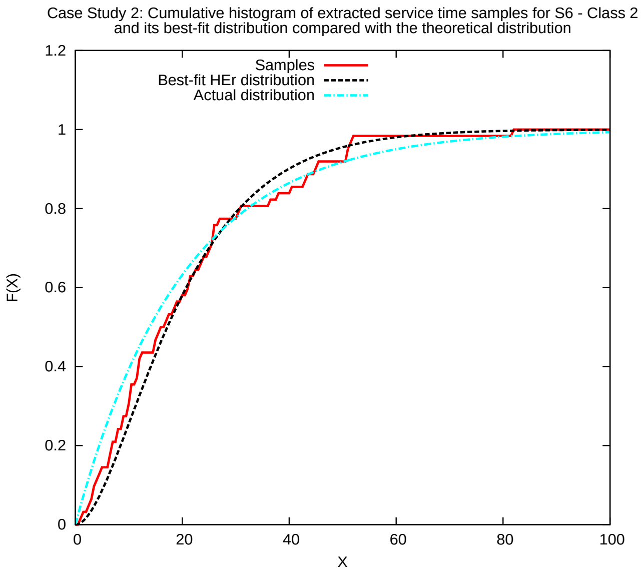
(c)



(d)

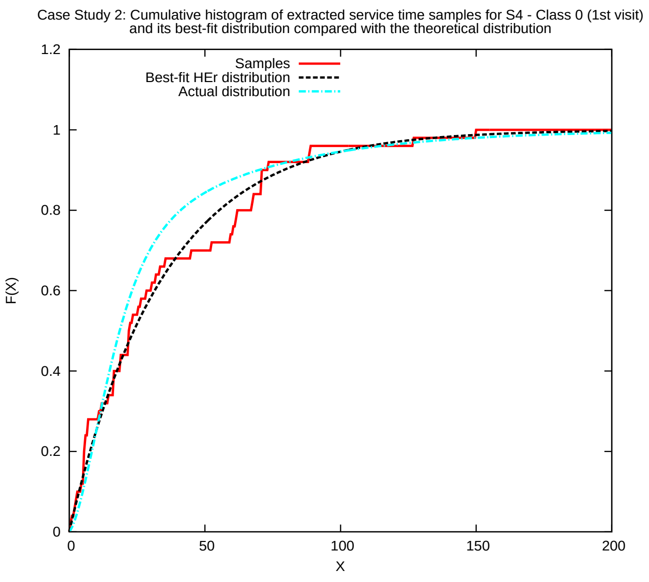


(e)

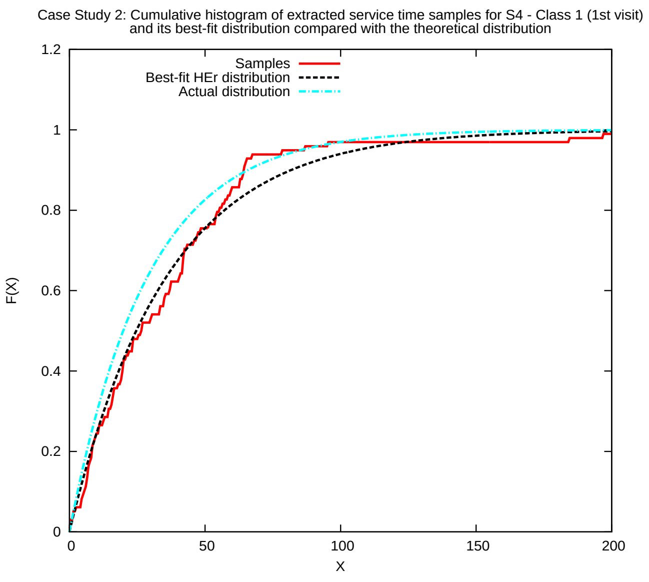


(f)

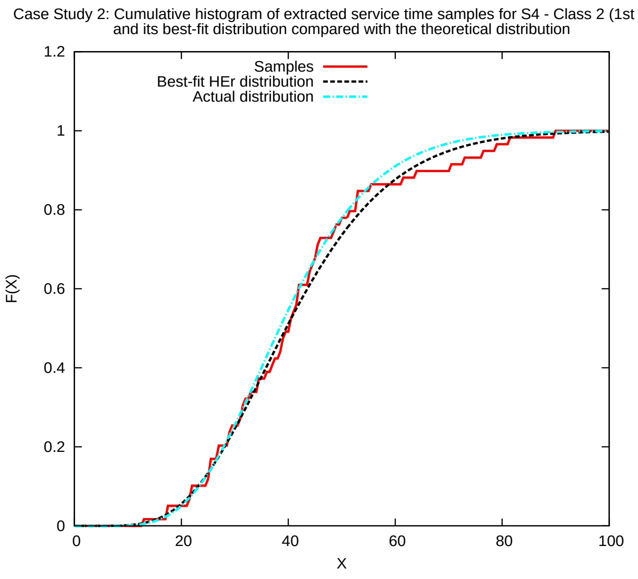
Figure B.2: Chapter 6, Case Study 2: Graphs B.2(a), B.2(b), B.2(c), B.2(d), B.2(e) and B.2(f) show the cumulative histogram of the extracted service time samples for customer class 2 and its best-fit hyper-Erlang distribution compared with the corresponding theoretical distribution for S1, S2, S3, S4 (exit from service cycle), S5 and S6 respectively.



(a)



(b)



(c)

Figure B.3: Chapter 6, Case Study 2: Graphs B.3(a), B.3(b) and B.3(c) show the cumulative histogram of the extracted service time samples for S4 (entry to service cycle) and its best-fit hyper-Erlang distribution compared with the corresponding theoretical distribution for customer classes 0, 1 and 2 respectively.

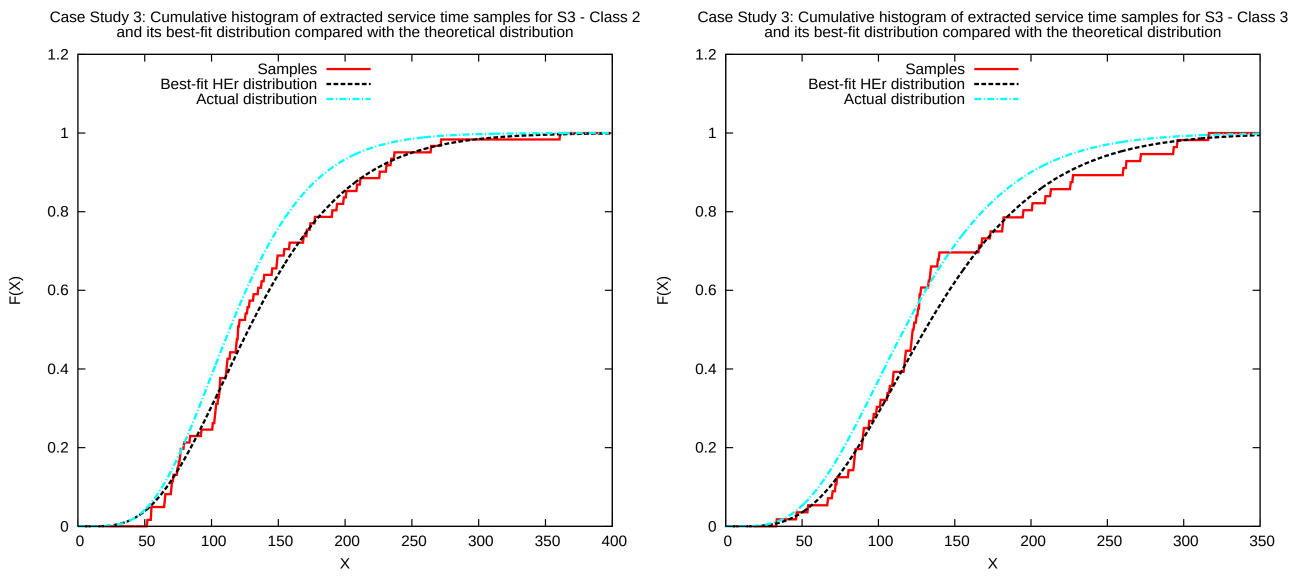
B.2 Case Study 3

		Service Time Density	Fitted HErD Parameters			Relative Entropy
			Phase Lengths	Rate (3 d.p.)	Weights (3 d.p.)	(3 d.p.)
Customer Class 0	S1 - (1st visit)	Exp(0.033)	1	0.034	1.000	0.001
	S1 - (2nd visit)	Exp(0.033)	1	0.034	1.000	0.001
	S2	Erlang$(2, 0.035)$	$2, 8$	$0.032, 5.686$	$0.985, 0.015$	0.017
	S4	Exp(0.033)	1	0.027	1.000	0.021
	S5	Exp(0.013)	1	0.013	1.000	0.000
Customer Class 2	S3	Erlang$(6, 0.05)$	5	0.037	1.000	0.049
Customer Class 3	S3	Erlang$(5, 0.04)$	5	0.036	1.000	0.030

Table B.3: The HErD parameters fitted by G-FIT for each service area's service time density with the relative entropy (in nat) between the theoretical and fitted probability density function for case study three of Chapter 6. Results are shown for the first (class 0), third (class 2) and fourth (class 3) customer class.

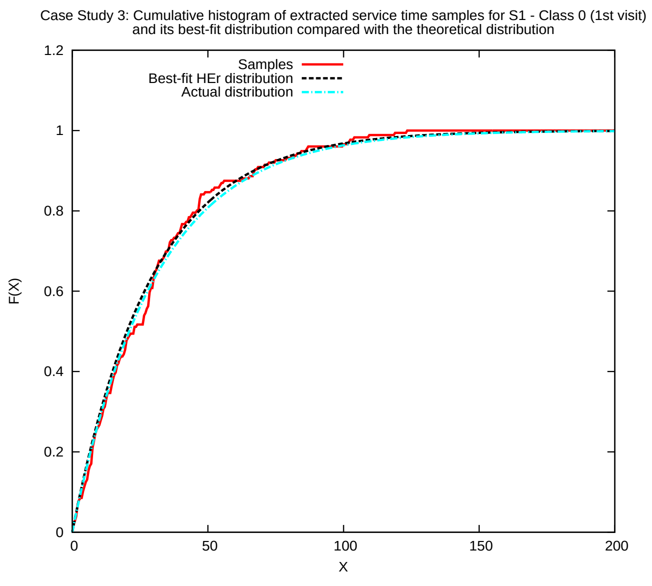
		Customer Class 0		Customer Class 2		Customer Class 3	
S1 - 1st Visit	Test Statistic	0.0741		N/A		N/A	
	α	0.1	0.05				
	Critical Values	0.0807	0.0920				
	Compatible ?	Yes	Yes				
S1 - 2nd Visit	Test Statistic	0.0798		N/A		N/A	
	α	0.1	0.05				
	Critical Values	0.0973	0.1109				
	Compatible ?	Yes	Yes				
S2	Test Statistic	0.0479		N/A		N/A	
	α	0.1	0.05				
	Critical Values	0.0809	0.0922				
	Compatible ?	Yes	Yes				
S3	Test Statistic	N/A		0.0708		0.1374	
	α			0.1	0.05	0.1	0.05
	Critical Values			0.1370	0.1562	0.1430	0.1630
	Compatible ?			Yes	Yes	Yes	Yes
S4	Test Statistic	0.1095		N/A		N/A	
	α	0.1	0.05				
	Critical Values	0.1244	0.1418				
	Compatible ?	Yes	Yes				
S5	Test Statistic	0.0928		N/A		N/A	
	α	0.1	0.05				
	Critical Values	0.1529	0.1743				
	Compatible ?	Yes	Yes				

Table B.4: Kolmogorov-Smirnov test at significance levels 0.1 and 0.05 applied to the extracted service time samples for each service point from the third case study of Chapter 6. The null hypothesis is that each extracted sample belongs to the corresponding best-fitted HErD.

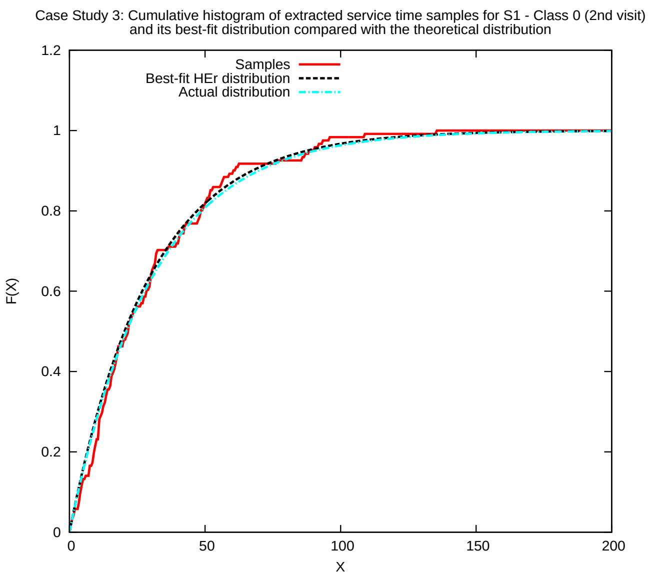


(a) (b)

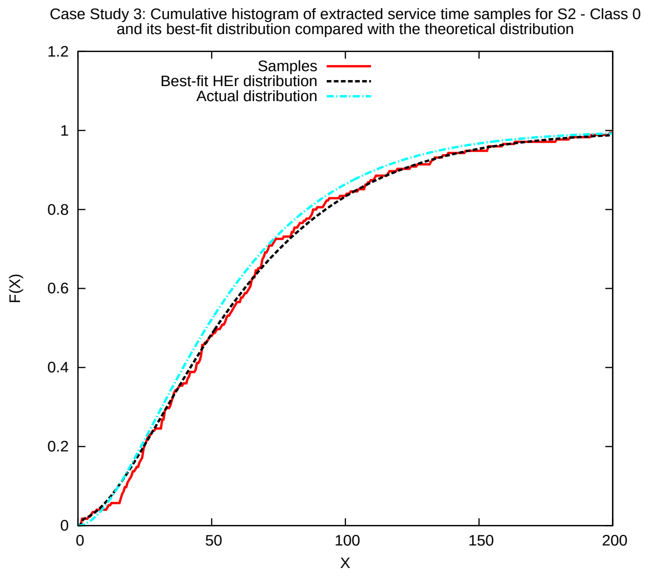
Figure B.4: Chapter 6, Case Study 3: These graphs show the cumulative histogram of the extracted service time samples for customer classes 2 (left) and 3 (right) and their best-fit hyper-Erlang distribution compared with the corresponding theoretical distribution for S3.



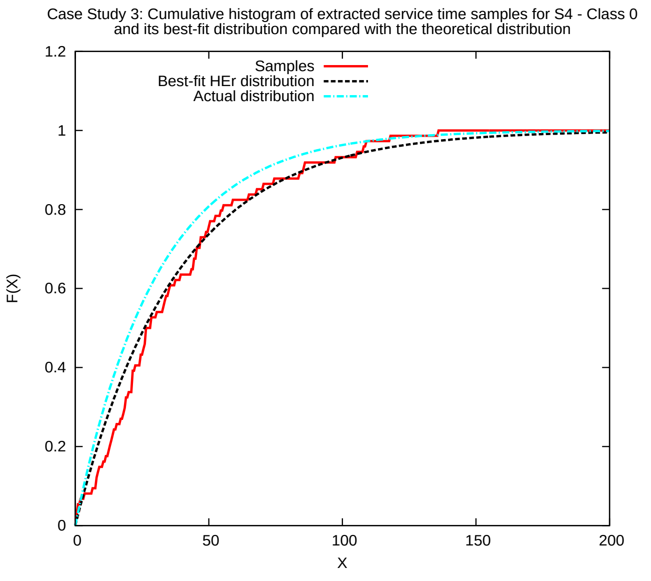
(a)



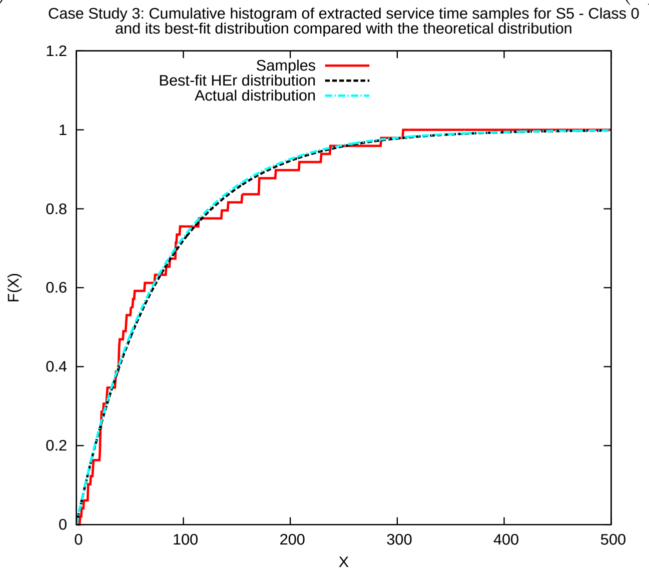
(b)



(c)



(d)



(e)

Figure B.5: Chapter 6, Case Study 3: Graphs B.5(a), B.5(b), B.5(c), B.5(d) and B.5(e) show the cumulative histogram of the extracted service time samples for customer class 0 and its best-fit hyper-Erlang distribution compared with the corresponding theoretical distribution for S1 (entry to service cycle), S1 (exit from service cycle), S2, S4 and S5 respectively.