Queue mining

combining process mining and queueing analysis to understand bottlenecks, to predict delays, and to suggest process improvements

de Smet, L.P.J.

Award date:
2016

Technische Universiteit **Eindhoven** University of Technology

Where innovation starts

Master thesis

Queue Mining: Combining Process Mining and Queueing Analysis to Understand Bottlenecks, to Predict Delays, and to Suggest Process Improvements

Author:
Luc de Smet
l.p.j.d.smet@student.tue.nl
luc.zero@gmail.com

Supervisor:
prof.dr.ir. W.M.P. van der Aalst
Technical advisor:
dr.ir. H.M.W. Verbeek

September 28, 2014

Abstract

Corporations such as insurance companies, postal companies and companies in the telecom world benefit from having a good understanding of their business processes. Questions that often get asked are: “How long will it take to complete this task?” and: “Will we finish this job before the next fiscal quarter?” in addition to: “What part of our system is making the process slow?” To answer these questions it is necessary to have a good understanding of what the bottlenecks in the process are and it is necessary to have the ability to predict how long tasks in a process will take. Luckily, many corporations store a large amount of digital data logs concerning historical process information, which can be utilized to answer these questions. This work introduces the basis for a novel approach based on process mining and queueing theory that focuses on using real-life historical data of processes to find the bottlenecks in such processes and make predictions about future instances of the processes. At the core of this approach are the day and week patterns of real life business processes, which are used for making high quality predictions. The techniques proposed have been implemented in ProM and were experimentally evaluated using both synthetic and real-life event logs to demonstrate its applicability. The predictor and bottleneck finding plugins are shown to be effective in both a lab and practical setting and have been published for the community to use.

Keywords: Process mining, Queueing theory, Business processes, Real-life data

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This master thesis, completed as part of the Computer Science and Engineering master at Eindhoven University of Technology, is created within the Architecture of Information Systems (AIS) group of the Mathematics and Computer Science department.

In this chapter the following topics are covered. In Section 1.1 the context in which this thesis is written is presented along with a motivation of why the presented new approach is useful. Then, the research goal of this work is presented in Section 1.2 and the scope of this research is given in Section 1.3. Next, related work is presented in Section 1.4 and finally the outline for the coming chapters is described in Section 1.5.

1.1 Thesis context

Corporations such as insurance companies, postal companies and companies in the telecom world benefit from having a good understanding of their business processes. Questions that often get asked are: "How long will it take to complete this task?" and: "Will we finish this job before the next fiscal quarter?" in addition to: "What part of our system is making the process slow?" The purpose of this work is to allow companies to use historical data to get an answer to these questions. Often, business processes are supported by information systems such as Workflow Management Systems or Enterprise Resource Planning systems. These systems allow for the collection and storage of a large amount of detailed historical data concerning the tasks completed in business processes. The proposed solution should analyze this data with the purpose of gaining insight into the reality of business processes and the capability of predicting future events. Both can help managers and analysts to get a better view of their process and provide operational support, i.e.: to give supervisors of business processes insight in those processes to help decision making.

A new emerging type of data analysis techniques known as process mining [37] has been proven very useful by offering new insights into business processes. Classical approaches in process mining have already been successful at making predictions for and giving insight in processes, but have limited potential. For that reason, a new approach is taken within the area, which is inspired by queueing theory. Work in business processes often has queue-like behavior and for that reason it makes sense to explore the possibilities extracting queue-based models from event logs and using them as a means to provide operational support. This leads to the research goal of this thesis: *How to combine process mining and queueing theory to gain insights in the bottlenecks of a business process and provide operational support.*

1.2 Research goal

Operational support using process mining as a basis has become a popular topic [37, 39, 38]. A number of approaches that predict the future for running cases already exist. Most process mining based techniques perform well in finding patterns in processes and other data, but often ignore queueing behavior of humans in real life settings. On the other hand, queueing theory based techniques perform well with respect to human behaviour, but do often take a very general approach over all cases and do not consider fluctuations in the process or different contexts. The purpose of this thesis is to extend these approaches with a novel one: using the strengths of both queueing theory and process mining to provide operational support in a way that both real life human behavior and real life patterns are taken advantage of. Patterns depending on relative time and context information will be used to attempt improving previous results.

Operational support can be provided in a multitude of ways, e.g.: identifying bottlenecks, predicting completion times and guiding or suggesting actions given some process state. This approach focuses on identifying bottlenecks and predicting *event* and *case* completion times. The approach to solve this problem is fitting queues on the most important bottlenecks of processes. These queues can then be used to provide a more realistic view of how resources work, depending on their context. For this approach the input is assumed to be an *event log*, i.e. a log of a number of cases which contain a number of events executed within some process. The research question of this thesis is as follows:

Research Question: ***Given an event log of a process, how can one combine queueing theory and process mining to gain insights in the bottlenecks of the process and provide operational support for the process?***

This research question is split up into multiple smaller questions. The first goal is to find relevant queues, given an event log. A queue is defined by resources and activities. The queues with the biggest delays are the ones most interesting for analysis. Hence, the following sub-question arises:

Sub-Question 1.2.1 *Given an event log of a process, how can one find the most relevant queues within this log, i.e.: the queues that cause the biggest delays in the process, in an automated way?*

In addition to finding the main bottlenecks, it is necessary to find the characteristics of their corresponding queues, e.g.: arrival rate, service rate, which resources can serve this queue and how many actually serve it. This leads to the following sub-question:

Sub-Question 1.2.2 *Given a bottleneck in a process, what are the essential characteristics of the corresponding queue and how can these characteristics be derived in an automated way?*

Then, given these bottlenecks and their characteristics, the question remains how to use this information. Useful things to capture from these characteristics would be: Understanding the bottlenecks better, predicting where delays and problems will occur given a process state and suggesting improvements given a current process state. The final sub-question captures this:

Sub-Question 1.2.3 *Given the characteristics of bottlenecks, how can one understand these bottlenecks better, predict delays and problems given a current process state and suggest improvements given a current process state?*

Throughout this thesis these subquestions will be answered in detail. The next subsection will focus on the scope in which these questions will be answered.

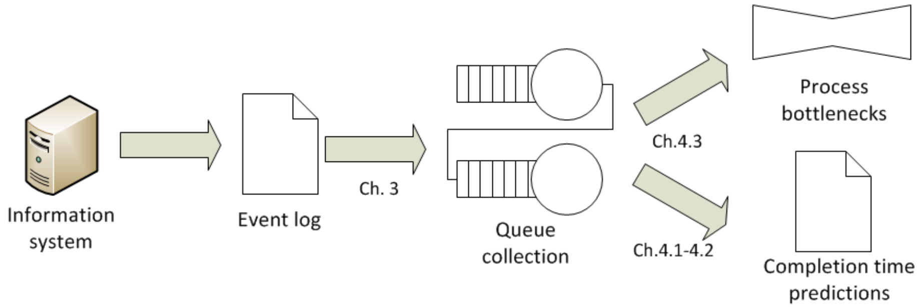


Figure 1.1: Conceptual model of the global approach.

Queue	Resources	Activities	Bottleneck factor
Queue 3	Henk, Anita	Sort package	0.99
Queue 1	Nick, Katie	Send package	0.71
Queue 2	Edward	Inspect packages	0.60

Table 1.1: Example bottleneck identification results.

1.3 Research scope

To provide an answer for the research question stated in the previous subsection, an approach is needed that allows an analyst to use an event log to create a *queue collection*, i.e.: A collection of queues describing the process obtained by analyzing an event log. Then, an approach is needed to use this queue collection to find bottlenecks in the process and to do predictions on future data.

In Figure 1.1 a high level overview of the suggested approach is shown. In practice, information systems generate event logs of business processes, for example the process of delivering packages of a postal company. A method needs to be devised to automatically derive a queue collection from an event log, given a limited amount of user input. The method should be constructed in such a way that an analyst can easily perform the conversion with only knowledge of the business process and no need for extended knowledge of the techniques used. An example of a queue in such a network is the department in a postal company that handles distribution of large packages to different transport vehicles. Solving this problem provides an answer for Research Sub-Question 1.2.2 and will be discussed in Chapter 3.

Once a queue collection is obtained, it should be possible for an analyst to do two things. First, the queues in the process, i.e. the clusters of resources and activities in some information system, need to be analyzed such that the bottleneck queues can be identified, answering Research Sub-Question 1.2.1. This should be done automatically and give an analyst insight in what the queues are that produce the biggest delays in a process. For the example postal company a bottleneck queue could be the department that has to process packages with unreadable addresses, since it requires a lot of manual labor. An example result would be a ranking of bottlenecks over a set of queues, as shown in Table 1.1. In this example, the queues are ranked on how much they are a bottleneck for the system, Queue 3 being the biggest bottleneck. This problem and a possible solution are discussed in detail in Section 4.3.

Secondly, it should be possible to do predictions based on a queue collection. I.e. it should be possible to automatically predict how long an event (discussed in Section 4.1) or case (discussed in Section 4.2) is going to take, given some information about that activity and the queue collection. Operational support will then be provided in the form of a number of predicted values. An example prediction output is shown in Table 1.2. In the case of a postal company, providing the user with a good estimate of when their package will arrive is an application of these predictions. Solving these two problems provides an answer to Research Sub-Question 1.2.3.

Case ID	Arrival time	Predicted exit time
Case 1	25-09-2014, 13:53	29-09-2014, 11:23
Case 2	27-09-2014, 08:59	29-09-2014, 13:25
Case 3	30-09-2014, 17:30	05-10-2014, 10:11

Table 1.2: Example case duration prediction results.

1.4 Related work

In this section related work will be covered, starting with other process mining techniques. Then, other work in the main areas of this thesis, namely operational support, time predictions and performance analysis are discussed.

Process mining is a field that has been studied since the mid nineties by a number of groups. Some of the initial research is summarized in *Workflow mining: A survey of issues and approaches* [43] by Van der Aalst et al. Early process mining techniques have been developed to support multiple fields, including Workflow Management systems (WfMS) [1], business process models [10] and software engineering processes [8]. More profound techniques were proposed by Herbst [18].

In the last few years much research has been done on different novel process mining techniques. While mining for a workflow pattern was introduced as early as 2004 by Van der Aalst et al. [38], many other successful techniques have surfaced since then. These include, but are not limited to the *Heuristic miner* [50] and *Flexible Heuristic miner* [49] bij Weijters et al., the *Fuzzy miner* [17] by Günther et al., a genetic process mining approach by Medeiros et al. [29], a process mining technique based in linear programming by Van der Werf et al. [45] and the *Inductive Miner* by Leemans et al.[27]. A number of summarizing works and applications have also surfaced, for example Discovery, Conformance and Enhancement of Business Processes [39], the *Process mining manifesto* [37] by Van der Aalst et al. and a transition and region based solution by Van der Aalst et al.[40]

The concept of queue mining has been approached from a different angle by Senderovich et al. [36] around the same time as this thesis was developed. This approach shows much promise on well-formed queue data and focuses more on queueing theory than the approach proposed in this work. It does not, however, cover a general approach to deal with real life business process logs. The approach of Senderovich et al. uses the assumption of knowing *Effective Process Times* (EPT's) or them being deducible in some way, to allow for clear measurements on how long activities take waiting and being executed. More work on EPT's is provided by Jansen et al. [23] and Etman et al. [13]. In addition, Veeger et al.[47] introduced a queue and effective process time in generating cycle time-throughput curves in a semiconductor fabrication setting, Jacobs et al.[21] introduce a technique for finding operational time variability using EPT's and Jacobs et al.[22] present a paper on quantifying the variability of batching equipment using EPT's. Even though it would be very beneficial to have knowledge of EPT's, the approach proposed in this work assumes that EPT's are not automatically deducible from a general event log.

The most important practical goal of this approach is to provide operational support for business processes. Operational support has been provided by a multitude of approaches and angles, this thesis included. A paper by Van der Aalst et al. [44] discusses how to provide assistance, rather than guidance, in a case handling system, while a paper by Wynn et al. [53] discusses simulating of business processes from some intermediate state, rather than from scratch as a means of operational support. In addition, Rozinat et al. [34] have discussed using past and current state information of some process to predict near-future scenario's. Practical applications have been shown as well, one example being discussed in the paper by Bana et al. [3], which covers operational support in improving the process at a textile plant in Brazil.

This thesis covers doing time predictions based on a queue collection, yet other techniques in the area of time predictions have been proposed. Van der Aalst et al. [41] discusses predicting the completion times of running instances of a process, taking into account historical and present data. Chen et al. [6] elaborates on how to use a fuzzy model to make completion time predictions for the practical application of wafer fabrication. Chien et al. [7] introduce another approach that uses historical and present time data to predict online travel times based on traffic. In addition, a number of master theses have been written on the topic of time predictions, including the work by Crooy [9] and the work by Schellekens [35]. In addition, work by Pika et al. [32] has shown how to predict deadline transgressions using event logs.

One of the main strengths of this work is using time and context patterns from real life to improve predictions. Other work in this area includes a paper by Eren et al.[12] which focuses on providing running case predictions based on contextual information.

Performance analysis is a topic touched upon by doing bottleneck analysis and using it to find which part of the process is the slowest. Performance analysis has been shown to be effective in other contexts. Bolch et al. [4] propose an approach that uses queue collections for predicting system performance before a system is put in operation. Wetzstein et al. [52] discuss a framework for monitoring performance in *Web Services Business Process Execution Language* (WS-BPEL) specified processes using machine learning.

1.5 Outline

In Chapter 2 the preliminaries are presented, most importantly the basics of process mining and queueing theory used as a basis for the approach presented in this thesis. Next, Chapter 3 discusses the methods for finding a queue collection that can serve as a predictor, which holds into account the patterns of real life activities. The techniques used for predicting the completion times of individual activities and complete traces are discussed in Chapter 4. Details on the realization of the techniques mentioned in the two previous chapters are discussed in Chapter 5, after which the experimental evaluation of those techniques and corresponding results are discussed in Chapter 6. Finally, this thesis is wrapped up in Chapter 7 with a summary and the strengths and limitations of the proposed approach in addition to a section describing possible future work in queue mining.

Chapter 2

Preliminaries

Within this section, preliminary notions and concepts that are used for both conceptual techniques and realization are explained. First, some general notation is introduced in Section 2.1. Then, in Section 2.2 a short introduction in the area of queueing theory is given. Furthermore, the notion of process mining is introduced in Section 2.3 and an in depth explanation is given on what an event log is and what it is used for in Section 2.4. Finally, a number of tools used for the realization of the approaches proposed in this thesis are introduced in Section 2.5.

2.1 Notation

To support the approaches presented in this work, the following notation is used:

- A multiset of items $S = [a^2, b, c^3, ...]$ represents a set of items in which duplicates are allowed. In this example the item a occurs twice, while c occurs three times.
- $|S|$ for some set S indicates the number of elements in S.
- $s \in S$ represents an element s from set S.
- $L[i]$ for some (ordered) list L indicates the element at index i of the list L.
- $S \parallel S'$ for some strings S and S' indicates the concatenation of these strings.
- $max(S)$ denotes the maximum of values over some set S with comparable elements.

2.2 Queueing theory

Next to process mining, *queueing theory* is one of the areas that inspired the techniques presented in this work. Queueing theory [15] is a theory first introduced by Agner Erlang [5] as early as 1909. The theory has been applied in many fields, including telecommunication and computer science [30] and has been used for the design of many kinds of businesses, including factories and hospitals [28]. Queueing systems have been a popular topic of research both for universities and industry [2]. Queueing theory is often tied to operations research because one of queueing theory's main applications in practice is providing operational support for business decisions.

The most basic form of a queue is illustrated in Figure 2.1. Items arrive at the queue on the left side with a mean arrival rate, i.e. the average amount of items arriving within some time interval, which is denoted by λ. These then go into the queue which can be of any finite length, for example the input conveyer belt of a sorting machine used for postal packages. Alternatively,

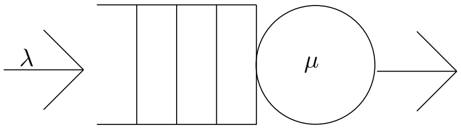


Figure 2.1: Basic queue concept model.

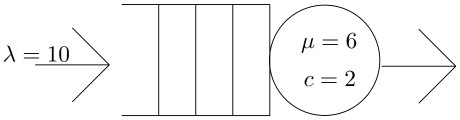


Figure 2.2: Basic queue example with steady state.

the queue size can be considered of infinite length. If an item arrives at a full queue, it will be dropped. Once an activity works its way to the front of the queue, it gets removed from the queue and some work will be done at a work station. The mean completion rate for this work, i.e. the the average amount of items handled within some time interval, is denoted by μ. Items that are being worked on are taken out of the queue, making room for new items. Items that are complete leave the work station and are done.

Now a basic example will be shown of a single queue, for which the following assumptions hold: there is no fluctuation in completion rate and arrival rate and there are no other external factors influencing the system. Figure 2.2 shows such a queue which has a steady state system, i.e.: the sojourn time for items stays the same over time. For such a steady state system, one can analytically calculate a number of characteristics of this queue. The arrival rate $\lambda = 10$, the service rate $\mu = 6$ and the amount of servers $c = 2$. The system utilization is defined as the arrival rate divided by the total service rate and is $(c \cdot \mu)/\lambda = 10/(2 \cdot 6) = 0.833$. The average amount of time a work item spends in the system is defined by $W = 1/(c \cdot \mu - \lambda) = 1/(2 \cdot 6 - 10) = 0.5$. The average amount of work items in the system is defined by Little's law as: $L = \lambda \cdot W = 10 \cdot 0.5 = 5$. The average amount of time spent by a work item in the queue is defined by $W_q = \lambda/(c \cdot \mu)(c \cdot \mu - \lambda) = 10/(12)(12 - 10) = 10/24 = 0.416$. The average amount of work items in the queue is defined as $\lambda * W_q = 10 * 0.416 = 4.16$. Now consider the scenario with the same queue, but $c = 1$. The system utilization is then equal to $10/6 = 1.67 > 1$, which implies this system is broken, i.e.: the queue will grow infinitely large and the work will never all be done. These calculations, however, do not work on more complex real life models that are influenced by a large number of external factors and often have no short term stability.

There are many aspects that are not covered in this simple model. For example, it is not made explicit how items are taken from the queue, also known as the queueing discipline. Example queueing disciplines are First In First Out (FIFO), Last In First Out (LIFO) or some scheme where items with a higher priority are chosen first (For example, gold customers go before regular customers). Another aspect is the amount of servers that help complete items from the queue. There could be any number from one until an infinite number of servers for a queueing model.

The many different possible configurations of a queue have lead to a a standard notation to describe a part of the configurations, called Kendall's notation [25]. Kendall's notation works as follows: using the form A/S/c, A denotes the arrival process, which can for example conform to a Poisson distribution. The service process is denoted by S, which indicates the service time distribution, for example defined by a Normal distribution or a Poisson distribution. The c denotes the amount of servers working in the queue. An example notation would be M/M/1, where both M's denotes a Markovian chain Poisson process. Arrival and service times are negative exponentially distributed, the number of times that activities arrive and that they are serviced in some period are Poisson distributed. The amount of servers is one.

There also exists an extended version of Kendall's notation including the queue capacity, queueing discipline and amount of items to be handled. The extended notation is denoted as A/S/c/K/N/D, where A, S and c are the same as in the previous notation. The queue capacity, i.e.: the amount of items that can be in the queue before others are dropped, is indicated by K. The total amount of items that has to be handled is denoted by N, which can be any positive number. Finally, D indicates the queueing discipline, for example FIFO or LIFO.

A set of queues that form a bigger system is considered a queueing network. Items enter a queueing network and then enter and leave a number of queues according to the queue specifications and process flow, until the entire process for such items is completed. Different kinds of queueing network have been proposed, for example Jackson networks [20] and G-Networks [14].

The typical assumptions stated for the simple queue example and often also made for more complex queues are not valid in real life and as such it is not possible to analytically solve problems this way. First, the assumption of having a constant arrival rate is false. Consider looking at a normal workweek for a company where employees work from 9:00 to 17:00, the arrival rate at 10:00 will be vastly different from that at 03:00. Similarly, the service rate will be unstable over the day and week. For example, at a Sunday, the service rate will be a lot lower than at Tuesday morning. Finally, the assumption of a steady state is also unsafe. When running a postal office, the queue of work around the Christmas periods will most probably build up as well as the average sojourn times for deliveries, while afterwards these will go down again.

In addition to these assumptions, factors such as the context in which a process is or the availability of fluctuations in how hard servers work or how much servers are present at all, make it impractical to use a classical queueing model. Because the standard assumptions for queues do not hold in this approach, a queue collection consisting of G/G/inf queues (Kendall's notation) is extracted, where G indicates any distribution.

2.3 Process mining

The techniques described in this thesis are inspired by queueing theory and *process mining*. The latter will be introduced in this section. Process mining is, in short, "a set of techniques to discover, monitor and improve real processes (i.e., not assumed processes) by extracting knowledge from event logs" [37].

Process mining is part of a larger scope, as seen in Figure 2.3. Information systems are, as noted before, used to support and control a real life process. The actions performed in real life that pass through the information system are logged as historical and present data in event logs, also denoted as *provenance*. Event logs are the basis of process mining and serve as a means for creating models describing the real life situation, denoted as *cartography*. In addition, models created using event logs can be used to gain insight in the log, also denoted as *navigation*. Finally, one could compare man-made models (*de juro*) and models crated by some cartography technique (*de facto*) to understand the difference between the designed model and the actual situation. The methods proposed in this thesis fit in the cartography and navigation parts of the process mining scope, or more precise: they fit into discover, predict and explore.

2.4 Event log

Business processes are often supported by information systems. These information systems most likely provide logging functionality that produces *event logs*. An explanation of the context of event logs is given in Section 2.3. This section will focus on introducing the abstract and formal basics of what an event log is and how they can be used in process mining and queue mining.

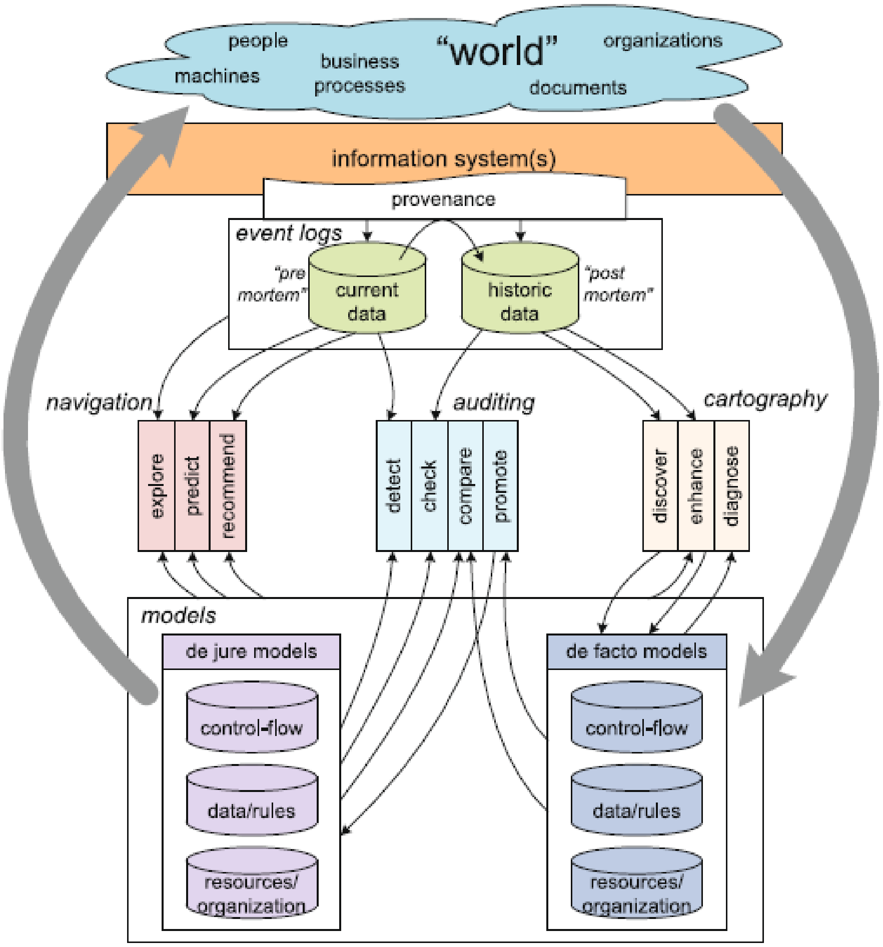


Figure 2.3: Process mining scope.

Activities executed inside a process are the basis for process mining techniques, as they represent an activity being done within a business process. As such, it is necessary to give an exact definition of what an activity is.

Definition 2.4.1 ***Event, Attribute*** *Let $\mathcal{E}$ be the* event universe*, i.e., the set of all possible event identifiers. Events may be characterized by various attributes, e.g., an event may have a timestamp, correspond to an activity, is executed by a particular person, has associated costs, etc. Let AN be a set of attribute names. For any event $e \in \mathcal{E}$ and name $n \in AN$: $ATT_n(e)$ is the value of attribute n for event e. If event e does not have an attribute named n, then $ATT_n(e) = \perp$ (null value).*

For convenience, the following standard attributes are assumed:

- $ATT_{type}(e)$ is the activity associated to event e.
- $ATT_{time}(e)$ is the timestamp of event e.
- $ATT_{res}(e)$ is the resource of event e.
- $ATT_{context}(e)$ is the context in which event e exists.

An example of an event would be as follows: Say that for an event with activity $ATT_{type}(e) =$"Send package" we have the following additional attributes: the timestamp $ATT_{time}(Sendpackage) =$ the 26th of August 2014 at 22:35 and the resource $ATT_{res}(Sendpackage) =$ "Henk". That means that event describes a package being sent on the 26th of August 2014 at 22:35 by a resource named Henk. The context of sending this package is left unknown, i.e.: $ATT_{context}(Sendpackage) = \perp$. In addition to formalizing a single activity, it is also preferable to have some notion of a *case* and an entire *event log* of activities, i.e.: a single execution of a process.

Definition 2.4.2 ***(Case, trace, event log)*** *Let $\mathcal{C}$ be the* case universe*, i.e., the set of all possible case identifiers and let AN be a set of attribute names. Cases, like events, have attributes. For any case $c \in \mathcal{C}$ and name $n \in AN$: $ATT_n(c)$ is the value of attribute n for case c ($ATT_n(c) = \perp$ if case c has no attribute named n). Each case has a special mandatory attribute trace: $ATT_{trace}(c) \in \mathcal{E}^*$. It is assumed $ATT_{trace}(c) \neq \langle\rangle$, i.e., traces in a log contain at least one event.* trace *is a finite sequence of events $\sigma \in \mathcal{E}^*$ such that each event appears only once, i.e., for $1 \leq i < j \leq |\sigma| : \sigma(i) \neq \sigma(j)$. An* event log *is a set of cases $L \subseteq \mathcal{C}$ such that each event appears at most once in the entire log, i.e., for any $c_1, c_2 \in L$ such that $c_1 \neq c_2$: $\partial_{set}(ATT_{trace}(c_1)) \cap \partial_{set}(ATT_{trace}(c_2)) = \emptyset$. If an event log contains timestamps, then the ordering in a trace should respect these timestamps, i.e., for any $c \in L$, i and j such that $1 \leq i < j \leq |ATT_{trace}(c)| : ATT_{time}(ATT_{trace}(c(i))) \leq ATT_{time}(ATT_{trace}(c(j)))$.*

So a case consists of a trace and some attributes, where the trace is a sequence of at least one event which described a single process execution Say, for example, that the case with identifier "HandlePackage" has a trace consisting of two activities, the first being the activity introduced as an example for activity after Definition 2.4.1 and the second being as follows: The *type* would be "Sort package", the *time* would be the 26th of August 2014 at 22:10 and the *resource* would be "Anita". The trace would then consist of the activities "Sort package" and "Send package" in that order, since they have to be ordered on timestamp non-decreasingly. Additionally, it is possible for that trace to have attributes, such as a department code: $ATT_{dept\_code}(HandlePackage) = B100$. The set of all encountered process executions, also known as traces, is the event log. An example of an event log, based on the examples in the definition of activities and traces is provided in Table 2.1.

Trace id	Activity	Resource	Timestamp	...
12324	Receive package	Henk	14-02-2014 13:45	...
12324	Sort package	Henk	20-02-2014 19:42	...
12324	Repackage	Anita	20-02-2014 19:55	...
12324	Send package	Anita	21-02-2014 14:03	...
12326	Receive package	Henk	17-11-2014 15:12	...
12326	Sort package	Anita	19-11-2014 17:22	...
12326	Send package	Anita	21-11-2014 15:12	...
...	...	...	...	...

Table 2.1: Example event log.

2.5 Supporting tools

In this section the tools supporting the research, development and experimental evaluation of this thesis are touched upon. The tools and technologies used are ProM (Subsection 2.5.1), the XES format (Subsection 2.5.2) and CPN Tools (Subsection 2.5.3).

2.5.1 ProM

The *Process Mining framework* ProM [42, 46] is used for the realization of the approaches proposed in this thesis. This subsection will touch upon what ProM is and how it is used. ProM is a generic open-source framework for implementing process mining tools in a standard environment. The framework provides an easy access for researchers and analysts to many process mining oriented techniques and helpful tools. ProM is easily extendable and allows researchers to implement new techniques and share them. The current framework already offers a great variety of model types, such as Petri Nets or Declare models and plugins such as the Alpha algorithm or the Fuzzy miner. Additionally it provides convenient general functions such as importing, exporting and visualizing data files. ProM is also able to handle the XES Event Log format (see next subsection) which will be used as a basis for event logs. ProM 6.3 ("Salt") was used for realizing process mining techniques, importing event logs and exporting and visualizing results.

2.5.2 XES event log format

An important part of ProM is the ability to import and use the *XES Event Log* format[1] and the ability to use it in a large number of plugins. The XES format has been selected as the standard format for event logs by the IEEE Task Force on process mining. The XES format is argued to be an improved version of the XMXL format, as discussed by Verbeek et al. [48]. The plugins designed for this thesis use the XES format for its input files. Figure 2.4 shows the metamodel describing the XES standard. The metamodel is very similar to the formal model of a log given in Subsection 2.4 and Definition 2.4.2: Events in the formal description are comparable to XEvent's in the XES standard with Attributes corresponding to XAttributables. Similarly, cases from the formal description correspond to XTrace's in the XES standard. Finally, an event log corresponds to an XLog. When, at any point, an event log is mentioned in this work, in the realization an XLog with all its components was used.

In addition to the XES metamodel, the extension interface is also found in Figure 2.4. An XExtension allows for introducing additional standardized attributes in the XES framework. XExtensions are defined using a xesext file and are stored at some URI location, where it can be referenced from within ProM.

[1] http://www.xes-standard.org/

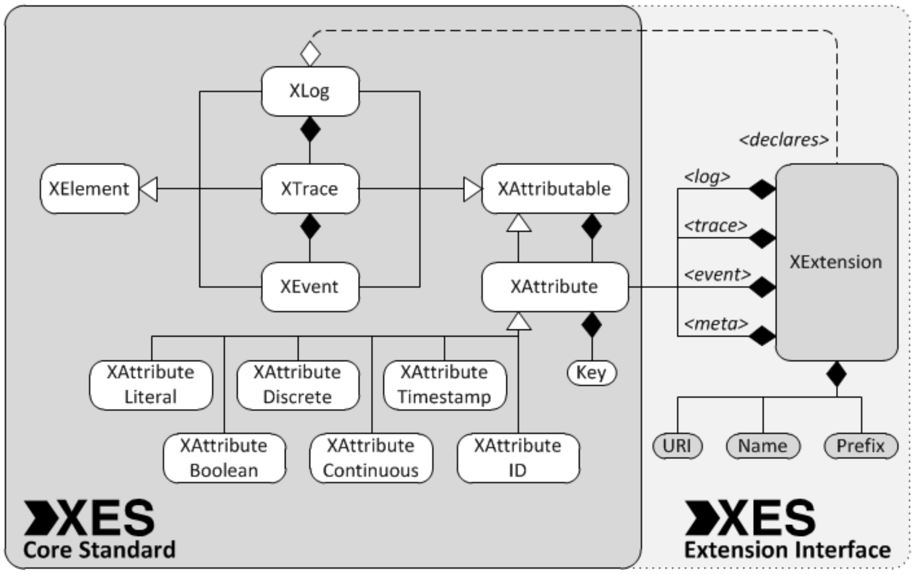


Figure 2.4: XES metamodel

2.5.3 Synthetic log generation tools: CPN Tools and Prom Import Framework

CPN Tools [24, 33] is a tool for creating (Colored) Petri-nets, which can be used for simulation and log generation. In this thesis, CPN Tools is used to create Petri-nets that produce synthetic logs, which are useful for doing several experiments with respect to validating and verifying the techniques proposed in this thesis. CPN Tools offers an extensive set of tools to create and manipulate Petri-nets and allows for simulation of fictional business processes. CPN Tools has already been shown that it is fit for creating test logs, for example by De Medeiros et al. [11] and also for doing performance analysis as shown by Wells [51]. All synthetic logs used for testing have been generated using CPN Tools 4.0.0. Logs created by CPN Tools are being pre-processed to be used in ProM with help of the ProM Import Framework [16]. The ProM Import Framework is a tool designed to convert log data from a large number of sources into data that can be handled by ProM. For the realization phase of this thesis ProM Import Framework 7.0 (Propeller) was used.

Chapter 3

Finding a queue collection based on an event log

The main goal of this thesis is to gain insight in the bottlenecks of a (business) process and provide operational support for that process. To achieve this, the strengths of queueing theory and process mining will be combined by generating a *queue collection* in which each queue should accurately simulate a real life queue that is susceptible to changing contexts and properties. The first step in achieving this goal is to find a queue definition that describes the process to be analyzed. The input process is assumed to be described in the form of an event log such as given in Table 2.1. It is assumed that for this event log L holds that $\forall c \in L : [\forall e \in ATT_{trace}(c) : ATT_{res}(e) \neq \perp \wedge ATT_{time}(e) \neq \perp \wedge ATT_{type}(e) \neq \perp]$, i.e.: all events have a non-empty value for the activity, resource and a timestamp attribute. The wanted output is a queue collection that accurately describes the process described by the event log and in that way provides an answer to Research Sub-Question 1.2.2.

The method for finding a collection of real life queues builds upon the principle of patterns found in real life logs. The first kind of pattern is apparent when looking at an example as shown in Figure 3.2. This figure shows the amount of activity starts for every hour over the course of three weeks, starting on a Saturday. It can be observed that there is a clear day and week pattern in the amount of activity starts, which leads to the intuition that different parts of the week have different arrival and service rates. In addition, consider Figure 3.3 which shows a clear pattern that is not due to the influence of a work week. This leads to the intuition that in different contexts there is a different arrival and service rate, which should be accounted for. The innovation

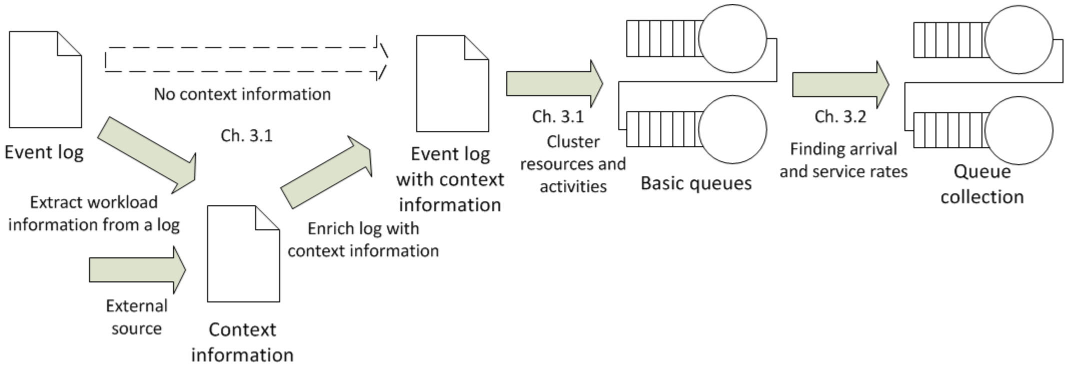

No context information
Event log
Ch. 3.1
Extract workload information from a log
External source
Context information
Enrich log with context information
Event log with context information
Ch. 3.1
Cluster resources and activities
Basic queues
Ch. 3.2
Finding arrival and service rates
Queue collection

Figure 3.1: Conceptual model of extracting queues from event log.

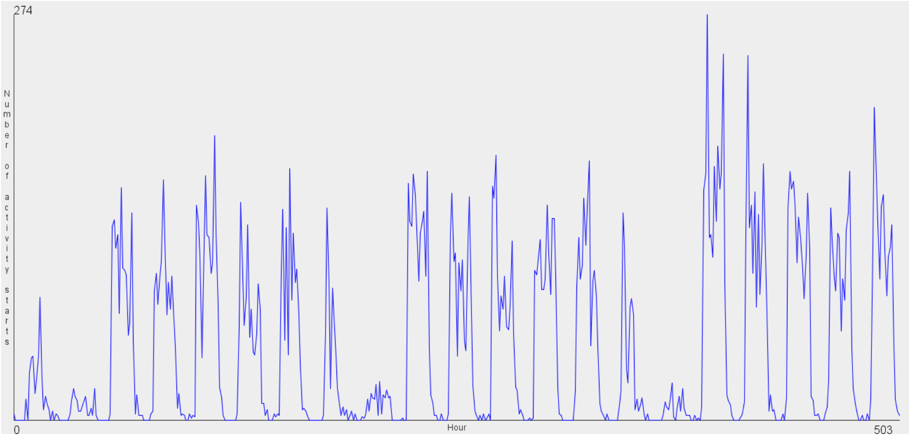


Figure 3.2: Example log showing clear week patterns in activity starting count. The pattern has a granularity of one hour, concerns three weeks and starts on a Saturday.

proposed to achieve this, is splitting up arrival rate and service rate data based on the relative time they arrive and the context at the moment of arrival.

A conceptual model of the approach is found in Figure 3.1. To allow splitting up data using context information, one must be able to enrich an event log with context information. As it is necessary to have context information to run any experiments on the effects of using context information, some kind of context information had to be acquired. The context chosen for this purpose is the workload in the system at some point in time, since it seems like an important factor in real life: if it is a busy week at work, things are different than in a regular week. As such, a technique was developed to extract workload context information from an event log. Both extracting context information and enriching a log with context information is covered in Section 3.1. In addition, this section describes how to cluster activities and resources such that each cluster correctly represents a queue and work station in the real process. The output of this method is a set of basic queues.

Then, Section 3.2 covers how to use context information and temporal information to enrich basic queues with important queue characteristics. The characteristics deemed most important are the service and arrival rates. So, the input of this method is a set of basic queues and the output is a queue collection.

3.1 Finding basic queues and handling context information

This section discusses, as a part of the conceptual approach shown in Figure 3.1, two separate subjects. First, there is the matter of obtaining contextual information from an event log and adding contextual information to an event log, which is covered in Subsection 3.1.1. Then, enriching a log with context information from any source is covered in Subsection 3.1.2. In the realization, these actions need to be performed as a preprocessing step, before making basic queues. Then, the conceptual technique on how to find a set of basic queues, given an event log, is presented in Subsection 3.1.3. The resulting set of basic queues is used in the techniques described in Section 3.2 to obtain a queue collection.

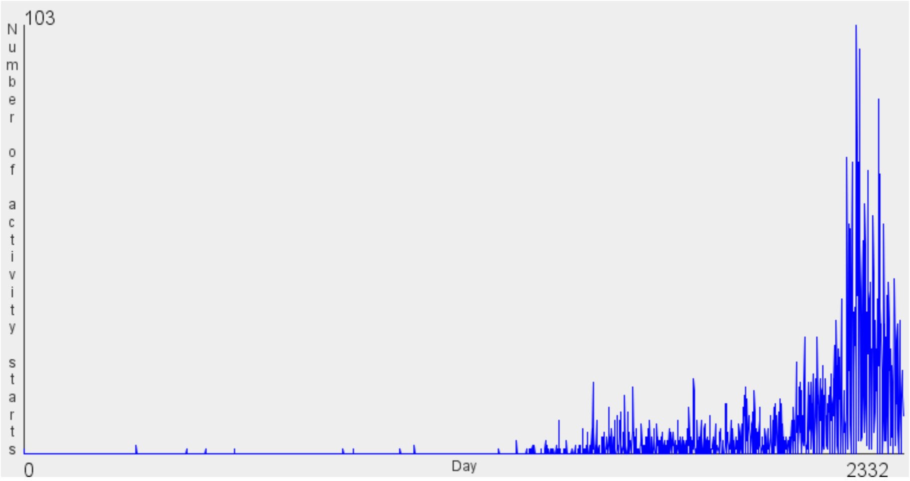


Figure 3.3: Example log showing clear non-week related patterns in activity starting count. The pattern has a granularity of one day and concerns 2332 days in total.

3.1.1 Extracting workload context information from an event log

The approach for tackling the weakness of the classical queueing models is to split up historical data based on the context and the arrival time of activities. To allow using context data later on, it is necessary to obtain context information from some source and to add context information to an event log. This subsection introduces a method, given an event log, to derive context information on the workload over time. In addition, a method is introduced that allows adding context information to the context attributes of events. For example, weather data or stock market information.

Workload information was chosen as the context information to be extracted from an event log. Workload information is in essence information about how busy or not busy the system is at some point in time. The goal is to determine for each day how busy that day is. Extracting workload information can be done in a number of different ways. The first is for each day to look at the global arrival rate and global service rate and define the workload as the service rate divided by the arrival rate and bin these workloads in a small number of classes. Another possibility is to sum the arrival rate over the last weeks and take this as the workload. Then bin these workloads in a small number of classes. Next, it is possible to find the workload for each day by looking at the arrival rates in some window. Then, the workload class is defined by taking the average workload of the entire historical data and using the standard deviation to define classes for busy, quiet and regular. The last of these possibilities was chosen to be implemented, but there is no clear evidence that this is the best method.

The purpose of this method is, given some event log as input, to calculate some metric for the average workload over the entire log. Then, for every event, compare the workload at that instance to the average and put it into either the class “Quiet”,“Regular” or “Busy”. This leads to two challenges.

First, what is an example metric for the workload of a day in the process? Given a day d, the interval three days before that day and three days after are used as a window. The window size of seven is chosen such that any patterns with respect to a week, e.g. nothing being done on the weekend, does not influence the workload metric. The reason for choosing days both before and after the day d is that both can have an influence on the duration of events on day d. It is obvious that items that happen before d could have an influence by being in the same queue. Items that arrive after d can also have an impact, depending on the queueing discipline. For example, when

Day	1	2	3	4	5	6	7	8	9	10	11	12
Number of events	10	4	4	4	4	4	4	4	1	1	1	1
Workload	-	-	-	34	28	25	22	19	16	-	-	-

Table 3.1: Context workload example values.

using LIFO or some priority scheme where gold customers go before other customers it would matter which items come after *d*. Since it cannot be assumed which queueing discipline is used, both items before and after *d* are considered. Strictly speaking this metric does not compute the absolute workload, but rather an expected amount of work to be done before the tasks of this day are handled. Within the sliding window the amount of occurrences of events are counted and that is considered the metric for workload. An example is shown in Table 3.1.

In this example the days 1-3 and 10-12 are part of a warming up and cooling down period, since the window size is three days in both directions. Only the events of days 4-9 will be considered. On day 4, there is a total of 34 events in the scope, namely 4 on the day itself and 30 on days 1-3 and days 5-7. This is summed for all days and is displayed in the Workload row.

The second challenge is: given some metric for workload, what should be the bounds for the three classes. As noted before, there are a number of possibilities. One could choose to bin the workloads in a finite number of sets. Alternatively, one could try to find the mean value and classify any values that are statistically distant from this mean as either significantly busy or quiet. The last approach was chosen, but is not necessarily the best. An argument for this metric is that the standard deviation gives a good indication of cases deviating from average with a statistical significant amount. It is not seen as a problem that a large number of the cases falls into the class of “Regular”, as it would be expected that most days will be regular days.

It is possible to calculate the mean value for the workload by taking the average of the workload over all days. The mean for this toy set is the sum of all the workload values for all days divided by the amount of days considered, which is:

$$(34 + 28 + 25 + 22 + 19 + 16)/6 = 24$$

There is a multitude of choices for which distance from the mean would indicate something being significantly quiet or busy, for example the standard deviation, possibly multiplied by some constant, or a fraction of the total range of values. The choice was made to classify an event context as “Busy” if the workload of that event is more than the mean plus the standard deviation, but this is not necessarily the best choice. Similarly an event context is classified as “Quiet” when the workload value is less than the mean minus the standard deviation. Otherwise, the context will be classified as “Regular”. A formalization is given below in Definition 3.1.1.

Definition 3.1.1 ***Workload context*** *Assume there exists a day counter $d \in \{0, ..., NRDAYS\}$ where $NRDAYS$ indicates the range of days in log L. Assume an ordered list of workload values is supplied where workload(d) indicates the workload on day d, i.e. the amount of activities arrived in a window of 7 days around d. The mean workload is then defined as:*

$$meanW = \frac{\sum_{d \in \{0,...,NRDAYS\}} workload(d)}{NRDAYS}$$

The workload standard deviation is then:

$$stdW = \sqrt{\frac{\sum_{d \in \{0,...,NRDAYS\}} (workload(d) - meanW)^2}{NRDAYS}}$$

Listing 3.1: Workload context information example result

```
<?xml version=”1.0” encoding=”UTF-8” ?>
<contextInfo>
<contextVariable name=”Workload”>
        <contextItem>
                <context value=”Busy”/>
                <time value=”4”/>
        </contextItem>
        <contextItem>
                <context value=”Regular”/>
                <time value=”5”/>
        </contextItem>
    <contextItem>
                <context value=”Regular”/>
                <time value=”6”/>
        </contextItem>
    <contextItem>
                <context value=”Regular”/>
                <time value=”7”/>
        </contextItem>
    <contextItem>
                <context value=”Regular”/>
                <time value=”8”/>
        </contextItem>
    <contextItem>
                <context value=”Quiet”/>
                <time value=”9”/>
        </contextItem>
</contextVariable>
</contextInfo>
```

For each day, the context value $context(d)$ will then be set using the following function:

$$context(d) = \begin{cases} \text{“}Quiet'' & if\ workload(d) < meanW - stdW \\ \text{“}Busy'' & if\ workload(d) > meanW + stdW \\ \text{“}Regular'' & otherwise \end{cases}$$

The standard deviation for the example can be calculated as shown in Definition 3.1.1 and is 5.916. This means any day with a workload lower than $18,084$ would be considered “Quiet”, which corresponds to day 9. Any day with a workload higher than $29,916$ would be considered “Busy”. That means day 4 is considered “Busy”. The rest of the events would be classified as “Regular”. The resulting context information file for this case is described by an example XML file in Listing 3.1.

3.1.2 Adding context information to an event log

This subsection will cover how to add context information from any source to event logs. The input is an event log and context information data. Examples of influential contexts are the weather, the level of workload within the company or the stock market. To support context data not bound to events a format was introduced to support context information as an independent data object. The meta-model of this format can be found in Figure 3.4.

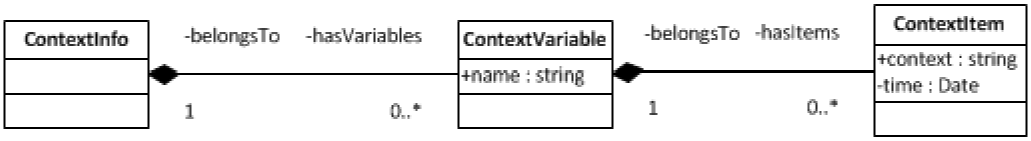


Figure 3.4: Context information metamodel.

Event id	ATT_{type}	ATT_{time}	...
e_1	Sort package	day 2, 14:22	...
e_2	Send package	day 4, 12:11	...
e_3	Inspect package	day 8, 23:59	...
e_4	Send package	day 12, 09:00	...

Table 3.2: Simple event log example with day timestamp attributes.

As can be observed in Figure 3.4, context information has a number of context variables. A context variable is a value for the kind of context being described, for example the weather, the workload or the stock market. All these context variables can have context items, indicating the current state of a context variable at a current time. Example context information data for workload can be found in Listing 3.1.

After obtaining a context, be it from an external source, or be it from distilling it from a business process, it is necessary to enrich event logs with this information in order to use it. The intuition is to look at the time of an event and find a context value based on context information that is relevant at the time of that event. It seems logical to either take the context value that happened right before the event, or the context value that is closest in time to the event. The time right before the event was chosen, although it is not necessary the best option.

For example, take the context information from Listing 3.1 as an input and say the input event log is described by Table 3.2. For the first event e_1 the timestamp attribute $ATT_{time}(e) =$ day 4, 12:34. The context value just before this is that of day four, which is "Busy". Any event (as an example e_2) for which the timestamp is before the first context value, is classified as null. Any event after the last timestamp is classified as the last timestamp. The result of enriching the example log with the example context information can be seen in Table 3.3.

3.1.3 Finding basic queues

After adding all wanted context information to an event log, the goal is to find a set of *basic queues*, which means to cluster all resources and activities that occur in the event log such that each cluster of resources and activities accurately represents the servers and work items at a queue and workstation in the real process. As input for this method we have an event log and as output a set of basic queues which are useful for finding a queue collection. An example output set of basic queues is illustrated in Table 3.7. It is assumed for every event e in the input log that the attributes for activity $ATT_{type}(e)$ and resource $ATT_{res}(e)$ are not null, i.e.:

Event id	ATT_{type}	ATT_{time}	$ATT_{context}$	...
e_1	Sort package	day 2, 14:22	$\perp$	...
e_2	Send package	day 4, 12:11	"Busy"	...
e_3	Inspect package	day 8, 23:59	"Regular"	...
e_4	Send package	day 12, 09:00	"Quiet"	...

Table 3.3: Simple event log example with day timestamp and context attributes.

Activity / Resource	Henk	Anita	Nick
Sort package	10	16	0
Send package	10	16	1
Inspect package	12	0	1
Approve package inspection	0	1	30

Table 3.4: Matrix showing event executions for given resource and type of some log.

$\forall c \in L : [\forall e \in ATT_{trace}(c) : ATT_{type}(e) \neq \perp \wedge ATT_{res}(e) \neq \perp]$. The solution for this problem is bound by a constraint: two clusters should never contain the same activity, since it should not be the case that work items can be handled at multiple queues. It should be possible for resources to work at multiple queues. Within these constraints, a method should be devised to cluster resources and activities.

There exist a multitude of ways to consider when resources and activities should or should not be clustered together. An intuitive way of deciding whether activities and resources fit together is to look at how many times resources have executed certain activities in historical data and see whether this amount exceeds some threshold indicating a strong link between the resource and activity. There are, however, many possible metrics for deciding what this threshold should be. The first idea is to take a fraction of the maximum times some activity is executed by some resource as the threshold. Another example is the average amount an activity is executed over all resources. One more example is to find the mean and standard deviation over the amount of times a certain activity is executed and include all pairs where a resource executes this activity at least the minus or plus the standard deviation amount of times. The last example is defining the threshold as: the average amount of times an activity is executed, considering only resources that execute that activity. This last example will be used as an example metric for implementing the technique, but it is not the case that one metric is distinctly better than the others.

Example method for finding a threshold: The threshold is to be defined as the average amount of times an activity is executed, considering only resources that execute that activity. The input for this method is execution count data such as in Table 3.4. The output is a list of fairness thresholds, an example of which is illustrated in Table 3.5.

To find the average amount of times a resource is executed, one must count the amount of times any resource executes any activity. This is done by summing all the events in some log L for which the resource r and activity a match these values, i.e.:

$$executes(a, r) = \sum_{c \in L} \sum_{e \in ATT_{traces}(c)} \begin{cases} 1 & \text{if } ATT_{type}(e) = a \wedge ATT_{res}(e) = r \\ 0 & \text{otherwise} \end{cases}$$

Say we want to count the amount of times resource “Henk” executes “Sort package”. Then all events e for which $ATT_{res}(e)$ = “Henk” and $ATT_{type}(e)$ = “Sort package” are counted. An interesting example input log would be very long, so instead an example set of aggregated counted data is shown in Table 3.4. In this table one can see that the combination of resource “Henk” and activity “Sort package” was observed ten times.

The main idea is to take a threshold of the average amount of executions of some activity by some resource. This is possible from two perspectives: the perspective of the activity and a fair share of work, i.e.: did this activity get executed by this resource a fair amount of times compared to other resources that execute this activity. The other perspective is that of the resource and a fair share of attention, i.e.: did this resource execute this activity a fair amount of times compared to other activities this resource executes. For both perspectives there will be an accompanying threshold value.

Obtaining the fair amount threshold for some activity a will go as follows: Sum the total amount of times a was executed in a log L and call this $executions(a, L)$.

$$executions(a, L) = \sum_{c \in L} \sum_{e \in ATT_{traces}(c)} \begin{cases} 1 & \text{if } ATT_{type}(e) = a \\ 0 & \text{otherwise} \end{cases}$$

Consider the set R as all resource attributes that exist in L, i.e.:

$$\forall_{r \in R} \exists c \in L \exists_{e \in ATT_{traces}(c)} ATT_{res}(e) = r$$

Then sum the amount of resources for which holds that they executed a at least once and call this $resHelping(a, L, R)$.

$$resHelping(a, L, R) = \left\{ \sum_{r \in R} \begin{array}{ll} 1 & \text{if } \exists c \in L \exists_{e \in ATT_{traces}(c)} ATT_{type}(e) = a \wedge ATT_{res} = r \\ 0 & \text{otherwise} \end{array} \right.$$

Divide the total executions by the amount of resources to obtain the threshold $actThreshold(a, L, R)$.

$$actThreshold(a, L, R) = executions(a, L)/resHelping(a, L, R)$$

As an example, consider the threshold for activity "Sort package" of the matrix example in Table 3.4. The amount of executions in the entire log are the same as the sum of all counts concerning activity "Sort package" which is: 10 + 16 + 0 = 26. The amount of helping resources is the amount of resources for which the count of "Sort package" is above zero. This concerns both "Henk" and "Anita" and not "Nick", so this value is two. The threshold is then: 26/2 = 13. The threshold values for all activities can be found in Table 3.5.

Obtaining the fair amount threshold for some resource r is similar to that of activities: Sum the total amount of times r executes something in a log L and call this $executedBy(a, L)$.

$$executedBy(a, L) = \sum_{c \in L} \sum_{e \in ATT_{traces}(c)} \begin{cases} 1 & \text{if } ATT_{res}(e) = r \\ 0 & \text{otherwise} \end{cases}$$

Consider the set A as all activity attributes that exist in L, i.e.:

$$\forall_{a \in A} \exists c \in L \exists_{e \in ATT_{traces}(c)} ATT_{type}(e) = a$$

Then sum the amount of resources for which holds that r executes them at least once and call this $activitiesDone(r, L, A)$.

$$activitiesDone(r, L, A) = \left\{ \sum_{a \in A} \begin{array}{ll} 1 & \text{if } \exists c \in L \exists_{e \in ATT_{traces}(c)} ATT_{type}(e) = a \wedge ATT_{res} = r \\ 0 & \text{otherwise} \end{array} \right.$$

Divide the total executions by the amount of resources to obtain the threshold $resThreshold(r, L, A)$.

$$resThreshold(r, L, A) = executedBy(a, L)/activitiesDone(r, L, A)$$

As an example, consider the threshold for resource "Anita" of the matrix example in Table 3.4. The amount of times "Anita" executes something in the entire log are the same as the sum of all counts concerning resource "Anita" which is: 16 + 16 + 0 + 1 = 33. The amount of activities done is the amount of activities for which the count of "Anita" is above zero. This concerns "Sort package", "Send package" and "Approve package inspection" and not "Inspect package", so this value is three. The threshold is then: 33/3 = 11. Now a metric for a threshold of a fair share for both activities and resources was defined. The fair share threshold values based on Table 3.4 are shown in Table 3.5.

Resource	Fair share of attention threshold
Henk	11
Anita	11
Nick	11
Activity	Fair share of work threshold
Sort package	13
Send package	9
Inspect package	7
Approve package inspection	16

Table 3.5: Matrix showing threshold values for resources and activities.

Example cost function definition based on threshold values: The next step is to introduce a cost function for a certain proposed queue. Given this cost function it then becomes possible to find which clustering is the best one, i.e.: the one with the lowest cost according to this function. The input of this function is a cluster of activities, for example resources {"Henk","Anita"} and activities {"Sort package","Send package"}. In addition, event log information in the form of counting data from an event log as in Table 3.4 and fair thresholds, as illustrated in Table 3.5 are needed. The output of the cost function is a value indicating how bad or good a cluster is. The main constraints of this function are: There should be a cost when pairs inside a cluster fail to meet threshold amounts. In addition, there should be a cost when pairs partially in the cluster do meet threshold requirements.

Now an example cost function conforming to the set constraints will be discussed. Before defining the cost of an entire cluster, the cost for a pair of resource and activity is defined by the following rules:

- Initially, the cost is zero.
- If a (resource,activity) pair is in the cluster, but the resource does not execute the activity a fair amount of times from the resource viewpoint, increase the cost by that fair amount minus the amount of executions.
- If a (resource,activity) pair is in the cluster, but the resource does not execute the activity a fair amount of times from the activity viewpoint, increase the cost by that fair amount minus the amount of executions.
- If a (resource,activity) pair is not in the cluster, but either the resource or the activity is in the cluster and the resource executes the activity at least the fair amount of times from the resource viewpoint, increase the cost by that amount minus the fair amount.
- If a (resource,activity) pair is not in the cluster, but either the resource or the activity is in the cluster and the resource executes the activity at least the fair amount of times from the activity viewpoint, increase the cost by that amount minus the fair amount.

Note that multiple cases can be true at the same time and hence the cost of all those cases will be summed.

As an example, consider resource "Anita" and activity "Sort package". The amount of times "Anita" executes "Sort package" is 16. The threshold value for "Anita" is 11, which is smaller than 16, so no cost is added. Similarly, the threshold value for "Sort package" is 13, which is also smaller than 16 and hence also includes no cost. The total cost of this pair remains zero, indicating a good match. As another example, consider resource "Anita" and activity "Approve package inspection". The threshold value for "Anita" is 11, yet she only executes "Approve package inspection" once. The cost from the resource viewpoint is hence 11 - 1 = 10. The threshold value for "Approve package inspection" is 16, yet "Anita" executes it only once. The cost from the activity viewpoint is hence 16 - 1=15. The total cost for this pair is then 10 + 15 = 25.

Activity / Resource	Henk	Anita	Nick
Sort package	10	|16|	0
Send package	10	|16|	1
Inspect package	|12|	0	1
Approve package inspection	0	1	|30|

Table 3.6: Matrix showing event executions for given resource and type. Activities and resources that are matched, are indicated as |#|.

The cost function for the cluster of resources and activities is then defined as follows: Sum for each pair of (resource,activity) in the cluster the cost of this pair.

As an example, the cost for the queue with resource "Anita" and activities {"Sort package","Send package"} will be calculated, with help of the fair share values in Table 3.5. A previous example has already shown the cost of the pair ("Anita","Sort package") to be zero. The pair ("Anita","Send package") will also result in a cost of 0, since the thresholds for "Anita" and "Send package" are respectively 11 and 9, which is both smaller than 16. Apart from the pairs inside the cluster, all pairs consisting one element from the cluster need to be taken into account. This concerns the pairs ("Anita","Inspect package"), ("Anita",Approve package inspection"), ("Henk", "Sort package"), ("Nick","Sort package"), ("Nick","Send package") and ("Henk","Send package"). Of these pairs, only the pair ("Henk","Send package") is the only one with a cost. The threshold for "Send package" is 9, while the amount of times "Send package" is executed by "Henk" is 10. The threshold for "Henk" is 11, so there is no cost from the resource viewpoint. Hence, the cost for this pair is 10-9 = 1. All other pairs produce no cost, since none of their execution values are higher than the threshold. The total cost for this queue is hence 1. This value can then be compared to other possible configurations, such as including "Henk" in the queue.

The weakness of this technique is illustrated when considering the queue with resources {"Henk", "Anita"} and activities {"Sort package", "Send package"}. Even though the amount of times "Henk" executes "Sort package" is 10, which is not much lower than "Anita" with 16 and certainly not much lower than the 12 times that "Henk" executes "Inspect package", the value falls under the threshold for both perspectives. Intuitively, this seems like bad behaviour. Additionally, the activity "Send package", where "Nick" executes once, has a threshold of 9 instead of 11, making the cost of pair ("Henk","Send package") from the activity perspective zero. This is a large influence by only a single execution in the entire log and it seems such a small difference should not have such an impact. A metric that has no weakness such as this one has not been found during the development of these techniques and is left as future work.

Example cluster finding algorithm based on cluster costs: The goal is to find a clustering of resources and activities. The input for this method is a cluster cost function as described in the previous paragraph. The wanted output is queue clustering similar to that in the example of Table 3.7.

An attempt to use a brute force method for trying all possible clusterings turned out too slow. For this reason, a greedy algorithm was implemented. Given the constraint that no two queue should hold the same activity, this algorithm tries to cluster all activities with a set of resources for which the cost metric is low and then tries to match activities with other activities (including their resource sets). Each step of the way the greedy algorithm will compare if the cost of adding some activity and resource is lower or if the cost of not merging is lower. Each time the lowest cost choice will be executed, until for every cluster it holds that not merging has the lowest cost. After running this algorithm, a clustering of resources and activities is obtained. Table 3.6 shows the result on the example counting matrix of Table 3.4.

The problem stated in this section was to cluster all resources and activities that occur in the

Basic queue ID	Activity set	Resource set
1	Inspect package	Henk
2	Sort package, Send package	Anita
3	Approve package inspection	Nick

Table 3.7: Illustration of queue clustering output based on the input counting matrix in Table 3.4.

event log such that each cluster accurately represents the servers and work items at a queue and workstation in the real process and in that way obtain a set of basic queues. The set of basic queues that follows from the example in Table 3.6 are shown in Table 3.7 and illustrate a typical output for this method. This queue clustering will be an input for the techniques described in the next section and is ultimately used for finding a queue collection.

3.2 Finding queue characteristics

In the previous section a technique for obtaining a set of basic queues has been shown. This section will focus on explaining what the problem is with current queueing models and introduces a way to tackle this problem. Then, a method will be proposed to convert basic queues and event log information to obtain a queue collection.

A limitation of the classical queueing model is that predictions are often made for an average over the entire time-frame. In real life, processing times are heavily influenced by the time of day at which an activity arrives and the context of the time an activity arrives. To provide a new approach, it is necessary to first consider the important characteristics of queues and the input information that is necessary to find the characteristics that will be covered. These things are described in Subsection 3.2.1. Next, a few examples will illustrate the necessity of the techniques that will be covered in the coming subsections.

The first example is a secretary that works only in morning hours and has to do relatively small tasks, say 15 minutes per task. In the classical approach all tasks that arrive in the afternoon would be predicted to take a very long time, while the tasks arriving in the morning would take a relatively short time. On average this would give a very large expected completion time. A better estimate would be to say that in the morning the expected time until completion is 15 minutes, while in the afternoon the expected time is some time the next day. This idea forms the basis of splitting the arrival and service rates, based on the relative arrival time of an activity and obtain a distribution useful for predictions for each part. Subsection 3.2.2 covers how splitting the arrival and service rates based on time can be done.

Another example is an ice-cream company that sells ice-cream directly to customers. The classical queueing model would model the amount of arriving customers in a general scenario. However, the weather is of great influence for the amount of ice-cream that is bought at some point in time. As such, it is not only relevant to look at the time at which an activity arrives, but also at the context at that time. So, it is also relevant to split arrival and service time predictors based on context. Subsection 3.2.3 covers how splitting the arrival and service rates based on context would work.

In Subsection 3.2.4 the techniques proposed in Subsections 3.2.2 and 3.2.3 are wrapped up and combined to form a method for finding a queue collection, given a set of basic queues and an event log.

3.2.1 Important queue characteristics

To obtain a network of queues that accurately describes a process, it is necessary to think about which characteristics would be important to have. The most basic characteristics are the servers and the types of activities that arrive at the queue, which have already been covered as basic queues in Section 3.1.

Other possibly interesting characteristics of a queue will now be listed with a short explanation. This list is not exhaustive. In this work only the arrival and service rates are chosen to be looked at, because these are necessary groundwork for most of the other characteristics.

- *Arrival rate*, the average amount of tasks that arrive at the queue in some time interval.
- *Service rate*, the average amount of tasks that are completed at the work station in some time interval.
- *Service discipline*, the order in which items are handled. For example FIFO, LIFO, or some priority scheme where certain item types are more important than others (gold customers versus regular customers). This also includes other patterns such as batch execution.
- *Buffer size*, the amount of items that fit in the queue before new arriving items will be dropped. Can be infinite.
- *Resource availability*, information concerning when certain servers of this queue are available.

Some metric needs to be found to find the duration of events. We assume that it is not possible to derive Effective Processing Times (EPT's) [19], because it is on one hand difficult to monitor human resources to such an extent and on the other that there is no standard form of describing EPT's in existing logs that can be exploited. It is assumed that it is not possible to extract whether an activity with corresponding timestamp indicate entering a queue, entering a work station or exiting a work station in a generic way. As such, an estimate has to be made using the timestamps of the events. Some possibilities of how to estimate durations are now listed:

- The timestamp of some event indicates the time an event enters a workstation. The duration is inferred from the timestamp of the current an next event, for example taking a factor of the difference in time.
- The timestamp of some event indicates the time an event enters a workstation. The duration is defined as the difference between the timestamp of the current and the next event.
- The timestamp of some event indicates the time an event enters a queue. The duration is defined as the difference between the timestamp of the current and the next event.

We have chosen to estimate event start and end times as in the last case: The start time for an event is the timestamp of that event. The end time (and the start time of the next event) is the timestamp of the next event in the trace. It is assumed that the start times correspond to the time when the work item enters the queue, rather than the time it arrives at the workstation. The end time is assumed to be the time when the item leaves the work station.

So, the duration of an event e is estimated by looking at the timestamp $ATT_{time}(e)$ and the timestamp $ATT_{time}(e')$ of the event e' that follows directly after e. Looking at the example event log in Table 2.1 the starting time for the event with activity "Receive package" of trace "12324" would be 14-02-2014 13:45, the end time would be 20-02-2014 19:42 and hence the total duration would be 6 days, 5 hours and 57 minutes. When considering the last event of a case, the end time will be equal to the start time. These activity durations will be used to predict arrival and service rates for a set of basic queues. In the next subsections splitting arrival and service distributions based on time and context will be discussed.

Case ID	Activity	Resource	Timestamp
1	Inspect package	Henk	Monday 29-09-2014, 10:02
1	Sort package	Anita	Monday 29-09-2014, 17:12
1	Approve package inspection	Nick	Monday 06-10-2014, 10:24
2	Inspect package	Henk	Monday 06-10-2014, 15:56
2	Sort package	Anita	Tuesday 07-10-2014, 09:12
3	Inspect package	Henk	Monday 05-01-2015, 09:00
3	Sort package	Anita	Monday 05-01-2015, 17:30

Table 3.8: Example event log with three cases that have events with activity, resource and timestamp attributes.

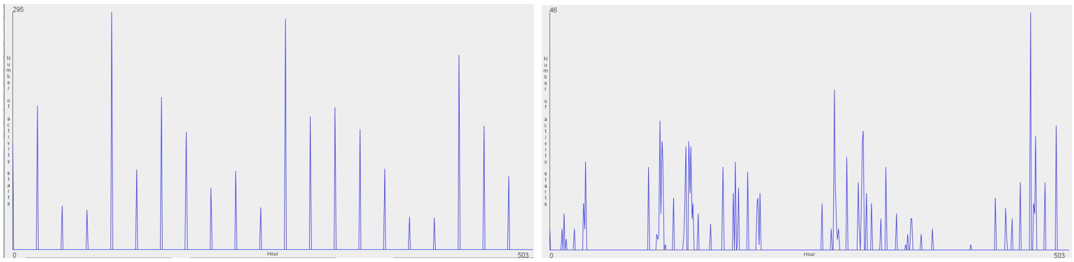


Figure 3.5: BPI 2011 challenge log and Dutch municipality process log showing clear week patterns. On the x-axis the hours of three consecutive weeks are plotted. The y-axis shows the amount of activities started in some hour.

3.2.2 Time-based log splitting

The approach for tackling the weakness of the classical queueing models is to split up historical data based on the relative arrival time of activities. A reason for pursuing this kind of split is showed in Figure 3.2, where a clear week pattern is shown to exist in a log. This is not a special case: two more real-life event logs from different sources with such a pattern are shown in Figure 3.5. This subsection will cover how one can split up data based on the relative arrival time, given an event log as input. An example input is given in Table 3.8 The output will be a set of events split up based on their relative arrival time.

This approach aims to find a golden mean between these two extremes by looking at a number of different parts of a larger period of time and assigning arrival rates to those parts. The challenge is to decide how long should this larger period be and how long should the individual parts be. In business processes describing a regular business, tasks are bound by week and day patterns, as has been observed in Figure 3.5 and Figure 3.2. For this reason, it was chosen that the larger period should be a week. In general this week should be split up such that there is no underfitting in the parts, i.e.: it should be split up in enough parts to find patterns such as working 9 to 5 and people working half days. On the other hand, there should be no overfitting in the parts, i.e.: the week should not be split up in such a way that only a few cases are present for each part and hence no statistically sound predictions can be made. Definition 3.2.1 formalizes the time quantities used in this approach.

Definition 3.2.1 ***Time quantities*** *Let the larger time period $MainTime$ be defined as one week time. Then, let the minimal amount of parts be $MinParts = 7$, indicating the week should at least be split up in parts of one day time. The value $k \in \mathbb{N}_{>0}$ indicates in how many parts each day will be split up. The total amount of parts in a week is $MinParts \cdot k$. Each part has a size of $MainTime/(MinParts \cdot k)$ time.*

So, the values for $MainTime$ (one week) and $MinParts$ (7) are treated as constant truths. The variable k is introduced for indicating in how many parts a day of the week should be split up hence it defines what the length of those parts is. For the coarsest granularity ($k = 1$) that means the week is split up in 7 parts of a single day. Higher k implies finer grained results, e.g.: $k = 24$ means every hour of every day in a week is a different part.

The challenge is now simplified to: what is the right k such that patterns in the business process are visible, but the amount of events per part is not too small? Or: What is the right k to split up a log, such that no overfitting and no underfitting occur?

At this moment, an automated method for extracting a good value for k, given some event log, is not evident. A possible approach is to try number of different k's, while keeping track of how "good" these tries are. To incorporate the day and week patterns, it does not suffice to look at multiples of two. Rather, it would be a good idea to look at the multiples of some small set of prime numbers, such that a larger part of the total number space can be traversed, while ignoring numbers that are probably not interesting. For this purpose the Primetree structure is introduced in Definition 3.2.2.

Definition 3.2.2 ***Primetree*** *Let P be the set of all possible primes. Consider the finite set of primes $P' \subset P$. A Primetree is a tree* T *with vertices T_V, edges T_E and root value 1. Consider a function $val(v)$ which for a vertex returns the value of that vertex. Each vertex $v \in T_V$ has an edge towards all vertices v' for which holds: $\exists_{p \in P'} val(v') = val(v) \cdot p$.*

Figure 3.6 gives an example of a Primetree using a limited set of primes, in this case two, three and five and a limited set of vertices: one, two, three, five, six, nine and fifteen. A Primetree can be used as follows: Imagine the values for all vertices in the Primetree are in fact possible values of k, i.e.: the value of a vertex indicates in how many parts the log would be split, namely $7 \cdot k$. Then one could find a limited search space for possible values of k, given a PrimeTree.

Then consider automatically traversing such a Primetree, trying out all possibilities and finding which k is good. For each vertex it would then be necessary to check for underfitting and overfitting. To automatically discern when overfitting becomes a problem is relatively trivial, namely setting the threshold to any number for which the part has a statistically significant size, e.g. 100 and checking how much parts are overfitting. To discern when underfitting happens and what pattern is the one fitting the business process, is far from trivial. An expert on the process would need to look at the patterns and see whether they fit their business process. For this reason an automatic approach to this was not pursued, but instead a method was provided for experts to explore a Primetree and find the best fitting k. More information on how this was achieved in practice is covered in Subsection 5.3.2.

Next a technique for splitting log information based on their relative arrival time and putting them into the corresponding time parts will be explained. Consider an event e with timestamp $ATT_{time}(e)$ and consider a value k that indicates the amount of parts a day will be split up in. Then the absolute arrival time $ATT_{time}(e)$ needs to be mapped to a relative arrival time. This is achieved by performing a modulo operation based on the $MainTime$ value, i.e.:

$$\text{Relative arrival time for } e : relTime(e) = ATT_{time}(e) \% MainTime$$

Then, using the relative arrival time, the corresponding time part for this event has to be found by dividing the relative time to the amount of time that is in one part of the week. Recall that $MinParts$ is the constant indicating in how many parts $MainTime$ should be split.

$$\text{Time part for } e : timePart(e) = \lfloor relTime(e) / \frac{MainTime}{MinParts \cdot k} \rfloor$$

Splitting up events is then achieved by calculating the time part for each event and grouping events by this time part.

Now let's look at an example: say $k = 2$, and we want to find the time bucket for an event for which the time attribute is Tuesday the 26th of August 2014 at 12:34. The relative weektime, acquired by doing a modulo on the size of a week will be Tuesday at 12:34. There are ($MinParts \cdot$

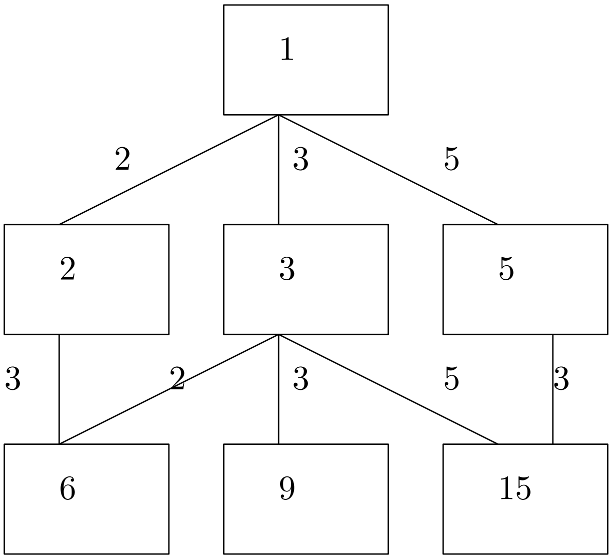


Figure 3.6: Partially unfolded primetree with primes 2,3,5.

Time part	Activity	Resource	Timestamp
1 (Monday 0:00-11:59)	Inspect package	Henk	Monday 29-09-2014, 10:02
	Inspect package	Henk	Monday 05-01-2015, 09:00
	Approve package inspection	Nick	Monday 06-10-2014, 10:24
2 (Monday 12:00-23:59)	Inspect package	Henk	Monday 06-10-2014, 15:56
	Sort package	Anita	Monday 29-09-2014, 17:12
	Sort package	Anita	Monday 05-01-2015, 17:30
3 (Tuesday 0:00-11:59)	Sort package	Anita	Tuesday 07-10-2014, 09:12
4 (Tuesday 12:00-23:59)	-	-	-
...	...	...	...
14(Sunday 12:00-23:59)	-	-	-

Table 3.9: Event log data from Table 3.8 split up based on time part for $k = 2$. Only parts 1, 2 and 3 have data based on the input event log.

$k) = 7 \cdot 2 = 14$ buckets which have the size of half a day. Tuesday after 12:34 would fall into the 4th bucket, since it falls into the 4th half day of the week. An partial example output for this method can be seen in Table 3.9, based on the event log example in Table 3.8.

So, all events in a log can be grouped by relative time part in the week. For each time part it is then possible to create a distribution over the durations of all the events in those buckets. This shall be covered in more detail in Subsection 3.2.4.

3.2.3 Context-based log splitting

The approaches for tackling the weakness of the classical queueing models is to split up historical data based on the context at the arrival time of activities. A reason for pursuing this kind of split is showed in Figure 3.3, where a clear shift in context is shown to exist in a log. This subsection will cover how one can split up data for a number of different contexts, given an event log as input. The context of an event is provided in the context attribute. An event e should have context value $ATT_{context}(e) \neq \perp$ for this approach to make sense. An example input log is shown in Table 3.10.

Splitting the events on basis of context is similar to that of splitting based on time. Events

Case ID	Activity	Resource	Context
1	Inspect package	Henk	Busy;Rainy
1	Sort package	Anita	Regular;Rainy
1	Approve package inspection	Nick	Regular;Sunny
2	Inspect package	Henk	Quiet;Sunny
2	Sort package	Anita	Quiet;Sunny
3	Inspect package	Henk	Regular;Sunny
3	Sort package	Anita	Busy;Rainy

Table 3.10: Example event log with three cases that have events with activity, resource and context attributes.

Time part	Activity	Resource	Context
Busy;Sunny	-	-	-
Busy;Rainy	Inspect package	Henk	Busy;Rainy
	Sort package	Anita	Busy;Rainy
Regular;Sunny	Approve package inspection	Nick	Regular;Sunny
	Inspect package	Henk	Regular;Sunny
Regular;Rainy	Sort package	Anita	Regular;Rainy
Quiet;Rainy	-	-	-
Quiet;Sunny	Sort package	Anita	Quiet;Sunny
	Inspect package	Henk	Quiet;Sunny

Table 3.11: Event log data from Table 3.10 split up based on context value. Context combinations not present in the log have no values.

should be grouped based on their unique context string. So, for example an event with context "Cloudy weather" would end up in the context bucket for "Cloudy weather". If multiple contexts are present, i.e.: there is a context variable for weather and one for workload, which can be either "Busy" or "Quiet". There will be buckets for all combinations of these context values, i.e.: "Cloudy;Busy", "Cloudy;Quiet", "Sunny:Busy" and "Sunny;Quiet". Based on the example input log of Table 3.10, the result of splitting on context value is shown in Table 3.11. The next subsection will cover how to combine these techniques to create a queue collection.

3.2.4 Finding characteristics based on time and context

The previous subsections described how to find the context and relative part of the week of a given activity. The next and final step is splitting the activities belonging to a queue into a matrix of event sets that represent the historical data of a relative arrival time within a certain context. For each unique matrix cell the data will be saved as distribution functions for the service time and a number indicating the arrival rate over the duration of events that fall within that time and context frame. An example input log for this technique is shown in Table 3.12.

Assume a queue clustering corresponding to this log is present as in Table 3.13. The first step in this technique is for all queues to split all data in an event log based on their relative time and context attribute. This is done for each queue as was described in Subsection 3.2.2 and Subsection 3.2.3 respectively. The result of this step on the example input can be seen in Table 3.14.

The final step in this process is converting the sets of events for all cells into a distribution of service times. This is achieved by binning the durations of the events in a cell using bins with the same sizes as time parts. Using the example queue collection in Table 3.14, consider the queue with activities "Inspect package","Sort package" and consider the cell for time part "1" and context "Busy;Rainy". There are two events in this cell, both of which can be found in the input event

Case ID	Activity	Resource	Timestamp	Context
1	Inspect package	Henk	Monday 29-09-2014, 10:02	Busy;Rainy
1	Sort package	Henk	Monday 29-09-2014, 17:12	Busy;Rainy
1	Approve package inspection	Nick	Monday 06-10-2014, 10:24	Busy;Rainy
2	Inspect package	Henk	Monday 06-10-2014, 15:56	Quiet;Rainy
3	Inspect package	Henk	Monday 05-01-2015, 09:00	Quiet;Sunny
3	Sort package	Henk	Monday 05-01-2015, 09:30	Quiet;Sunny
4	Inspect package	Henk	Monday 16-11-2015, 08:12	Busy;Rainy
4	Sort package	Henk	Monday 16-11-2015, 21:22	Busy;Rainy
4	Approve package inspection	Nick	Monday 16-11-2015, 11:14	Busy;Rainy

Table 3.12: Example event log with three cases that have events with activity, resource, timestamp and context attributes.

Queue ID	Activities	Resources
1	Inspect package, Sort package	Henk
2	Approve package inspection	Nick

Table 3.13: Example queue clustering based on the example log of Table 3.12.

Queue activities					
Inspect package, Sort package	Time part Context	1	2	3	...
	Busy;Rainy	(Henk, Inspect package, Monday 29-09-2014,10:02, Busy;Rainy) (Henk, Inspect package, Monday 16-11-2015, 08:12, Busy;Rainy)	(Henk, Sort package, Monday 29-09-2014, 17:12 , Busy;Rainy) (Henk, Sort package, Monday 16-11-2015, 21:22, , Busy;Rainy)	- - - - - -	... ...
	Busy;Sunny	-	-	-	...
	Quiet;Rainy		(Henk, Inspect package, Monday 06-10-2014, 15:56, , Quiet;Rainy)	- - -	...
	Quiet;Sunny	(Henk, Inspect package, 05-01-2015, 09:00 , Quiet;Sunny) (Henk, Sort package, 05-01-2015, 09:30 , Quiet;Sunny)		- - - - - -	... ...
	...	...	...	...	...
Approve package inspection	Time part Context	1	2	3	...
	Busy;Rainy	(Nick, Approve package inspection, Monday 16-11-2015, 11:14, Busy;Rainy) (Nick, Approve package inspection, Monday 06-10-2014, 10:24, Busy;Rainy)	-	-	...
	Busy;Sunny	-	-	-	...
	...	...	...	...	...

Table 3.14: Example queue collection for two queues. For every queue a matrix with relative time parts and contexts is shown with example events from Table 3.12.

log of Table 3.12 as well. Using this event log, one can obtain the durations for these two events by looking at the timestamp of those events and the events that come directly after them. For this example, the event at 29-09-2014,10:02 has a duration of 5 hours and 10 minutes, given that the following event starts at 29-09-2014, 17:12. For the event at 16-11-2015, 08:12 the duration is 13 hours and 10 minutes, since the following event starts at 16-11-2015, 21:22. This results in the values 5 and 13 which then have to be binned. Since the time part size is half a day, 5 hours will be mapped to 0 time parts and 13 hours will be mapped to 1 time part. The distribution for this particular example is not very interesting with one sample at 0 hours, and one at 12 hours. A more interesting example distribution is illustrated by Figure 4.2.

Chapter 4

Using queue collections for predictions and analysis

One of the main goals is to predict delays and sojourn times for a process. The queue collection described in Subsection 3.2.4 will be used as an input to do such predictions. This chapter will cover three prediction methods based on the queue collection: Queue exit time prediction (the time it takes an activity to be completed once it arrives at some queue) in Section 4.1, activity routing prediction (the most likely activity that follows after a certain activity is completed) and activity sojourn time prediction (the time it takes for a complete trace to be completed). Section 4.1 and Section 4.2 provide an answer to the Research Sub-Question 1.2.3. In addition, an analysis method is proposed to find the biggest bottlenecks in the queue collection in Section 4.3, which provides an answer to Research Sub-Question 1.2.1.

Throughout this section, the design choices and techniques explained revolve around a large assumption: The queues have infinite servers. This means there will be no queueing model like a Markov chain or a state machine. This approach will focus only on the expected service times, based on the context and time of an arriving event.

4.1 Queue exit time prediction

This section will discuss how to use a queue collection to predict how long it takes for an event arriving at some queue exits to exit that queue. To achieve this, the relative time part and context of this event will be derived according to the techniques presented in Subsection 3.2.2 and Subsection 3.2.3 respectively. Assume that as input there is a queue collection such as shown in Table 3.14. Then, depending on the activity of an event one can find the queue in the queue collection at which this event should arrive. Using the relative time part and context information of the event, one can then find the service time distribution corresponding to that queue, time part and context. The question then is, how to use this service time distribution to predict the time this event will spend in the queue and at the work station. There are a few alternatives to what value to choose from the distribution as the predicted exit time.

First of all, it is possible to use the expected value of the distribution to predict the exit time. An example will now illustrate why the expected value might not be a good estimate. Consider Figure 4.1 as an example service time distribution of a queue for some time part and context. When calculating the expected value of this distribution, it is possible that the resulting value is right between two peaks, indicating a period where no work is done. It would make no sense to

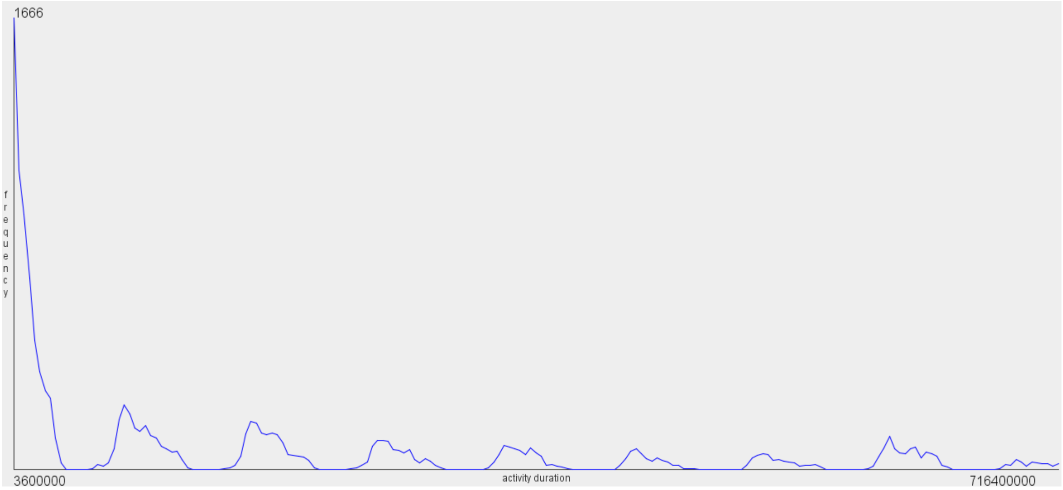


Figure 4.1: Typical duration distribution for time cycle based processes

predict a duration that never occurred in the historical data and would never normally occur in the process. For this reason this approach was not pursued.

The second option is using the most likely value of the distribution for predicting the exit time. Looking at Figure 4.1 again, it seems that the the highest probability would be within the time-frame in which events are completed and would at the same time be very likely close to the actual value. The downside of this approach is that less probable but also valid alternatives are not taken into account. This is the method that will be used.

Another possible option is using another piece of information from an arriving event. If it is known how long the event has already waited in this queue, this can be taken into account. For example, if it is known that it has already waited a day after arriving and the most likely value for completion is somewhere within the day that has already passed, the most likely value after that point in time would be a better estimate. A simple way to achieve this is to not take the highest probability of the entire distribution, but to look at the highest probability in the distribution that is longer than the waiting time. This technique has not been implemented, but is an interesting future work topic.

So, predicting the exit time will go as follows: Consider a queue collection C and some arriving event e as input. The first step is finding the queue in the queue collection which corresponds to the activity of e, i.e.: The queue $q \in C$ for which activity $ATT_{type}(e)$ is in the set of activities belonging to q. Once that queue is found, the service time distribution corresponding to the event's time $ATT_{time}(e)$ and context $ATT_{context}(e)$ will need to be found. Before that is possible, the time part needs to be extracted from the timestamp, as specified in Subsection 3.2.2. Assuming the time part is found, one can obtain the distribution using this time part and the context, with the method specified in Subsection 3.2.4. The final step is then to find the most likely value in this distribution. If multiple equally likely values exists, the first value encountered is predicted.

Now an example will be given for a simple case. Suppose there is an event for which we would want to predict the exit time and there is a queue collection. Then suppose the correct queue in the queue collection is found and assume that the distribution corresponding to the context and time part of the event is shown in Figure 4.2. The predicted duration time will be the most likely value, i.e.: the highest peak in this distribution, which is six hours. The queue exit time predictor will then predict the starting time of the input event plus six hours as the exit time.

So, once a queue collection is known, it becomes possible to find the distribution that conforms to the time part and context in which some event arrives. Then, the most likely value will be

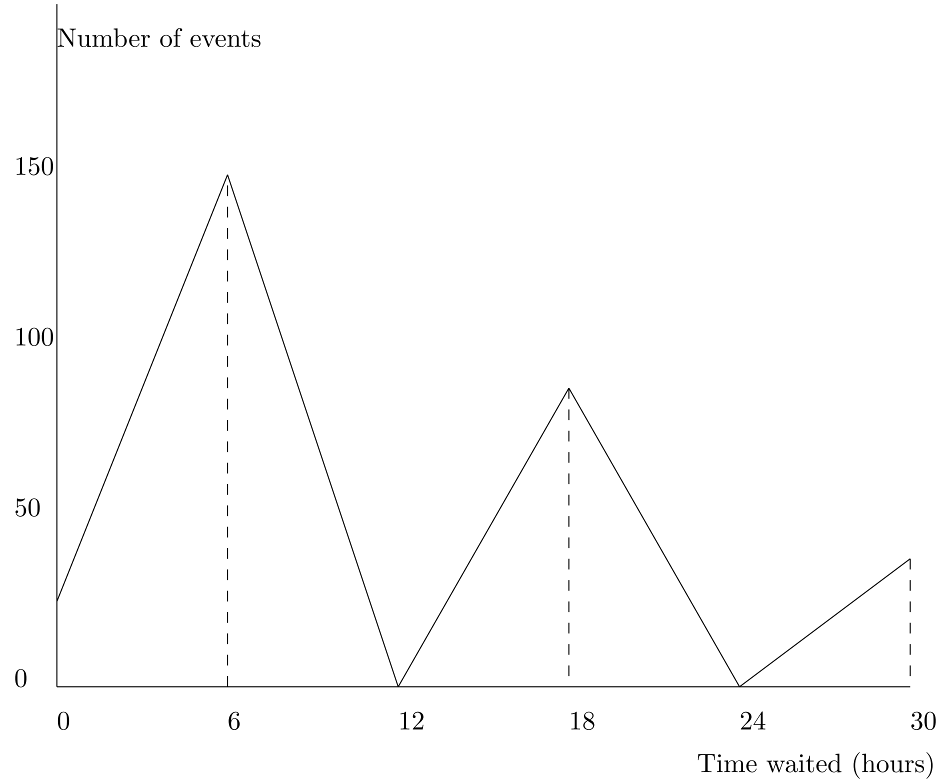


Figure 4.2: Example service time distribution: For each waiting time the amount of events that waited that long is counted.

predicted as the duration of the activity and the predicted exit time is the starting time plus the predicted duration. In the next section, it will be discussed how to, given the possibility to predict queue exit times, predict the sojourn times for some activity.

4.2 Sojourn time prediction

The purpose of this approach is to predict the sojourn time of a case, i.e. predict when a certain case is done, based on the arrival time and, if present, knowledge about the activities that already occurred. Once again a queue collection is the main source of input for this technique. This approach uses two techniques. The first is the queue exit time prediction as discussed in the previous subsection. The second is a way to predict the next event for a known (incomplete) trace.

To allow sojourn time predictions, it is necessary to predict what queue an activity will most likely arrive at next, if any, after exiting some other queue. In the paper by Van der Aalst [41] a number of techniques for creating a transition system useful for time predictions are proposed. An important factor in this approach is using a known set of previously executed activities, a so called *prefix*, to predict what the next activities will be and how long they will take. A number of different abstractions are proposed for creating these prefixes, for example set, bag, sequence, etc. There are a number of different ways to predict the next activity. The example shown in this section was chosen because it strongly utilizes the already present queue collection, but is not necessarily the best approach. A number of interesting alternatives are discussed in the future work section.

To utilize the queue collection model, the queue, context and relative time part of an event will be taken into account in predicting the next activity, in addition to a prefix. This means that every part for which the next activity has to be found will have a relatively small amount of historical data, since a filter takes place on not only queue, context and k, but also the prefix. As such, it seems a good choice to take an abstraction method that does not introduce a large number of unique prefixes. This taken into account, the bag abstraction was chosen.

At some point the end of the trace should be predicted. There are multiple ways to do this. One would be to find a list of ending events and always declare the end of the trace if one such event is executed. An alternative would be that in the case that no historical data is found about the combination of queue, context, k and prefix, the end of the trace should be predicted. The latter was chosen for this technique, but both approaches seem equally valid.

The main goal is to, given some partial trace t where the final event in this trace is e, find the most likely next activity based on: the prefix, i.e. the bag of events that happened before in t, the relative time part of e as extracted by using the technique of Subsection 3.2.2 and the context attribute $ATT_{context}(e)$. This is achieved by considering the set of events S that have the same type, time part, context and prefix as e. Then consider all events that follow after S and count how many times each following activity occurs. The activity that follows most consider the events in S will be the most likely event to happen next.

Now an example will be given to illustrate the concept of temporal and context-based routing. Consider Table 4.1, which illustrates a partial set of values that followed after a certain prefix, time part and context. Say an event arrives for which “Receive package” has been executed, the time part is 1 and the context is “Quiet”. Then there are two possible next events to complete: “Sort package” or “Package lost”. In this case, the event “Sort package” occurred many more times than the alternative, so that type of event will be predicted as the next step. Now consider the same prefix and time part, but with a “Busy” context. Once again the same options are available. This time, however, the count for “Package lost” is higher, so an event with type “Package lost” will be the next step.

Prefix	Time part	Context	Following activity	Amount
...	...	...	...	...
{Receive package}	1	Quiet	Sort package	132
{Receive package}	1	Quiet	Package lost	2
{Receive package}	1	Busy	Sort package	63
{Receive package}	1	Busy	Package lost	94
...	...	...	...	...
{Receive package, Package lost}	1	Busy	__ENDTRACE__	62
{Receive package, Package lost}	2	Busy	__ENDTRACE__	63
...	...	...	...	...

Table 4.1: Partial list of possible nextActCount values.

Now both ingredients for predicting sojourn times are in place: A queue exit time predictor and a temporal and context-based router. There are multiple ways to predict the sojourn time. One is to create a transition system based on either the event log or devise a method to create a more specific transition system based on the queue collection. Alternatively, one could use a step by step method which considers each event in a trace separately and tries to find the next most probable event one at a time. The last option was chosen for this approach.

The following method will be used for predicting sojourn times of some given input trace c, assuming a temporal router as previously described and an exit time predictor as described in the previous section are available.

1. The timestamp of the last event in c is taken as the starting time.
2. The total duration is set to be 0.
3. Consider the last event in $ATT_{trace}(c)$ and call it e.
4. Given the event e and it's time part and context, let the exit time predictor predict the completion time of this event and call this time et. This time will be added to the duration.
5. Given the current prefix, i.e. the bag of all activities in c, the time part and context corresponding to et, use the temporal router to find the most probable next activity. Consolidate the time, context and activity into a new event e.
6. If e is not the end of the trace, return to step 3.
7. Otherwise, the duration so far is the total amount of time the trace will still take. The predicted completion time is the timestamp of the last event in c plus this duration.

Now let's look at an example for sojourn time finding. Once again consider Table 4.1 as an input for the temporal and context-based router and consider Figure 4.2 as an example service time distribution. Consider an event with type "Receive package" that arrives at 27-09-2014, 20:47 and consider Figure 4.2 as its service time distribution. The queue exit time predictor will predict that this event will exit the queue 6 hours after arrival. Now suppose we then get a prefix {"Receive package"}, a context of "Busy" and a time part of 1 (indicating that after 6 hours we arrive in time part 1). The temporal and context-based router will then find, based on Table 4.1 that the most likely next activity is "Package lost". So then the queue exit time predictor will calculate the exit time given "Package lost" as an input at time part 1 and with context "Busy". Since "Package lost" is always the final event in a trace, the predicted duration is 0, as no future event exists to estimate the duration. The following input is then obtained for the temporal context-based router: The prefix is {"Receive package","Package lost"}, the context is "Busy" and the time part is still 1. Considering Table 4.1 the end of the trace is now predicted. The final duration, considering the event with type "Receive package" as a starting point, is 0 + 6 = 6 hours. The predicted sojourn time is then 27-09-2014, 20:47 plus six hours, which results in 28-09-2014, 02:47.

	Queue 1		Queue 2	
Time part	“Quiet”	“Busy”	“Quiet”	“Busy”
1	3	10	12	23
2	12	40	24	35

Table 4.2: Arrival rate values for toy example.

To summarize, given a queue collection as input and given some arriving event, it is possible to predict the sojourn time of that activity based on the context and relative time at which that event arrives. This section and the previous one provided an answer to Research Sub-Question 1.2.3. The next section will cover how to use a queue collection to find the most important bottlenecks in the process.

4.3 Highlighting bottleneck queues in the process

One goal of the queue-based approach is to highlight bottlenecks in the process. This section provides an answer to Research Sub-Question 1.2.1. The queue collection result of the method described in Section 3 can be used as a basis for tackling this problem. It will be covered how to use a queue collection to find the most important bottlenecks, i.e.: the parts of the process that slow down the process the most.

To find bottlenecks a new view on the queue collection is necessary that makes a bottleneck apparent. There are a multitude of ways to gain insight in the bottlenecks, a number of which are discussed in the future work section. This section will propose an example method for finding the workload based on arrival and service rates.

The assumption for finding a bottleneck using arrival rates and service times is as follows: The queues in which completion times are increased when the overall arrival rate is increased, are bottlenecks. The approach to find which queues conform to this is to first define when an overall increase in arrival rate occurs. This is done by creating a ranking of all possible cells within queues of the queue collection, i.e.: Each unique time part and context combination will be ranked descending according to their arrival rate.

Take, for example, a queue collection with 2 queues, for which both there are 2 time parts and 2 contexts and the arrival rate is shown for all of these possibilities in Table 4.2. The arrival rate for context “Busy” and time part 1 is then the sum over all queues for a given context and time, which is 10 + 13 = 23. In this example, time part 2 has a higher arrival rate than part 1 and context “Busy” has, as one would expect, higher workload than context “Quiet”. The arrival rate ranking of these combinations would be {“2;Busy”,“2;Quiet”,“1:Busy”,“1:Quiet”}.

To find the queues for which the average duration increases most in busy periods, it is necessary to look at the behaviour of these queues in the parts of the week and contexts when global workload is high. To achieve this, for each queue the cells within a queue are ranked descending based on their mean service time. Then, some measure of comparison is necessary between the global arrival rate ranking and the service time ranking for a specific queue.

The Kendall-τ [26] method is used to compare the two orderings. Since the Kendall-τ measures a distance between two ranking, the inverse value is used to identify the similarity between the rankings. This ranking is used since it gives a good indication of how the queues mean service times behave in comparison to the global arrival rate. For each two pairs of context and time part values the relative position in the global load ordering and the queue load ordering are compared. If the relative position is similar, it is an indication that the queue has an increase in duration when there is a global increase in arrivals. The sum over all possible pairs is the bottleneck factor of the queue.

	Queue 1		Queue 2	
Time part	“Quiet”	“Busy”	“Quiet”	“Busy”
1	6	20	13	11
2	24	80	11	12

Table 4.3: Mean service time values for toy example.

To give an example of how the bottleneck factor would be calculated, an example set of service time distribution mean values are shown in Table 4.3. The mean service time for a certain queue, time part and context is the corresponding table value. Recall that the ranking for the global workload was {“2;Busy”,“2;Quiet”,“1:Busy”,“1:Quiet”}. Now consider the ranking for Queue 1: {“2;Busy”,“2;Quiet”,“1:Busy”,“1:Quiet”}. This ranking is exactly the same as that of the global workload and hence it obtains a high bottleneck factor. Now consider the ranking for Queue 2: {“1:Quiet”,“2;Busy”,“2;Quiet”,“1:Busy”,}. Without going into the exact Kendall$-\tau$ metric, this ranking is a lot less similar to the global ranking and as such Queue 2 and hence will have a higher Kendall$-\tau$ distance and a lower bottleneck factor than Queue 1.

To conclude this chapter, ranking the queues on similarity with the global workload ordering gives insight in how strong the relation is between an increase in arrivals globally and an increase in duration for a certain queue. The queues that score highest on this scale are estimated to be the biggest bottleneck queues and finding these bottlenecks provides an answer to Research Sub-Question 1.2.1. In the next chapter, the implementation of all techniques covered in this chapter and Chapter 3 will be covered.

Chapter 5

Realization

In this section the realization of the techniques discussed in Chapters 3 and 4 is presented. All realization is done within the ProM framework (see Subsection 2.5.1) and has been implemented as a number of plugins. Throughout this section screenshots, implementation details, links to plugins used and examples will be supplied. All software proposed in this work is available to the public as a ProM package called “QueueMiner”. One can test this software by installing ProM[1] and then adding the “QueueMiner” package from either the nightly builds or, later, a released version.

An overview of the realization approach is shown in Figure 5.1. At the start of the pipeline is an event log, which can come from any source. Before acquiring such an event log, it might be necessary to perform some preprocessing steps. These are discussed in Section 5.1. Then, it is possible to add context information to the event log. How context information is obtained and handled in the event log is discussed in Section 5.2. Next, a queue collection is acquired by using the event log as an input. This process is discussed in Section 5.3. Once a queue collection is acquired, it becomes possible to use it for predictions and other information gaining techniques, which are covered in Section 5.4.

5.1 Log preprocessing

Two topics will be covered in this section: How XES Event Logs are obtained from a number of data sources and how to remove the start-up and cool-down period from an event log.

In some cases an XES Event Log or XMXL file is readily available as an input file and can be directly imported in ProM and used for this approach in the XLog format. In other cases, however, it is necessary to preprocess data into an XES Event Log. Now all the cases relevant for this thesis will be covered.

First of all, to support tests with synthetically generated logs using CPN Tools (See Subsection 2.5.3), it is necessary to use the ProM Import Framework. The ProM Import Framework can use CPN Tools generated files and produce an XES log based on this data. These XES logs are then directly usable. Another possibility is event logs that are only available in a CSV format. ProM provides a standard plugin named “Convert Key/Value Set to Log” that allows for converting .CSV files to the XES Event Log format. With some knowledge of the process involved, fields from the .CSV file can be mapped to fields in the XES format. After using this plugin an XLog is returned which can then be directly used.

[1] http://www.processmining.org/prom/start

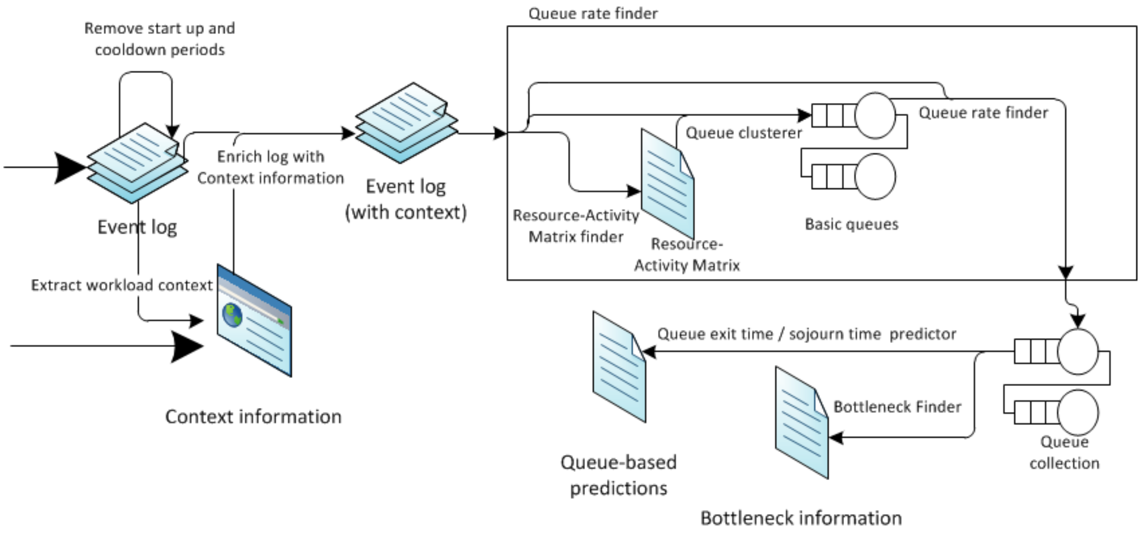


Figure 5.1: Overview of implementation Queue Network finder.

Now it will be discussed how the start-up and cool-down periods can be removed from an event log. Before analyzing a log, it might be necessary to zoom in on the most interesting period within that log. Event logs often contain a start-up and cool-down period in which cases are being started or finished, while the largest portion of the activities are done somewhere in between. To obtain a realistic view on the average situation within a company, it is wise to remove these startup and cool-down periods from the log before analyzing it.

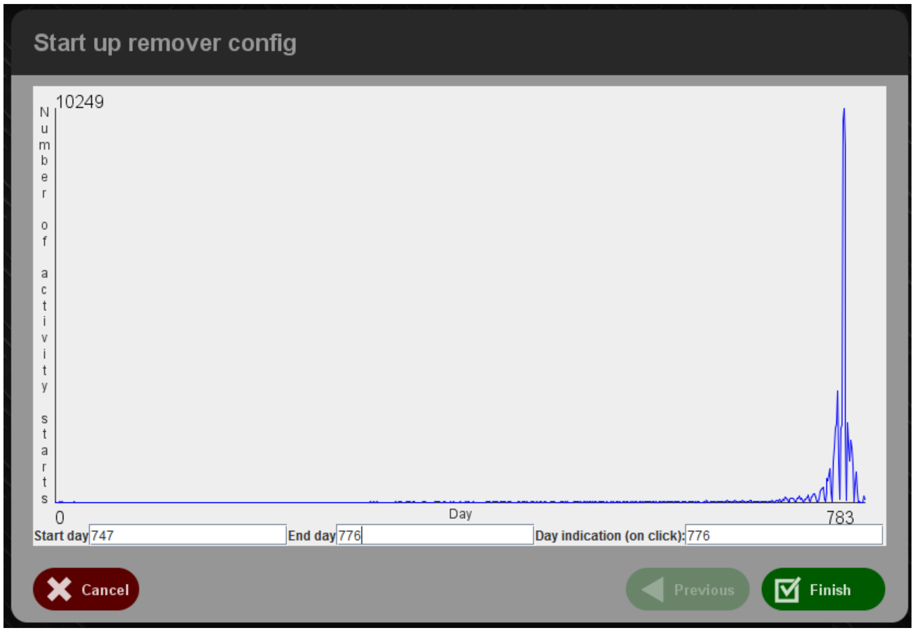
For this reason, the "Start up period remover plugin" (See Appendix B.1) was developed. The concept is straightforward: A user enters a starting and ending day that indicate the start and end of the "interesting" part of the log. Any activities that happen before the starting day or after the ending day are removed and the others remain to form a new log. It is important to note that traces are not removed if some activity falls outside of the scope, because that would result in a new log that has the same problem: a large number of events in the middle with a start up and cool down period. Traces that have only part of their events will exist in the output, possibly missing start or end events. If a trace has no more events after filtering, it is removed.

An example execution of the "Start up remover plugin" is shown in Figure 5.2 which has the BPI 2013 incidents log (as seen in Appendix C.3) as an input. It is obvious that there is a very long period of time at the beginning of the log in which almost no events happen compared to the rest of the log. The peak on the right, however, seems to contain many events. The start up remover plugin allows the user to select the starting and ending day of the peak to remove all the information in the tail. The user can see which days to select by clicking in the graph, which the plugin will respond to by showing the day of the position clicked. In this case the start of the peak is around day 741.

This concludes the preprocessing steps for obtained a decent input XES Event Log file. In the next section all implementation concerning context information is discussed.

5.2 Log context information implementations

This section will discuss everything concerning context information. Five plugins will be shown to support the use of context information in ProM. First, a formal definition for a context information format is given in Subsection 5.2.1. This is supported by three plugins that offer handling context



Start day 747

End day 776

Day indication (on click): 776

Figure 5.2: Start up remover BPI 2013 incidents example. The x-axis of the graph shows the time in days, the y-axis shows the amount of events that were started on that day. The start and end fields indicate the range of days to be used.

information: importing, merging and exporting context information from any source. Context information can come from multiple sources, one of which is a method for extracting workload information from an event log, which is discussed in Subsection 5.2.2. Once again this functionality is supported by a plugin. Workload context information or context information from an external source can then be added to the event log, enriching the log with context information. This is explained in more detail in Subsection 5.2.3 and is implemented using a ProM plugin as well.

5.2.1 Log context information definition

To support context information within logs, as described in Subsection 3.2.3, a new XESExtension (see Subsection 2.5.2 for details on XES extensions) was made, namely the context extension. First the formal description for this context information is given, after which the main functionality is discussed.

The context information extension allows the XEvents of XES Event Logs in ProM to be enriched with a specific context information field that consists of a string, describing the context relevant for that XEvent. In addition, an XML-format was introduced to describe context information files, be it with ProM as a source or any external source.

An example XML context file is shown in Listing 5.1. It describes two different context variables, namely “Weather” and “Workload”. For Weather two context items are present: From the 1st of January 1970 onward the weather was “Sunny”. Then the next entry states that onwards from a little past noon at the 28th of June the weather is “Rainy” and not Sunny anymore. The other variable (Workload) is similar and exists concurrent to the Weather data. In practice there are three points where the global context changes in this example, as illustrated in Table 5.1.

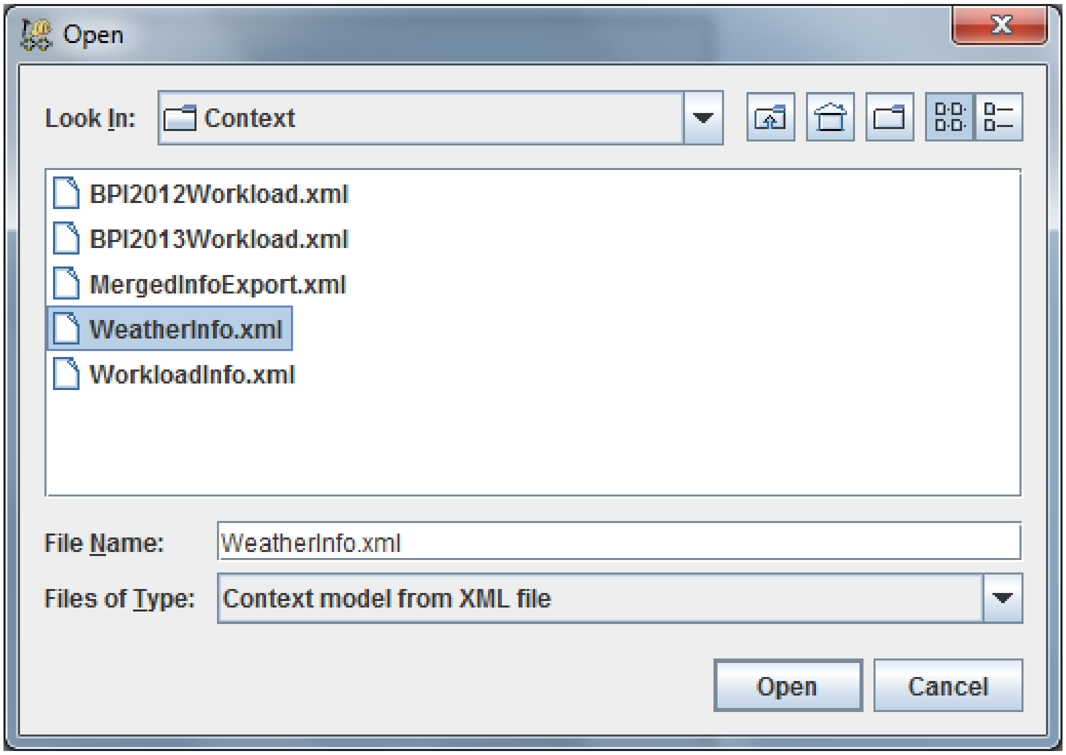
Another fact can be observed from Listing 5.1, namely that it is straightforward to merge context files from multiple sources. In practice, all that is necessary for merging is to take the

Listing 5.1: Context file example

```
<?xml version="1.0" encoding="UTF-8"?>
<contextInfo>
   <contextVariable name="Weather">
      <contextItem>
         <context value="Sunny"/>
         <time value="1970-01-01␣01:00:00"/>
      </contextItem>
      <contextItem>
         <context value="Rainy"/>
         <time value="1970-06-28␣12:41:46"/>
      </contextItem>
   </contextVariable>
   <contextVariable name="Workload">
      <contextItem>
         <context value="Very␣busy"/>
         <time value="1970-03-01␣00:00:00"/>
      </contextItem>
      <contextItem>
         <context value="No␣work␣available"/>
         <time value="1970-09-28␣12:41:46"/>
      </contextItem>
   </contextVariable>
</contextInfo>
```

Time	Before 1970	1970-01-01	1970-06-28	1970-09-28
Weather	none	Sunny	Rainy	Rainy
Workload	none	Very busy	Very busy	No work available

Table 5.1: Global context example



Look In: Context

BPI2012Workload.xml
BPI2013Workload.xml
MergedInfoExport.xml
WeatherInfo.xml
WorkloadInfo.xml

File Name: WeatherInfo.xml

Files of Type: Context model from XML file

Figure 5.3: Example importing context data

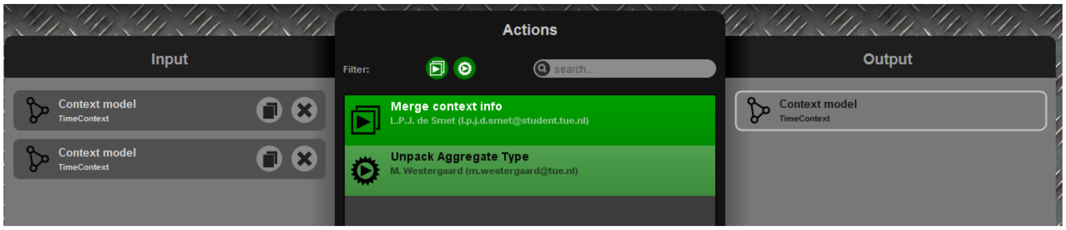


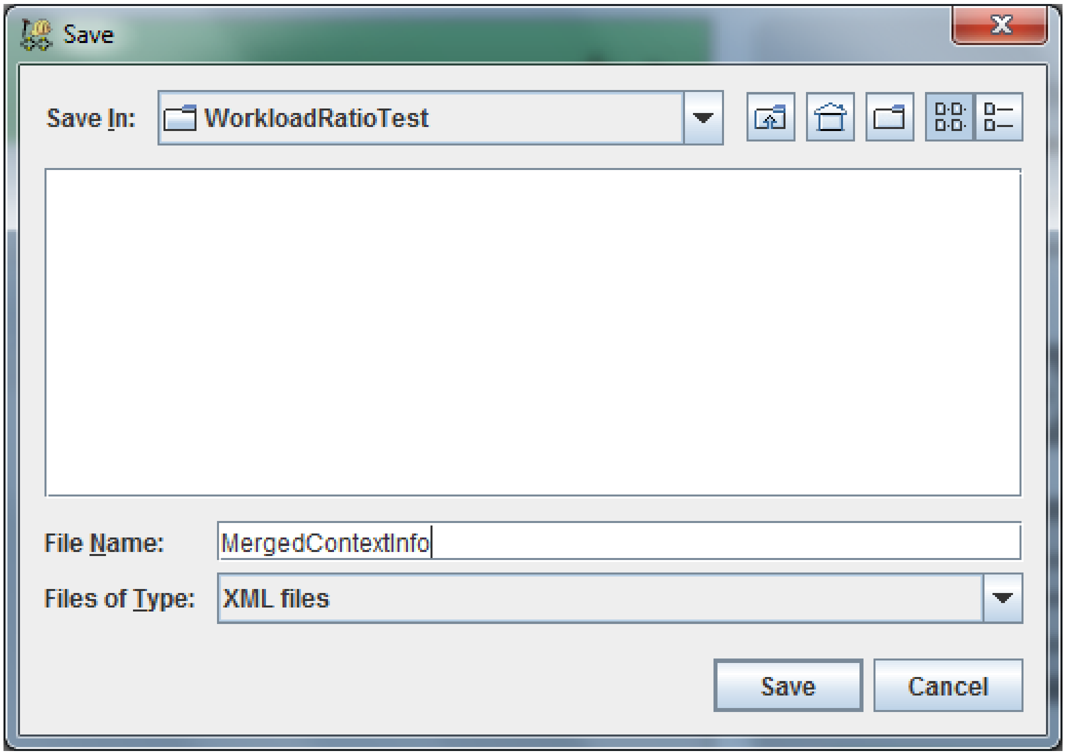
Figure 5.4: Example merging context data

"contextVariable" objects from both input files and concatenate them into an output file. A plugin was made that supports this functionality, which will be covered next.

The main functionality of the context information is based on three plugins. One for importing, one for exporting and one for merging. When context information from an external source is available, it is possible to use the import functionality of ProM on an XML file, selecting "Context model from XML file" as the input method. This is shown in Figure 5.3. After importing a TimeContext object will be present in ProM which can be used for merging, exporting and log enriching. The latter is covered in Subsection 5.2.3.

Merging context information with other context information is done by utilizing the "Merge context info" plugin. This plugin can take two context information TimeContext objects as input and generates a new TimeContext object that has the data of both seperate context information objects. Figure 5.4 shows an example in practice. After pressing start no other input is required.

The final plugin is the export plugin, which is accessed by using the standard Export button in ProM. This plugin is responsible for exporting generates an XML file that conforms to the standard described in Subsection 3.1.2. Figure 5.5 shows an example of exporting a new context



Save In: WorkloadRatioTest

File Name: MergedContextInfo

Files of Type: XML files

Figure 5.5: Example exporting context data.

information file.

5.2.2 Extracting workload context

To demonstrate the idea of enriching a log with context information, it was necessary to first obtain context information. A technique was implemented to extract the workload information of certain points in time from an event log. The abstract description of this technique can be found in Subsection 3.1.1. A number of implementation details will be discussed concerning that technique as well as the main functionality.

First, as mentioned in the description of Subsection 3.1.1, a sliding window moving with a step size of a day is used to decide the workload of a given day. The size of the window in both directions around that day, the day-radius, is set to three. This is because, including the day itself, it is equal to seven days, or: one week. Choosing an entire week for the sliding window makes sure no effects from the lower workload in the weekend have an effect on the final result.

Next, unlike the example provided in Subsection 3.1.1, start up and cool-down periods are not taken into account in practice. The boundary days, i.e.: days that do not have three days before or after them, simply have less days to consider and hence the workload is probably low on those days.

Now some details will be provided on how the sliding window has been implemented. The input for this approach is an Event Log, which in practice corresponds to an XLog from the XES Standard (as seen in Subsection 2.5.2). The XLog format, however, is not very convenient for this approach. For this reason an important internal data-structure based on the XLog data was developed: The Time-Activity Log. The Time-Activity Log (or T-Log) is, as its name suggest, a log consisting entries that specify an activity and the start and end times of that activity, sorted

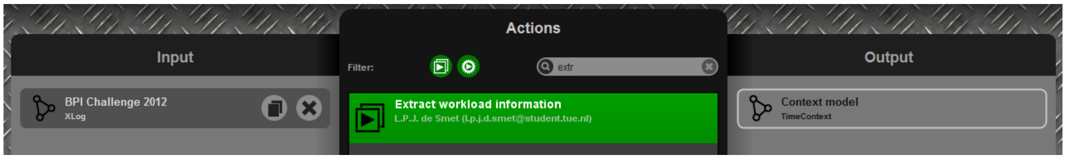


Figure 5.6: Example run of extracting context information from an event log.

ascending by starting time. Additionally, a reference to the XES XEvent is given with each T-Log entry, for the cases where additional information is needed. The time values are saved separately of the XEvent, since it allows for convenient sorting over all activities in a log, which is impossible in the XLog format.

Acquiring a T-Log is done by looking at every XTrace in some XLog. For each XEvent the following is consolidated into a T-Log entry. The time the XEVent starts as the starting time, the time the next XEvent starts as the ending time, the "concept-name" attribute of the XEvent as the activity and finally a reference to the XEvent. In case of the last XEvent in a trace, the starting and ending time are taken to be the same. After looping over all traces, a large list of T-Log entries is obtained. These entries are then sorted ascending by starting time.

Once a T-Log has been acquired, the sliding window is implemented as follows: The current position in the T-Log is kept track of. For each new day, the following two actions are performed: First, it is checked if some activities currently in the window should be removed, because they fall outside the day radius. This is done by checking whether the starting time is within the day radius. Then, from the current position in the T-Log it is checked whether the current activity should be added to the windows or not. If so, it is added and the position is increased by one. If not, the current window is done. This is repeated for each day and for each day the activities in the window are counted. After counting the activities executed on each day, the mean and standard deviation are found and the workload information "Quiet", "Regular" or "Busy" is chosen for each activity, as specified in Subsection 3.1.1. Finally, all T-Log information and the corresponding workload information is saved in the TimeContext format, as introduced at the end of Subsection 5.2.1, where the time corresponds to the T-Log entry starting time and the context information corresponds to the workload information.

Now this technique will be shown in practice. Taking some XES Event Log, is is possible to use the "Extract workload information" plugin to extract the workload for each unique time in the entire log. Figure 5.6 shows an example where context information is extracted from the BPI 2012 challenge log. A partial result of this technique can be seen in Listing 5.2. No further input is required for this plugin and a TimeContext context information object will be generated by the plugin. This object can then be used for merging, exporting and/or enriching an event log with context information. The latter will be covered in the next subsection.

5.2.3 Enriching a log with context information

This section will cover how to enrich an event log with context information from any source that conforms to the format introduced in Subsection 5.2.1.

The details discussed here are based on the methodology to find a context described in Subsection 3.2.3, which is, in short, to find for each context variable the last context value before a certain time. For each XEvent in the input XLog, the time at which the event arrives is extracted. Then, the TimeContext object is asked: What is the context, given this arrival time? In the case that the time is before any of the context items, "null" is returned. In the case that the time is after all context items, the combination of the last context values is returned. For any value in between, an iteration over all times a context changes is executed. If the time falls between any

Listing 5.2: Extracted workload file for BPI2012 log

```
<?xml version="1.0" encoding="UTF-8"?>
<contextInfo>
  <contextVariable name="Workload">
    <contextItem>
      <context value="Quiet"/>
      <time value="2011-10-01␣00:00:00"/>
    </contextItem>
    <contextItem>
      <context value="Quiet"/>
      <time value="2011-10-02␣00:00:00"/>
    </contextItem>
    <contextItem>
      <context value="Quiet"/>
      <time value="2011-10-03␣00:00:00"/>
    </contextItem>
     ...
    <contextItem>
      <context value="Regular"/>
      <time value="2012-02-08␣00:00:00"/>
    </contextItem>
    <contextItem>
      <context value="Regular"/>
      <time value="2012-02-09␣00:00:00"/>
    </contextItem>
    <contextItem>
      <context value="Busy"/>
      <time value="2012-02-10␣00:00:00"/>
    </contextItem>
     ...
    <contextItem>
      <context value="Quiet"/>
      <time value="2012-03-12␣00:00:00"/>
    </contextItem>
  </contextVariable>
</contextInfo>
```

Listing 5.3: Enriched BPI2012 log file

```
<log xes.version=”1.0” xes.features=”nested-attributes”
openxes.version=”1.0RC7” xmlns=”http://www.xes-standard.org/”>
 <trace>
  <string key=”concept:name” value=”173688”/>
  <event>
   <string key=”context:context” value=”Quiet”/>
   <string key=”org:resource” value=”112”/>
   <date key=”time:timestamp” value=”2011-10-01T00:38:44.546+02:00”/>
   <string key=”lifecycle:transition” value=”COMPLETE”/>
   <string key=”concept:name” value=”A_SUBMITTED”/>
  </event>
  <event>
   <string key=”context:context” value=”Quiet”/>
   <string key=”org:resource” value=”112”/>
   <date key=”time:timestamp” value=”2011-10-01T00:38:44.880+02:00”/>
   <string key=”lifecycle:transition” value=”COMPLETE”/>
   <string key=”concept:name” value=”A_PARTLYSUBMITTED”/>
  </event>
   ...
  <event>
   <string key=”context:context” value=”Regular”/>
   <string key=”org:resource” value=”11169”/>
   <date key=”time:timestamp” value=”2012-03-01T09:27:41.325+01:00”/>
   <string key=”lifecycle:transition” value=”COMPLETE”/>
   <string key=”concept:name” value=”W_Afhandelen␣leads”/>
  </event>
 </trace>
</log>
```

of those two changes, the context values before that time are chosen to be the context for that particular time.

Now a practical example will show how to use a TimeContext context information object and an event log to enrich the event log with context information. Figure 5.7 shows an example where the BPI 2012 challenge log and a TimeContext object consisting of the workload information of the BPI 2012 log are used as input. The output is once again an XLog object. For all the activities in this log the context attributes will have been set using the context information. A user will not need to give any input other than the TimeContext objects. A part of the resulting log is shown in Listing 5.3

This concludes the section on context information. Five plugins have been shown to support the use of context information in ProM. The first three consist of handling context information as is: importing, merging and exporting context information from any source. The fourth deals with extracting workload information from an event log, allowing a user to find context information on the load of the system at certain times. The final plugin offers functionality for enriching an event log with context information from any source.

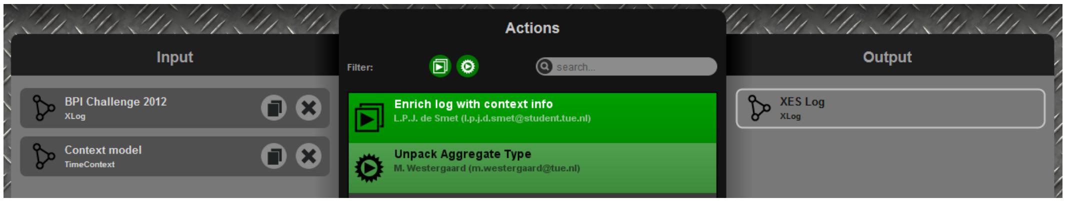


Figure 5.7: Example run of enriching an event log with context information.

5.3 Queue collection finder implementations

This section will discuss how to find a queue collection given an event log. A queue collection is obtained in a number of steps. First, the resource-activity matrix, the matrix indicating how much each resource has done each activity, is extracted from the event log. Then, the resource-activity matrix is used in combination with the event log to find a set of basic queues. The implementation details and plugins used for this are covered in Subsection 5.3.1. Then, to support the technique of enriching basic queues with temporal and context-based service and arrival rate data, it is necessary to find the right number of timeparts a week should be split up in. For this purpose the Week pattern visualizer plugin was developed, which allows users to achieve this. The viewer and the methodology for using it is covered in Subsection 5.3.2. Finally, the event log and the number of parts found with the Week pattern visualizer is used to enrich the basic queues with temporal and context-based data. This is covered in Subsection 5.3.3.

5.3.1 Finding basic queues

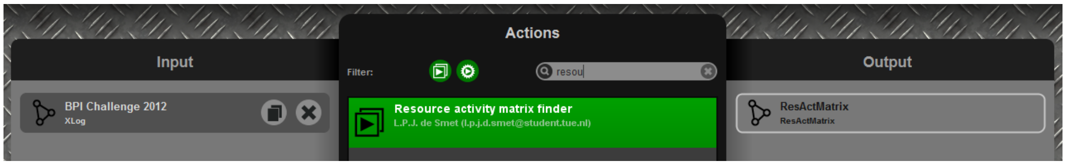
This subsection discusses the implementation details concerning finding basic queues, as introduced in Subsection 3.1. The basic queue finder has been implemented as a series of ProM plugins. Starting with an event log, one first uses the Resource activity matrix finder plugin, which returns a resource-activity matrix. Then, this matrix and the event log are used in the Queue clusterer plugin to obtain a cluster of basic queues. The actions of finding basic queues and finding the corresponding characteristics have also been consolidated in a single Queue rate finder plugin, which can be used directly on an event log. The implementation details of both the Resource activity matrix finder and the Queue clusterer will be shortly discussed in this subsection, while the Queue rate finder will be discussed in Subsection 5.3.3. Additionally, this subsection will show the main functionality of the plugins mentioned.

First, the Resource activity matrix finder details will be discussed. Given an XLog object, the Resource ativity matrix finder will loop over all XTraces and loop over all XEvents in these XTraces. For each XEvent considered, the resource and concept-name attributes will be checked. A list will be kept of all resources and all activities (concept-name attribute) found while looping over all XEvents. Then a matrix is created for which every row indicates an activity and every column indicates a resource. All the XEvents in the input XLog are scanned once more and the matrix cell value for each combination of resource and activity is increased once it is encountered. Finally, a matrix is obtained in which the amount of times each resource executed each activity is stored, which is to be used in the Queue clusterer plugin. Table 5.2 shows partial data for the matrix of the BPI2012 log.

The implementation of the Queue clusterer is based on the technique described in Subsection 3.1. In practice a greedy algorithm was implemented, since a brute force algorithm was too slow. The greedy algorithm has two phases. In the first phase, for each separate activity, a set of resources is found that produces the highest score by the metric introduced in Subsection 3.1. The data used for making these scores is based on the Resource activity matrix discussed in the previous paragraph. Note, however, that the thresholds for $Fair_r$ and $Fair_t$ are multiplied by a

Activity, Resource	112	null	10862	10913	11049	...
A_SUBMITTED	13087	0	0	0	0	...
A_PARTLYSUBMITTED	13087	0	0	0	0	...
A_PREACCEPTED	4852	0	42	77	0	...
W_Completeren aanvraag	4853	6478	0	1515	31	...
A_ACCEPTED	0	0	51	166	0	...
O_SELECTED	0	0	80	222	10	...
...	...	...	...	...	...	...

Table 5.2: BPI 2012 log partial resource activity matrix.



resou

BPI Challenge 2012
XLog

Resource activity matrix finder
L.P.J. de Smet (l.p.j.d.smet@student.tue.nl)

ResActMatrix
ResActMatrix

Figure 5.8: Example run of Resource activity matrix finder using an event log.

factor 0.75, to make the selection criteria less strict. This was necessary since in practice many intuitively good choices did not meet the demand of $Fair_r$ or $Fair_t$.

After phase one, we have a list of all activities with a corresponding list of best resources. For phase two, all activities are put into an ordered list in order of appearance in the original log. While the ordered list is not empty, the next activity is taken out. Then the score for not merging any other activities and the scores for merging any activities still in the list will be calculated. While the score for merging remains higher than that of not merging, activities will keep being added. Once not merging has the highest scores, the current set of activities and corresponding resources is saved and removed from the list. This is repeated until the ordered list is empty and a clustering of activities and resources is found. Figure 5.9 shows an example execution. There is no visualizer for the resulting set of basic queues, so a portion of the raw data has been transcribed and can be seen in Table 5.3. This set of basic queues is then saved to be used in the Queue rate finder plugin, which is discussed in Subsection 5.3.3.

5.3.2 Week pattern visualizer

A user has to provide input for the Queue rate finder, which is the amount of parts a day should be split up in: k. It is, however, not immediately obvious what this value should be. With this in mind the Week pattern visualizer was developed, which is an aid for users to find the right value for this variable such that there is no overfitting and no underfitting in all the subparts following from splitting every day of the week into k parts, resulting in a total of $7 \cdot k$ parts. The main

Queue nr.	Activities	Resources
1	A_SUBMITTED, A_PARTLYSUBMITTED	112
2	W_Completeren aanvraag, W_Nabellen offertes, W_Afhandelen leads	112, null, 10913, 11201, 11119,...
3	A_PREACCEPTED	112, 10982, 11169, 10910
4	W_Nabellen incomplete dossiers	null, 10913, 11049, 10629, 10809,...
5	W_Valideren aanvraag	null, 11049, 10629, 10809, 10609,...
...	...	...

Table 5.3: BPI2012 log partial basic queue clustering.

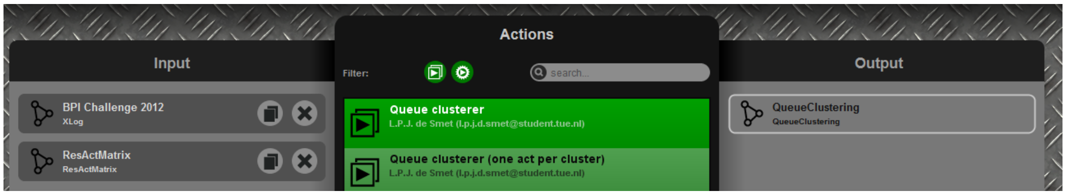

Input
BPI Challenge 2012
XLog
ResActMatrix
ResActMatrix
Actions
Filter:
search...
Queue clusterer
L.P.J. de Smet (l.p.j.d.smet@student.tue.nl)
Queue clusterer (one act per cluster)
L.P.J. de Smet (l.p.j.d.smet@student.tue.nl)
Output
QueueClustering
QueueClustering

Figure 5.9: Example run of Queue clusterer using an event log and Resource activity matrix.

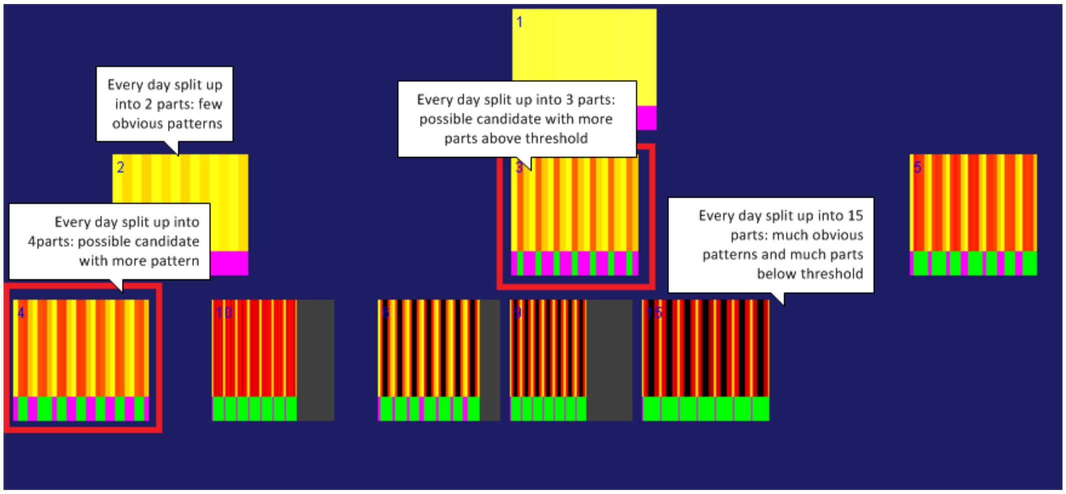


Figure 5.10: Prime tree traversal with good candidates highlighted. On the x-axis of each node are the time parts of the week. The y-axis is split up in two parts. The top part uses a black-body color-map to indicate the amount of activities occurring in each time part. The bottom part is used for indicating which time parts have a significant amount of samples. Green parts are below threshold, purple parts above.

functionality and methodology of using the week pattern visualizer is discussed in this subsection.

The underlying structure for the week pattern visualizer is the PrimeTree, which was discussed in Subsection 3.2.2. Each node in the PrimeTree represents a value k implying the amount of parts the log should be split up in is $7 \cdot k$. The Week pattern visualizer has multiple options for comparing and exploring possible values of k and showing the corresponding pattern and the amount of overfitting that occurs, i.e. the parts for which the size falls below a threshold. The threshold value is 100, since that is enough for most statistical tests and in particular it is enough to do good predictions with the resulting model.

An example of such an explored PrimeTree is shown in Figure 5.10. In the nodes the green and purple parts indicate whether a node part is below (green) or above (purple) the threshold. The nodes 10, 6, 9 and 15 obviously have a too high fraction of parts that fall below the threshold. Nodes 3 and 4 seem to show a patterns that have enough detail. This view can be obtained on any XES Event Log (or XLog) by selecting the PrimeLogTree Viewer when visualizing an event log using ProM's visualization function, as shown in Figure 5.11.

Now the methodology to find the right value for k will be discussed, in combination with showing the capabilities of the PrimeLogTree Viewer plugin. The first step is to explore the tree from the root to see which paths provide interesting patterns. The screenshot in Figure 5.12 shows an example tree of the BPI Challenge 2012 log as described in Appendix C.2 which was partially explored. Exploring the tree is done by clicking on individual nodes in the tree. Once a node is clicked, it expands into all possible children. In this case the product of the node value with

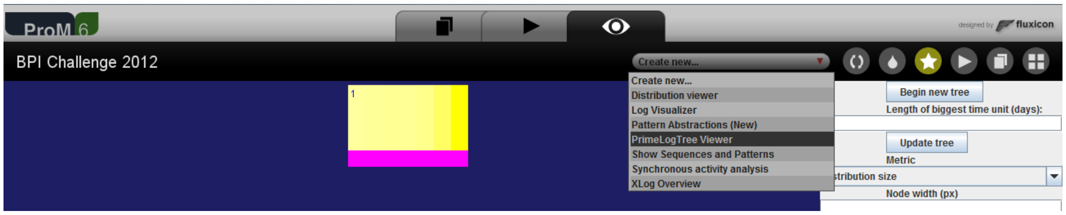


Figure 5.11: Example of starting a new PrimeLogTree Viewer instance.

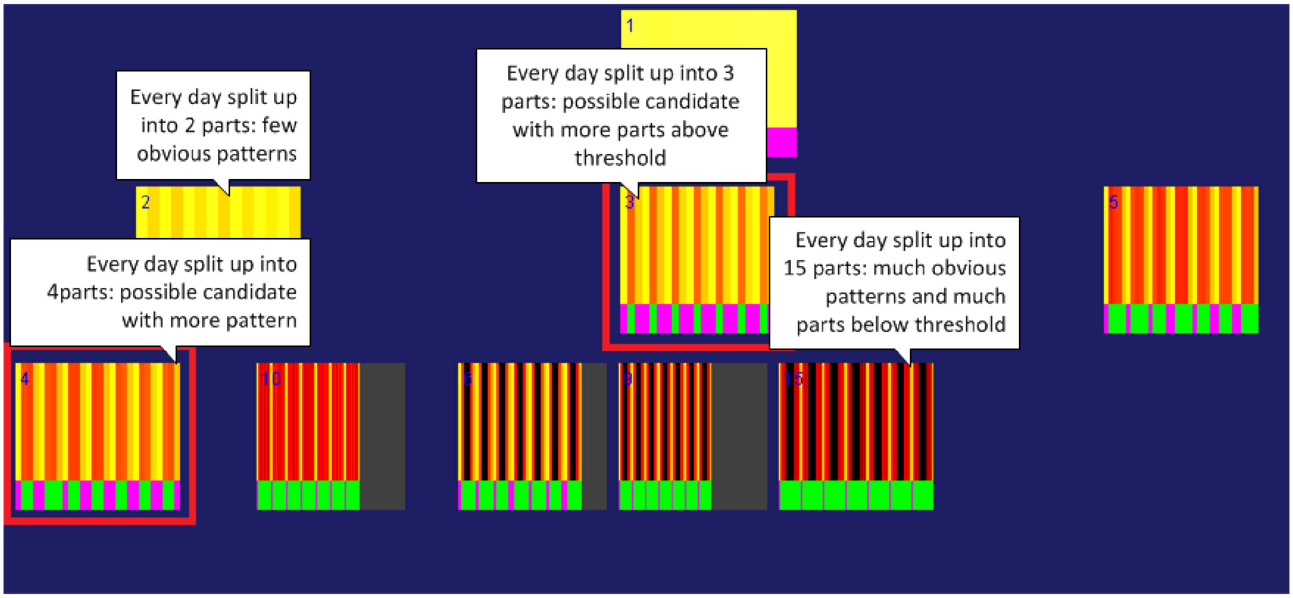
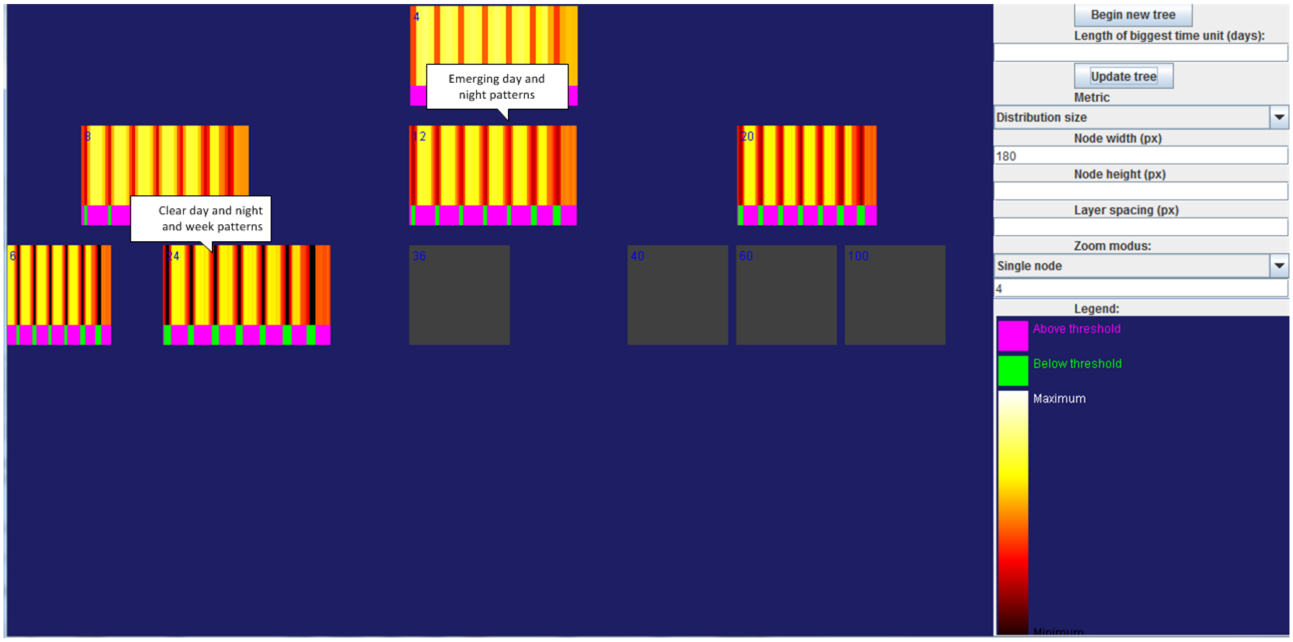


Figure 5.12: Visualization of partially traversed Primetree of BPI Challenge 2012 log. On the x-axis of each node are the time parts of the week. The y-axis is split up in two parts. The top part uses a black-body color-map to indicate the amount of activities occurring in each time part. The bottom part is used for indicating which time parts have a significant amount of samples. Green parts are below threshold, purple parts above.

the primes two, three and five. The log considered was generated by a system that does most activities in a normal work week. In this picture it is not immediately clear what the right way to go will be, but it is apparent that the branch of 5 (including 10 and 15) does not have very promising patterns, since it does not seem to have any similarity with a work week.

The second step is zooming in on branches of the tree that seem interesting. The user can choose to zoom in on a certain node, for example 4. Zooming in is achieved by selecting the "Single node" zoom modus and entering the number of the node that should be zoomed on below. Then the "Update tree" button should be pressed. Figure 5.13 shows the tree zoomed in on node 4 (indicating a total of $4 \cdot 7 = 28$ parts in the week). Here one can see that the nodes 24, 12 shows a promising result, but in this view it is difficult to compare exactly how the patterns relate. The grey nodes are nodes for which the pattern cannot be visualized in the space given without becoming imprecise. If knowledge of the pattern of these nodes is deemed relevant, one can zoom in on them, or, alternatively, zoom in on a set of nodes to compare them.

The final step is to compare a number of nodes. Comparing a set of nodes is achieved by selecting the "Node set" zoom option and entering a set of node numbers, separated by comma's. Then the "Update tree" button should be pressed. Let's say the user has chosen 12, 24 and 36 as possible candidates, based on his knowledge of the system. Figure 5.14 shows a comparison of



Length of biggest time unit (days):

Metric: Distribution size

Node width (px): 180

Node height (px):

Layer spacing (px):

Zoom modus: Single node

4

Figure 5.13: Primetree visualization of BPI Challenge 2012 log zoomed in on node 4. This is the same data as for Figure 5.12, but taken the node with value four as a root.

the nodes 12, 24 and 36. Although 24 and 36 have a larger amount of parts below the threshold than 12, it seems that 12 does not show enough of a pattern. Between 24 and 36 the choice is rather difficult. Both have about the same pattern and about the same amount of parts below the threshold. The user, knowing a normal workday in this process is 24 hours, would choose 24 over 36.

The value of k found is the one that can be used when utilizing the Queue rate finder plugin to find temporal and context-based data, which is discussed in the next subsection. Before that, the other functionality of the week pattern visualizer is touched upon. Consider Figure 5.13 and observe the options menu on the right. At this point visualization has been done using the distribution size as the indicator for patterns. However, one can use other "Metric" values, namely: The distribution mean or distribution variance for finding patterns. These patterns are mostly very similar to that of the distribution size, especially for the finer grained results. In addition to changing the pattern metric, it is possible to tweak the sizes of nodes and the space between nodes, to find a better picture of the tree, e.g.: fitting more nodes on the screen or making nodes larger to have a better understanding of their pattern.

At the very top of the options a field named "Length of the biggest time unit(days)" is available. For this approach the biggest time unit is a week, as has been argued in the text around Definition 3.2.1. For experimental purposes, however, one might be interested to see patterns not bound to a week, but instead for example a day or a month. When changing this option it is necessary to use the "Begin new tree" button for the changes to take effect. This is necessary since the structure of the log parts is very different depending on the biggest time unit. The next subsection will discuss how the value k found using the week pattern visualizer is used in the Queue rate finder.

5.3.3 Finding a queue collection

The previous subsections focused on how to obtain a set of basic queues and how to find the value k indicating the amount of parts a weekday should be split up in to do good predictions. The input

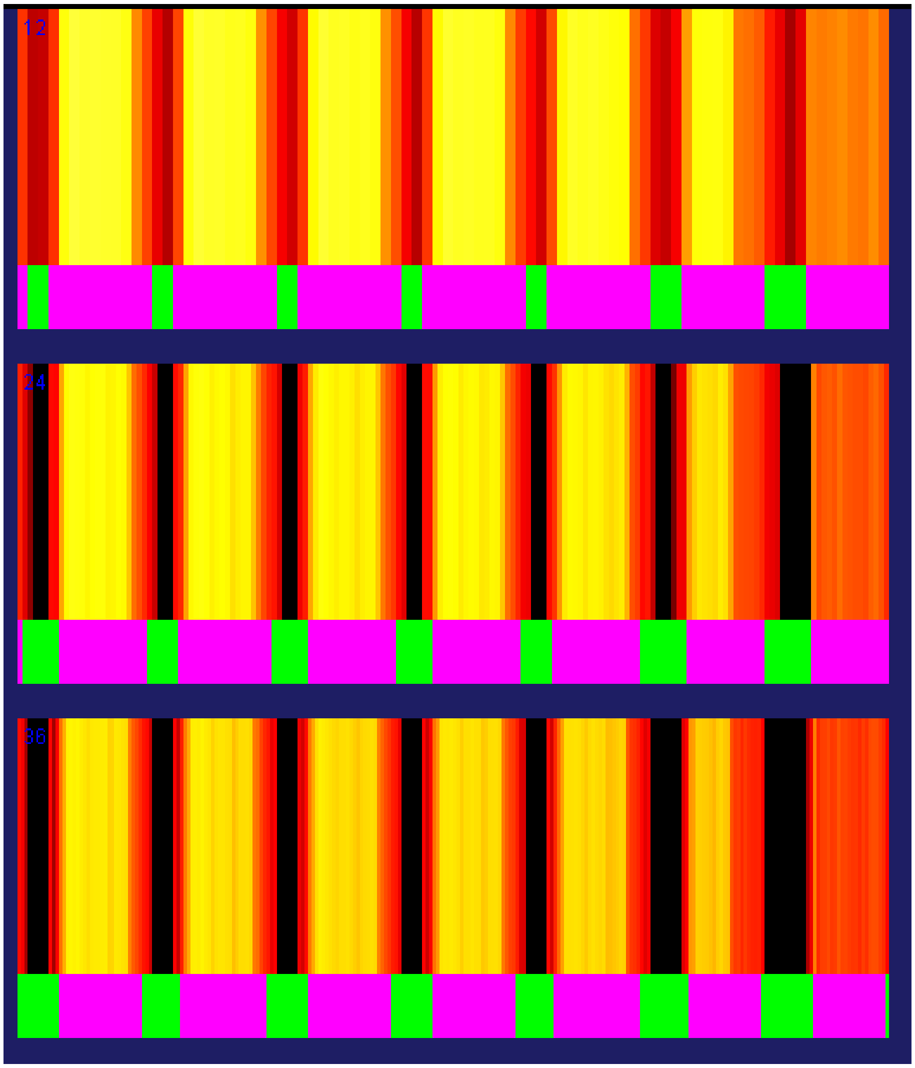


Figure 5.14: Primetree visualization of BPI Challenge 2012 log zoomed in on nodes 12, 24 and 36.

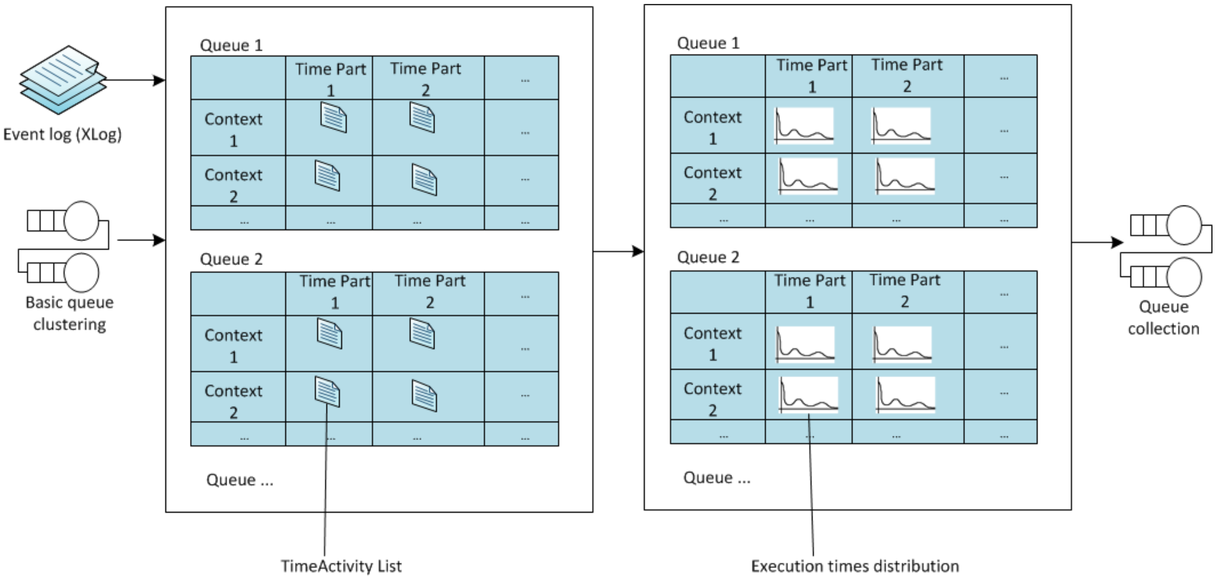


Figure 5.15: Schematic overview of how queue characteristics are found. Time parts indicate the relative time part in the week, e.g. the first hour of the week could be time part 1. Context indicates a unique context in which data can arrive. Each cell contains data that arrived in the corresponding relative time part and context.

for this method is an event log, a QueueClustering object corresponding to that event log, similar to the one presented in Table 5.3 and a variable k indicating the amount of parts a day should be split up in time-wise. This subsection will elaborate on how this set of basic queues and value k are used and the implementation details of the ProM plugin that finds the queue characteristics, namely the “Queue rate finder” are discussed. This plugin implements the concepts discussed in Section 3.2 and the implementation details of the “Queue rate finder” will be discussed as well. The output of this technique is a queue collection. Figure 5.15 shows a schematic overview of the technique that is now discussed.

First, the data inside the event log and QueueClustering object is split up into a number of smaller parts. The QueueClustering and the event log are used to split up all XEvents within the XLog based on: The queue they belong to, the relative part of the week they arrive (one of $7 \cdot k$ parts) and the context of the XEvent. So, for example, if $k = 24$ and an activity arrives on 29-09-2014 at 3:01, which is a monday, and the context at this time is “Busy”, then this activity will fall into the cell with time part 4 (since 3:01 is in the fourth hour of the week) and context “Busy”. All the XEvents are converted into TimeActivity objects, which are, as specified in Subsection 5.2.2, objects that contain the starting time, ending time, event type of the XEvent and a reference to the XEvent itself. These TimeActivity objects are put in lists which are put into a big Hashmap, based on their queue, relative part of the week and context, as in the left side of Figure 5.15.

Then, in the second phase, a loop is done over all elements of the big Hashmap mentioned in the previous paragraph. Every bottom element is a list of TimeActivity objects. For every bottom element, a service time distribution is found using the execution times (ending time minus starting time) of all the elements in list of TimeActivity objects. The values in these distributions will be binned with bin sizes equal to the time part sized. For example, if $k = 24$, there will be a bin for every hour. All these distributions are put into another Hashmap, also based on queue, relative part of week and context. The right part of Figure 5.15 shows the idea for this structure.

Apart from the service time distributions, information about the arrival rate is saved. For each Queue and each relative week part and context within this queue, the total number of activities is

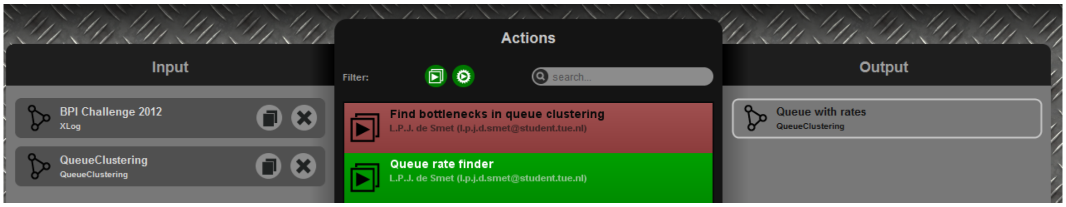


Figure 5.16: Starting screen for Queue rate finder with QueueClustering object.

counted. Then, the total number of weeks that was considered in the original event log is counted. The arrival rate of that particular week part, context and queue is then the number of activities divided by the number of weeks.

Next, the Hashmap is converted to a QueueClustering object by taking the Hashmap entry for a given queue and adding this info to a Queue object. In practice, this means that for each Queue there is a matrix of distributions corresponding to all relative week parts and contexts and an arrival rate number for each of these as well. All these Queues together form the queue collection that is produced as an output and that can be used for predictions and gaining insight.

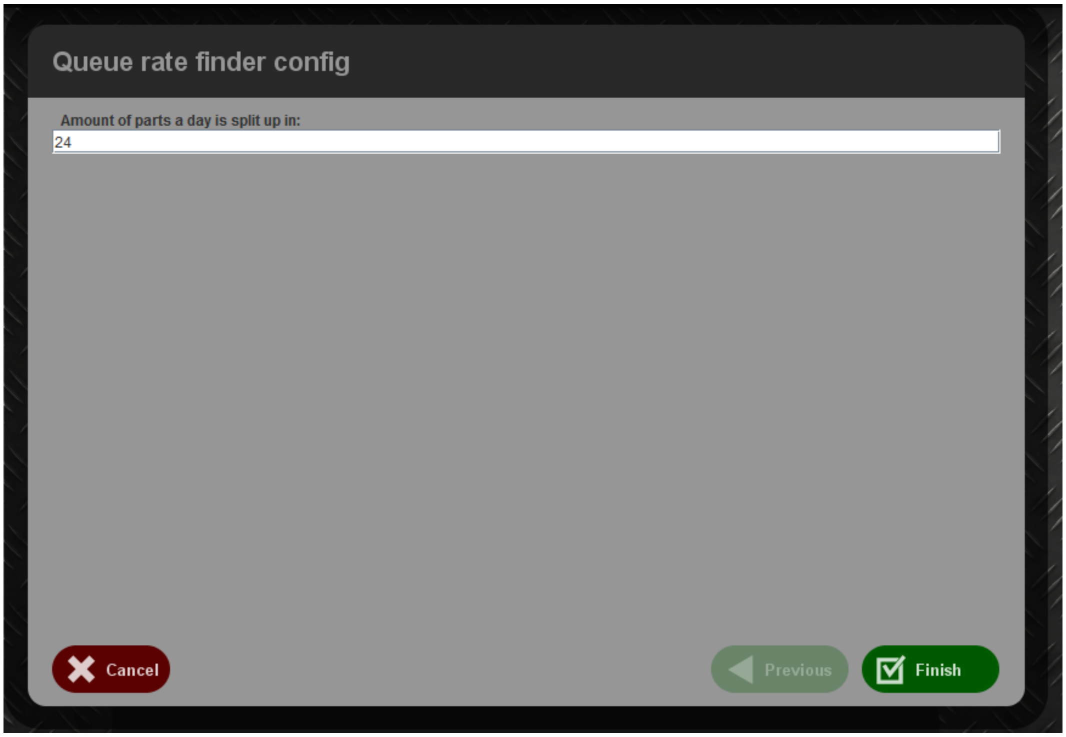
Now an example run of the Queue rate finder will be illustrated. First, the user should select an event log and a QueueClustering object corresponding to that event log. In this case the BPI 2012 challenge log was used. Figure 5.16 shows the way for starting the plugin and the resulting QueueClustering object. Once the start button is pressed, the user is prompted to enter the value for k, as seen in Figure 5.17. For the BPI 2012 log, 24 is a good value for k. After entering this value, the plugin does all the other work and produces a queue collection. A queue collection result for the BPI 2012 log is difficult to visualize, as it consists of a large number of service rate distributions and arrival rate numbers. To illustrate the produced result, Figure 5.18 shows the results for a single cell of a single queue. The figure shows the activities and resources that belong to this queue (information obtained from the basic queues) and shows the arrival rate and queue sojourn duration distribution for this queue in the context "Regular" and in relative time part 9.

5.4 Queue collection prediction implementations

This section will cover the implementation details and main functionality of the plugins that support the techniques introduced in Chapter 4, namely: making predictions for queues and finding the most important bottleneck queues. Predicting queue exit times is covered in Subsection 5.4.1. Predicting sojourn times is covered in Subsection 5.4.2 and finding the most important bottleneck queues is discussed in Subsection 5.4.3. All the individual pieces of implementation might be interesting from a research perspective, but are not so obviously useful from an analyst or any other ProM users's perspective. So, to allow end users, e.g.: analysts or managers, to easily use the functionality provided by the plugins mentioned in this section, a few extra all-in-one plugins were developed. Each subsection includes a short description of their corresponding all-in-one plugin.

5.4.1 Queue exit time predictor

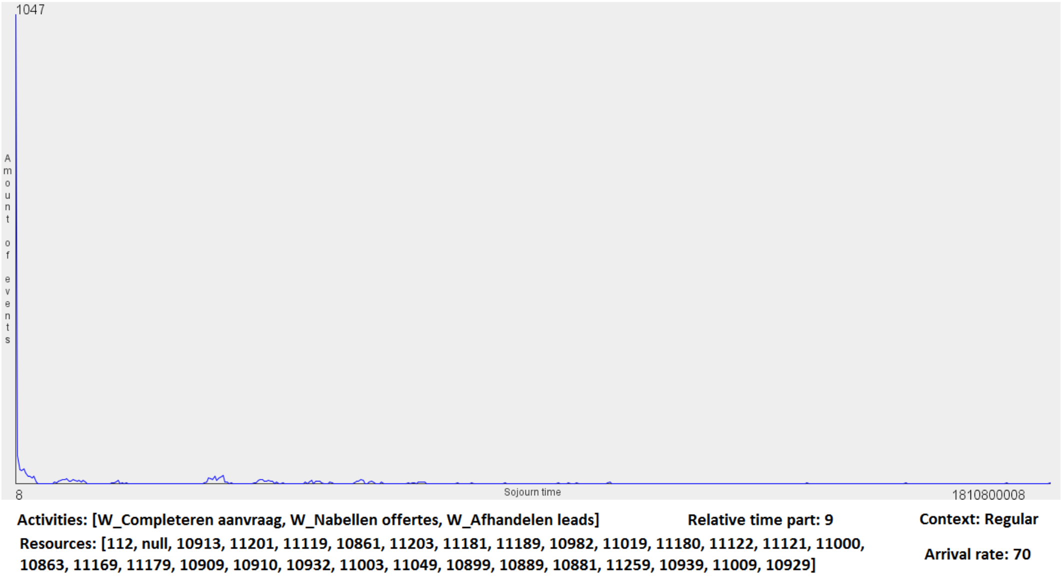
The queue exit time predictor, as discussed in Subsection 4.1, has been implemented through the "Make exit time predictions based on a queueing model" plugin for practical use. First, the workings of the Queue exit time predictor are discussed and then it is discussed how the predictor fits within the validation plugin and in the practical plugin.



Queue rate finder config

Amount of parts a day is split up in: 24

Figure 5.17: Input screen for Queue rate finder.



Activities: [W_Completeren aanvraag, W_Nabellen offertes, W_Afhandelen leads]

Relative time part: 9

Context: Regular

Resources: [112, null, 10913, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11019, 11180, 11122, 11121, 11000, 10863, 11169, 11179, 10909, 10910, 10932, 11003, 11049, 10899, 10889, 10881, 11259, 10939, 11009, 10929]

Arrival rate: 70

Figure 5.18: Partial result for BPI 2012 log queue collection. For a single queue, a single cell for time part 9 and context "Regular" the queue sojourn time distribution and arrival rate are shown.

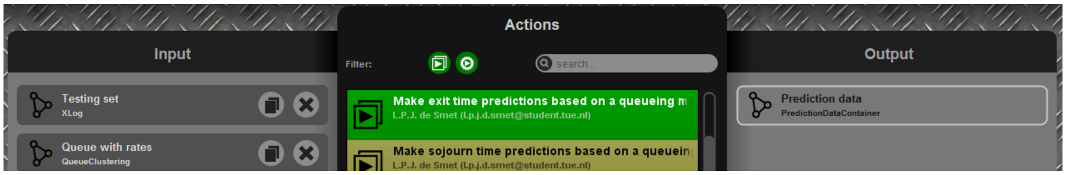
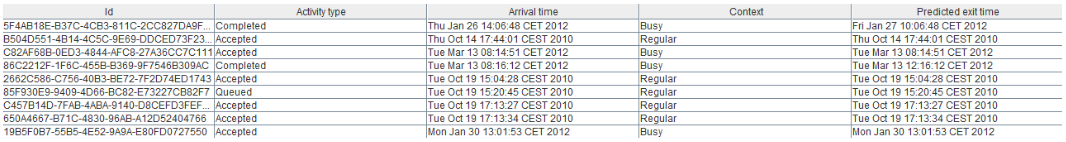

Actions
Input
Filter:
search...
Output
Testing set
XLog
Queue with rates
QueueClustering
Make exit time predictions based on a queueing m
L.P.J. de Smet (l.p.j.d.smet@student.tue.nl)
Make sojourn time predictions based on a queuein
L.P.J. de Smet (l.p.j.d.smet@student.tue.nl)
Prediction data
PredictionDataContainer

Figure 5.19: Example use of the exit time predictor plugin.



Id	Activity type	Arrival time	Context	Predicted exit time
5F4AB18E-B37C-4CB3-811C-2CC827DA9F...	Completed	Thu Jan 26 14:06:48 CET 2012	Busy	Fri Jan 27 10:06:48 CET 2012
B504D551-4B14-4C5C-9E69-DDCED73F23...	Accepted	Thu Oct 14 17:44:01 CEST 2010	Regular	Thu Oct 14 17:44:01 CEST 2010
C82AF68B-0ED3-4844-AFC8-27A36CC7C111	Accepted	Tue Mar 13 08:14:51 CET 2012	Busy	Tue Mar 13 08:14:51 CET 2012
86C2212F-1F6C-455B-B369-9F7546B309AC	Completed	Tue Mar 13 08:16:12 CET 2012	Busy	Tue Mar 13 12:16:12 CET 2012
2662C586-C756-40B3-BE72-7F2D74ED1743	Accepted	Tue Oct 19 15:04:28 CEST 2010	Regular	Tue Oct 19 15:04:28 CEST 2010
85F930E9-9409-4D66-BC82-E73227CB82F7	Queued	Tue Oct 19 15:20:45 CEST 2010	Regular	Tue Oct 19 15:20:45 CEST 2010
C457B14D-7FAB-4ABA-9140-D8CEFD3FEF...	Accepted	Tue Oct 19 17:13:27 CEST 2010	Regular	Tue Oct 19 17:13:27 CEST 2010
650A4667-B71C-4830-96AB-A12D52404766	Accepted	Tue Oct 19 17:13:34 CEST 2010	Regular	Tue Oct 19 17:13:34 CEST 2010
19B5F0B7-55B5-4E52-9A9A-E80FD0727550	Accepted	Mon Jan 30 13:01:53 CET 2012	Busy	Mon Jan 30 13:01:53 CET 2012

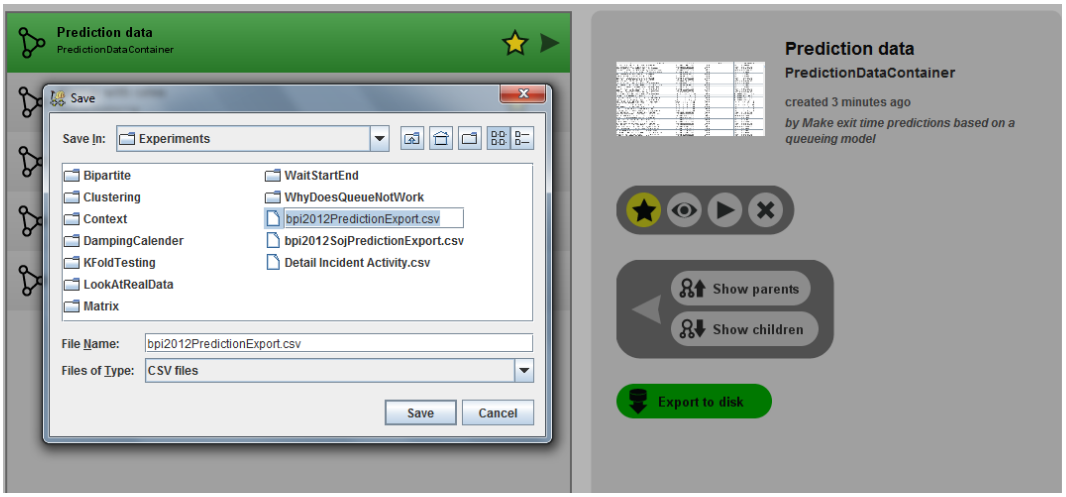
Figure 5.20: Visualization of results of exit time predictor plugin for the BPI2013 log. Note that some of the predicted values are the same as the arriving times. This is due to activities that take shorter than one time part to complete while the distributions are binned according to the size of a time part, e.g. one hour for $k = 24$.

To understand the queue predictor it is necessary to look at the queue collection format once more. As mentioned in Subsection 5.3.3, a queue collection is a list of queue objects. Each queue object contains a map with data split on basis of: 1. their relative arrival time in a week (e.g. the second hour of a Tuesday), 2. the context at the absolute time of arrival (e.g. "Busy"). For each combination of a relative time part and a context, the service time distribution and arrival rate have been stored. Say that a queue predictor gets some new arriving activity as input, in which the activity, the relative arrival time and context of the activity are known. First, the right queue is found, i.e.: the queue for which the activity is in the list of activities belonging to that queue. Since the queues are not stored with activities as an index, a loop is done over all queues and the queue that contains the right activity is taken.

Once the right queue is found, the data-structure inside the queue is used for finding the service time distribution conforming to: 1. the relative time part and 2. the context of the arriving activity. To find the relative time of an arriving activity, it is necessary to convert the absolute arrival time by applying a modulo, i.e.: the relative arrival time is the absolute arrival time modulo the amount of time in a week. To find the week part corresponding to this relative arrival time is then . Once the service time distribution is obtained, the predicted duration is found by extracting the most likely value from the distribution. The predicted exit time of the arriving event is then the absolute arriving time of the event plus the predicted duration.

The practical plugin, named the "Make exit time predictions based on a queueing model", can be found in ProM as seen in Figure 5.19. Internally, the plugin uses a queue collection and predicts the time for all events in a test set. The result is then shown in a visualizer, as shown in Figure 5.20, as a list of predicted exit times. This result can also be exported to a CSV file using an export plugin, accessed by the standard ProM export button. Figure 5.21 shows an example of this.

In addition to a plugin based on a queue collection and testing set, an all-in-one version was developed for end users. To start the all-in-one queue exit time predictor all that is needed is a training log and testing log. Figure 5.22 shows an example start of the plugin. The plugin will result in a set of prediction data which can be visualized and exported, as previously discussed.



Save In: Experiments

File Name: bpi2012PredictionExport.csv

Files of Type: CSV files

Figure 5.21: Exporting results of exit time predictor plugin.

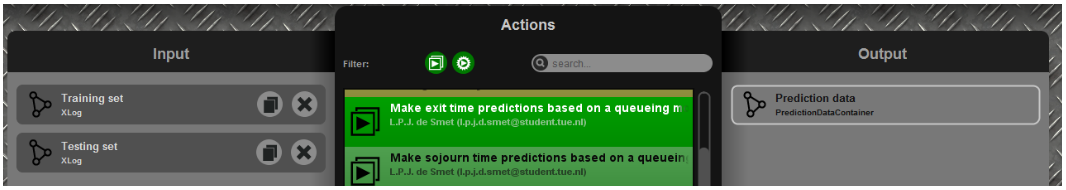

Input
Training set
XLog
Testing set
XLog
Actions
Filter:
search...
Make exit time predictions based on a queueing m
L.P.J. de Smet (l.p.j.d.smet@student.tue.nl)
Make sojourn time predictions based on a queuein
L.P.J. de Smet (l.p.j.d.smet@student.tue.nl)
Output
Prediction data
PredictionDataContainer

Figure 5.22: Example use of all-in-one queue exit time predictor plugin.

5.4.2 Sojourn time predictor

The implementation of the sojourn time predictor, as discussed in Section 2.4.2, consists of two subparts. The first is the queue predictor that was discussed in the previous subsection and Section 4.1. The other is the temporal and context-based process routing that was discussed in Section 2.4.2. First, the implementation details of the temporal and context-based router will be discussed. Then, the implementation of the sojourn time predictor itself will be explained. Finally, a practical prediction plugin where the predictor is used will be mentioned.

A temporal router is created and trained by using an XLog and value k indicating the amount of parts a week is split up in as input. Training the temporal router is done in two phases. First, for each XEvent in each XTrace in the input XLog, the following is counted: Given some prefix of activities, the last type occurring in a certain context, and in a certain part of the week, count the amount of times each activity follows given that prefix, context and part of the week. If no activity follows in the trace, the end of the trace is counted by using a constant "__END__". For example, if there are 5 occurrences of the activity "Package lost" following after activity "Sort package" in the context of "Busy" and the time part being a monday morning, the count for that combination would be 5. The count is done for each possible combination. The second phase is to look, for each prefix, in each context, in each part of the week, what was the following activity with the highest count? A loop is done over all possibilities and the activity with the highest count is stored. This means one obtains a HashMap in which the keys of activity prefix, context and time of the week part, one can obtain the most likely next activity.

The sojourn predictor builds upon the queue predictor and the temporal router for its results. To instantiate the sojourn predictor, an XLog training set, XLog testing set and queuing network based on the training set are required. A temporal router and context object are created using the training set on instantiation. To predict the sojourn time, the following parameters should be

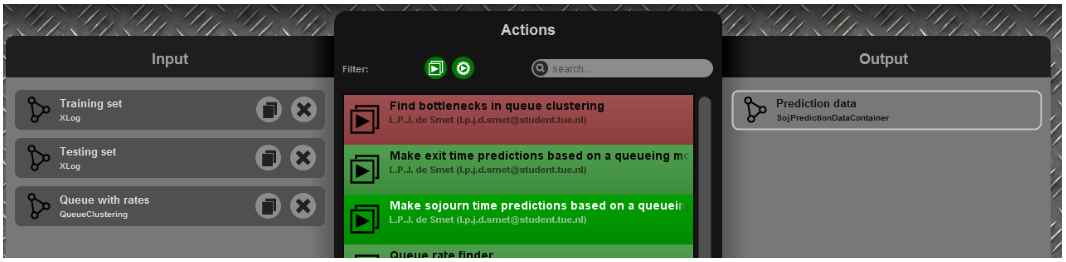
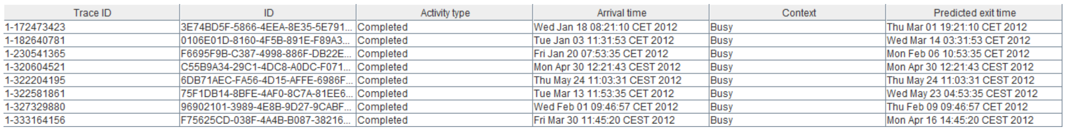

Input
Training set
XLog
Testing set
XLog
Queue with rates
QueueClustering
Actions
Filter:
search...
Find bottlenecks in queue clustering
L.P.J. de Smet (l.p.j.d.smet@student.tue.nl)
Make exit time predictions based on a queueing m
L.P.J. de Smet (l.p.j.d.smet@student.tue.nl)
Make sojourn time predictions based on a queuei
L.P.J. de Smet (l.p.j.d.smet@student.tue.nl)
Queue rate finder
Output
Prediction data
SojPredictionDataContainer

Figure 5.23: Example use of the sojourn time predictor plugin.



Trace ID	ID	Activity type	Arrival time	Context	Predicted exit time
1-172473423	3E74BD5F-5866-4EEA-8E35-5E791...	Completed	Wed Jan 18 08:21:10 CET 2012	Busy	Thu Mar 01 19:21:10 CET 2012
1-182640781	0106E01D-8160-4F5B-891E-F89A3...	Completed	Tue Jan 03 11:31:53 CET 2012	Busy	Wed Mar 14 03:31:53 CET 2012
1-230541365	F6695F9B-C387-4998-886F-DB22E...	Completed	Fri Jan 20 07:53:35 CET 2012	Busy	Mon Feb 06 10:53:35 CET 2012
1-320604521	C55B9A34-29C1-4DC8-A0DC-F071...	Completed	Mon Apr 30 12:21:43 CEST 2012	Busy	Mon Apr 30 12:21:43 CEST 2012
1-322204195	6DB71AEC-FA56-4D15-AFFE-6986F...	Completed	Thu May 24 11:03:31 CEST 2012	Busy	Thu May 24 11:03:31 CEST 2012
1-322581861	75F1DB14-8BFE-4AF0-8C7A-81EE6...	Completed	Tue Mar 13 11:53:35 CET 2012	Busy	Wed May 23 04:53:35 CEST 2012
1-327329880	96902101-3989-4E8B-9D27-9CABF...	Completed	Wed Feb 01 09:46:57 CET 2012	Busy	Thu Feb 09 09:46:57 CET 2012
1-333164156	F75625CD-038F-4A4B-B087-38216...	Completed	Fri Mar 30 11:45:20 CEST 2012	Busy	Mon Apr 16 14:45:20 CEST 2012

Figure 5.24: Visualization of sojourn time predictor results for the BPI2013 log.

supplied: The current prefix of activities that occurred, the current context and the current part of the week. Then, the following steps are alternated, until the Temporal router predicts the end of the trace:
Using the Queue Predictor, predict the queue exit time of the current activity. Use the context object to determine the context at the exit time, derive the part of the week of the exit time. Use the temporal router to predict the most likely next activity, using the context, part of the week and prefix of previous activities.
After the end of the trace is predicted, the final end time minus the absolute starting time is predicted as the sojourn time.

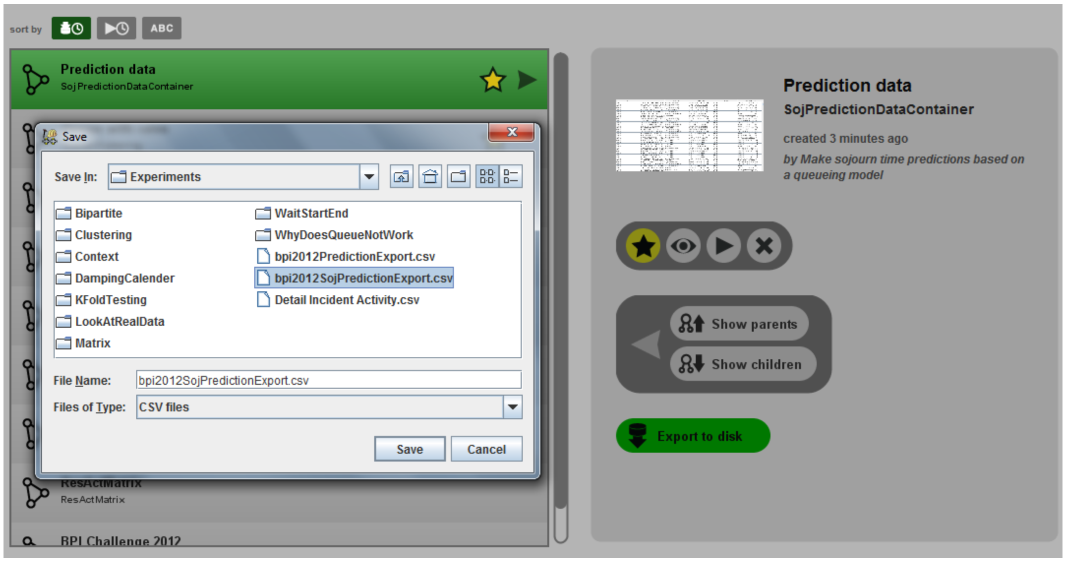
The practical plugin, named the "Make sojourn time predictions based on a queueing model", can be found in ProM as seen in Figure 5.23 and requires a testing set, training set and queue collection. Internally, the plugin uses a sojourn predictor and predicts the time for all traces in a test set. The result is then shown in a visualizer, as shown in Figure 5.24, as a list of predicted sojourn times. This result can also be exported to a CSV file using an export plugin, accessed by the standard ProM export button. Figure 5.25 shows an example of this.

In addition to a plugin based on a queue collection, testing set and training set, an all-in-one version was developed for end users. To start the all-in-one queue sojourn time predictor, all that is needed is a training and testing log. Figure 5.26 shows an example use of the plugin. Similarly to the sojourn time predictor, this plugin will also result in a set of prediction data which can be visualized and exported, as discussed previously.

5.4.3 How to find bottleneck queues

This subsection will discuss the implementation details of the bottleneck finder and the main functionality offered by plugins supporting these techniques. The bottleneck finder, as mentioned in Section 4.3, has been implemented as the Bottleneck Finder Plugin (See Section B.6) and has stayed close to its conceptual description. The algorithm works in two phases.

In the first phase, for each context and week part, the amount of arriving activities is obtained. This is done globally, so the count continues over all queues in the system. Then, the count of every context and week part is stored in a RankedPart object. The RankedPart object consists of the context, the week part and the amount of activities that arrived. The reason for putting them in this format, is that they can easily be sorted using standard Java Comparable operations. This results in a global ranking of how busy parts of the week in a certain context are.



Save In: Experiments

File Name: bpi2012SojPredictionExport.csv

Files of Type: CSV files

Figure 5.25: Exporting of sojourn time predictor results.

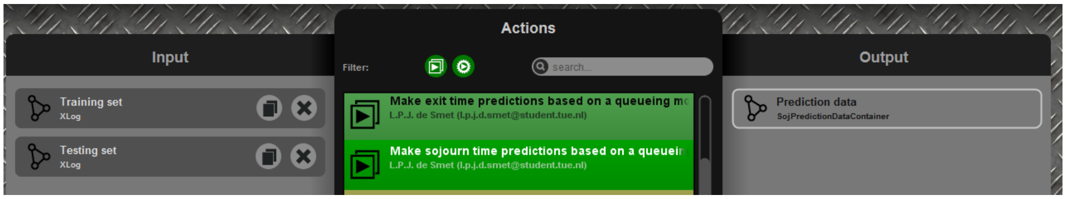

Input
Training set
XLog
Testing set
XLog
Actions
Filter:
search...
Make exit time predictions based on a queueing mo
L.P.J. de Smet (l.p.j.d.smet@student.tue.nl)
Make sojourn time predictions based on a queuein
L.P.J. de Smet (l.p.j.d.smet@student.tue.nl)
Output
Prediction data
SojPredictionDataContainer

Figure 5.26: Example use of all-in-one queue sojourn time predictor plugin.

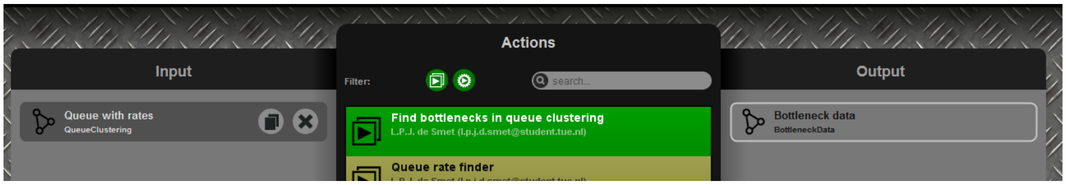
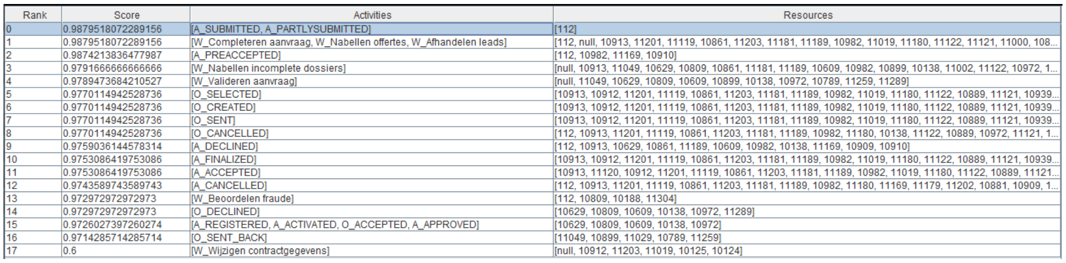


Figure 5.27: Example of Bottleneck finder use.



Rank	Score	Activities	Resources
0	0.9879518072289156	[A_SUBMITTED, A_PARTLYSUBMITTED]	[112]
1	0.9879518072289156	[W_Completeren aanvraag, W_Nabellen offertes, W_Afhandelen leads]	[112, null, 10913, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11019, 11180, 11122, 11121, 11000, 108...
2	0.9874213836477987	[A_PREACCEPTED]	[112, 10982, 11169, 10910]
3	0.9791666666666666	[W_Nabellen incomplete dossiers]	[null, 10913, 11049, 10629, 10809, 10861, 11181, 11189, 10609, 10982, 10899, 10138, 11002, 11122, 10972, 1...
4	0.9789473684210527	[W_Valideren aanvraag]	[null, 11049, 10629, 10809, 10609, 10899, 10138, 10972, 10789, 11259, 11289]
5	0.9770114942528736	[O_SELECTED]	[10913, 10912, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11019, 11180, 11122, 10889, 11121, 10939...
6	0.9770114942528736	[O_CREATED]	[10913, 10912, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11019, 11180, 11122, 10889, 11121, 10939...
7	0.9770114942528736	[O_SENT]	[10913, 10912, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11019, 11180, 11122, 10889, 11121, 10939...
8	0.9770114942528736	[O_CANCELLED]	[112, 10913, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11180, 10138, 11122, 10889, 10972, 11121, 1...
9	0.9759036144578314	[A_DECLINED]	[112, 10913, 10629, 10861, 11189, 10609, 10982, 10138, 11169, 10909, 10910]
10	0.9753086419753086	[A_FINALIZED]	[10913, 10912, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11019, 11180, 11122, 10889, 11121, 10939...
11	0.9753086419753086	[A_ACCEPTED]	[10913, 11120, 10912, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11019, 11180, 11122, 10889, 11121...
12	0.9743589743589743	[A_CANCELLED]	[112, 10913, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11180, 11169, 11179, 11202, 10881, 10909, 1...
13	0.972972972972973	[W_Beoordelen fraude]	[112, 10809, 10188, 11304]
14	0.972972972972973	[O_DECLINED]	[10629, 10809, 10609, 10138, 10972, 11289]
15	0.9726027397260274	[A_REGISTERED, A_ACTIVATED, O_ACCEPTED, A_APPROVED]	[10629, 10809, 10609, 10138, 10972]
16	0.9714285714285714	[O_SENT_BACK]	[11049, 10899, 11029, 10789, 11259]
17	0.6	[W_Wijzigen contractgegevens]	[null, 10912, 11203, 11019, 10125, 10124]

Figure 5.28: Example result of bottleneck finder, showing bottlenecks in process.

The second phase focuses on obtaining rankings for each individual queue and measuring the similarity between the ranking of the queue and the global ranking. The ranking within individual queues is the mean service time. For each context and week part the service time is saved in a RankedPart object, which is then sorted. To compare the rankings, the Kendall-τ metric was used. The resulting value from the metric is saved as the score for a given queue. A list of queues with scores is obtained, sorted and returned as the final ranking of how much a bottleneck every queue is.

Now the main functionality of the plugins that support this functionality is presented. The bottleneck finder consists of two plugins: The first is the actual bottleneck finding plugin called “Find bottlenecks in queue clustering”. Figure 5.27 shows an example use of the bottleneck finder. The second is the bottleneck data visualizer, which is automatically accessed after using the “Find bottlenecks in queue clustering” plugin or by using the standard ProM visualization functionality. Figure 5.28 shows an example result for the BPI 2012 challenge log. In this result one can see the ordering of the bottleneck factor of the queues, implying the top listed queue is the biggest bottleneck.

In addition to a plugin based on a queue collection, an all-in-one version was developed for end users. To start the all-in-one bottleneck finder, all that is needed is an input event log. Figure 5.29 shows an example use of the plugin. The bottleneck data resulting from this plugin will automatically be visualized after executions, as previously discussed.

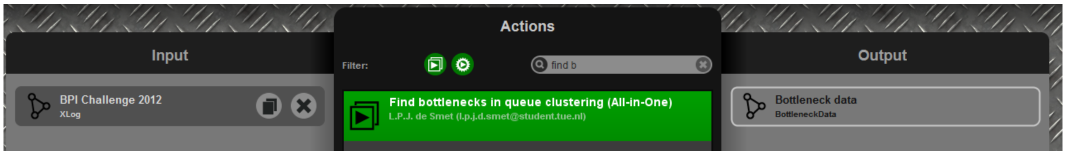

Input
Actions
Output
Filter:
find b
BPI Challenge 2012
XLog
Find bottlenecks in queue clustering (All-in-One)
L.P.J. de Smet (l.p.j.d.smet@student.tue.nl)
Bottleneck data
BottleneckData

Figure 5.29: Example use of all-in-one bottleneck finder plugin.

Chapter 6

Experimental evaluation

In this chapter the experimental evaluation of the techniques realized in the previous chapter is discussed. The experimentation will generally be done in two parts. The first is the validation of properties of the technique and trying to find the strengths and weaknesses of the approach. The second is to test the technique on real-life data sets and check how well they perform compared to other techniques. First, the experimental setup will be discussed in detail in Section 6.1. Then, there will follow three sections with experiments, one for each end-user plugin: Section 6.2 will discuss the experiments concerning the queue exit time predictor, Section 6.3 will cover the experiments on the sojourn time predictor and in Section 6.4 the experiments concerning the bottleneck finder will be discussed.

6.1 Experiment setup

This section will cover the experiment setup for all the experiments in the coming sections. In this section the machine used for experiments will be described, as well as the tools, their versions and if relevant their configuration. For most experiments two real data sets were used, namely the BPI 2012 challenge log and the BPI 2013 challenge log. Background information on these logs can be found in Appendix C.

The machine used for experiments contains an Intel©Core^{TM} i5 CPU and 4 gigabytes of RAM. For the experiments an eclipse environment was used in which a ProM environment version 6.3 (“Salt”) with UITopia was utilized with a maximum of 900 megabytes of memory for the java virtual machine. To generate synthetic logs CPN Tools version 4.0.0 and Prom Import Framework version 7.0 (Propeller) were used.

Throughout this chapter, experiments will be conducted using k-fold validation. To be able to perform k-fold validation, it is necessary to split up event logs into parts. A possibility would be to split up the event log on a trace or event level by randomly selecting traces or events. When splitting on event level randomly, however, it is probable that all information about traces is lost. When splitting on trace level, the problem of losing trace information is gone and would present a decent option. However, another valid option is introduced: splitting the event log based on time.

A k-fold validation method based on time is now discussed and is illustrated in Figure 6.1. Logs are split up on an event level. In Figure 6.1 the log is shown as a number of traces which are lines of blocks. Every individual block represents an event. The left side of each event block indicates the starting time and the right side indicates the ending time, which is defined as the start of the next event in the same trace. Each fold has a starting and ending time and all events for which the start falls between these times belong to that particular fold. This means that the

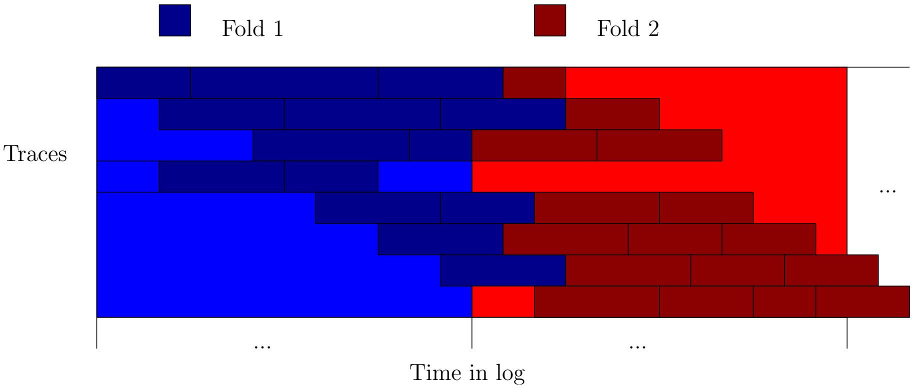


Figure 6.1: K-fold validation based on time splitting.

log is split at an event level and some traces will be split. For the experiments, a log is split up in n folds according to this strategy and $n-1$ folds are merged to form a training set. The other set will be used for testing. The assumption made for this validation method to work is that there is no concept drift, i.e.: the process considered remains largely stable over all folds. The amount of folds used in all experiments was chosen as a relatively small number: four, to prevent that a large amount of traces is split into many small parts.

For each experiment the error of the methods tested with in these experiments will be compared. An error metric will be used to compare predicted times resulting from some predictor and the actual times as they are in the event log. The metric used is the "normalized root mean square error", since it can be used independently of time sizes. This is necessary, since in real life logs there is a large difference between event durations, ranging from seconds to weeks. Performing better in these experiments thus means having a lower relative "normalized root mean square error".

6.2 Queue exit time prediction experiments

This section will cover all experiments concerning the queue exit time predictor plugin. First, a number of synthetic logs will be considered to show the properties of the queue exit time predictor in. Then, the performance of this technique on real-life data will be discussed.d.

For both experiments a number of techniques will be used for comparison. The first technique is the queue exit time prediction as described in Subsection 5.4.1. This technique will be tested for a different number of configurations of input variables k and with or without workload context information. Then, to provide a baseline comparison for this technique, a technique called "Average" is used. This technique takes the average over all the durations of activities in the training set and will predict this average in every case. In general the assumption is made that if a technique performs worse than the "Average" technique, it performs bad and if it performs better it is a good sign. Finally, the "Last event" technique is used, which is a snapshot predictor that predicts the duration of the last activity of some type as the duration for the incoming activity of the same type.

Idle period test: The purpose of the first experiment is to show the effect of the size of idle periods while maintaining a stable total idle time, where the size of the idle period indicates the amount of time in which no events are being executed. The expectations for this test are formalized in Hypothesis 6.2.1.

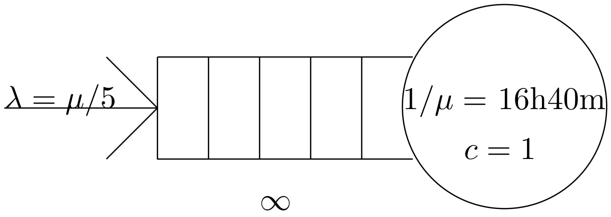


Figure 6.2: Idle period testing model.

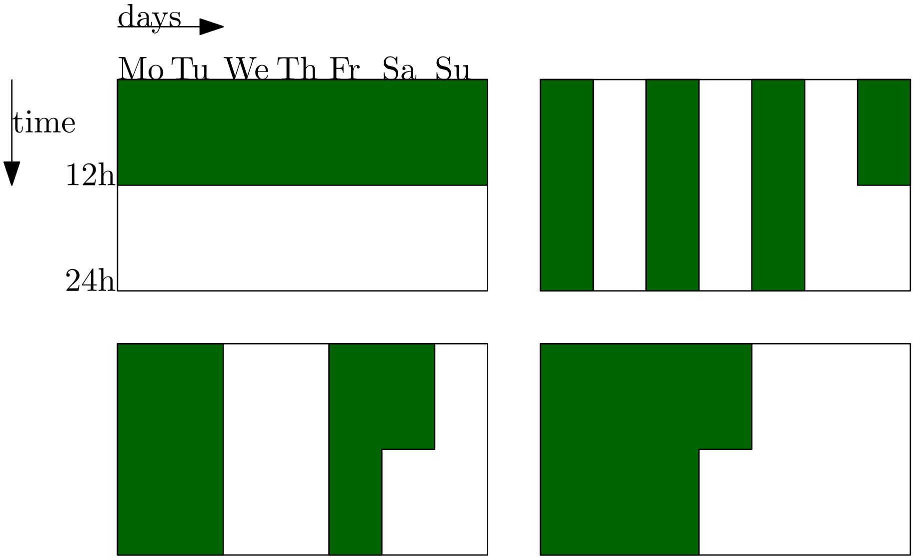


Figure 6.3: Idle period pattern configurations. For each graph the x-axis indicates the day and the y-axis indicates with green on which part of the day work is done, if at all.

Hypothesis 6.2.1 *While keeping the average service time and total idle time stable, the longer the amount of consecutive idle hours and hence smaller the amount of separate idle periods, the better the "Average all" predictor will work compared to the queue exit time predictor, since the completion times of only a few events will be influenced by the idle time. The "Last event" predictor will perform better once there are less and longer idle periods, for the same reason. The queue exit time predictor will perform relatively well in the scenario with a higher number of idle periods. The difference between using or not using workload context information should be negligible, since no change in workload on a large scale will happen.*

To test this approach a model designed in CPN Tools was used, of which the conceptual model can be seen in Figure 6.2. This model concerns a single queue, to provide a controlled testing environment. Jobs arrive with a stable rate λ following a Poisson process with mean arrival rate of five times the service rate. The service rate μ is defined by a Poisson process where the interarrival time $1/\mu = 16$ hours and 40 minutes. This time excludes the waiting time in the idle periods. The queue buffer is of infinite size and the amount of servers is one ($c = 1$).

The variation in this test is the number and length of idle periods in a week pattern. Figure 6.3 shows a number of week patterns that were used for testing. The granularity of the idle periods goes from 7 half days of idle period until one idle period of half a week long.

The following techniques were used in order to test the hypothesis: "Average", "Last event" and the queue exit time predictor with k = 24, for both cases with and without workload information. The resulting error values for these tests can be seen in Figure 6.4. In this graph, "Average"

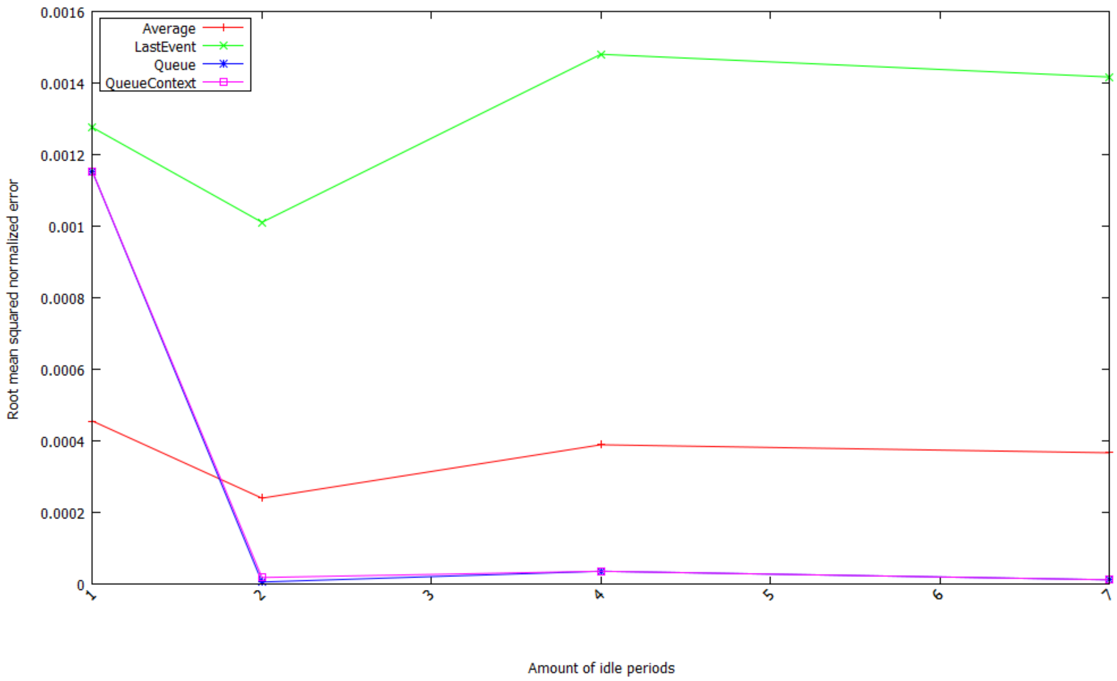


Figure 6.4: Idle period test results. The line for "Average" indicates the "Average all" method, the line "LastEvent" indicates the "Last event" predictor, "Queue" indicates the queue predictor with k =24 and no context information. "QueueContext" indicates the same method, but with context information. On the x-axis, seven idle periods corresponds to the top left time pattern in Figure 6.3. Four , two and one idle periods correspond to the top right, bottom left and bottom right pattern respectively.

indicates the "Average" technique, "LastEvent" denotes the "Last event" technique, "Queue" means the queue exit time predictor without workload information and "QueueContext" is the same, but with workload information. Workload information was extracted and added using the implementations described in Subsections 5.2.2 and 5.2.3 respectively.

The results in Figure 6.4 mostly conform to Hypothesis 6.2.1. As expected, the queue approach does well compared to the baseline "Average" and snapshot predictor "Last event" when the amount of idle parts increases. Additionally, both "Last event" and "Average" seem to perform somewhat better for the low amount of idle periods and it also seems to hold that there is hardly any difference between the queue exit time prediction techniques. The behaviour for both the "Average" and "Last event" techniques is expected for two idle periods, but the peak at one idle period is unexpected. The source of this effect is that within the large idle period a lot of activities pile up. Since these piled up activities are only started after the idle period, there exists heavy queueing. In the case of heavy queueing, activity durations become relatively long and create a sparse distribution. The latter makes it additionally difficult for the queue exit time predictor to find a good most likely value in distributions. The "Last event" method seems to perform bad overall, being outmatched by the bottom line of "Average". The reason for this is the fluctuation in event durations, i.e.: it often occurs that events with a relatively long duration and events with a relatively short duration alternate.

Idle period service time test: The second experiment is similar to the first in the sense that we will once again look at the size of idle periods. This time, however, the mean service time will increase as the largest idle period does. The expectations for this test are formalized in

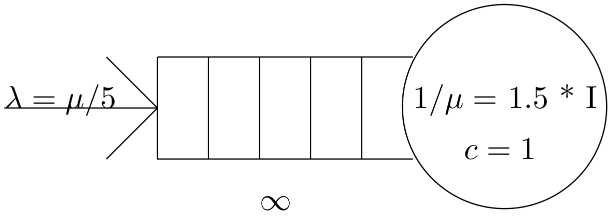


Figure 6.5: Idle ratio testing model.

Hypothesis 6.2.2.

Hypothesis 6.2.2 *While keeping the total idle time stable, when increasing the amount of consecutive idle hours and at the same time increasing the mean service time, the better the queue exit time predictor will score relatively to the "Average" and "Last event" techniques. It is expected that the queue exit time predictor will make equally good predictions with or without context information, since there is no big shift in workload.*

To test this approach a model designed in CPN Tools was used, of which the conceptual model can be seen in Figure 6.5, which is similar to that of the previous test. The only difference is that the service rate is now defined as $\mu = 1.5 * I$. The variable I here indicates the largest consecutive idle period present in the schedule. For variability, once again the week patterns of Figure 6.3 are used. The value I for the first pattern would be half a day, since every idle period has that size. The value for the second would be one day, since the largest idle period there is one day. This is done for each pattern and both mean arrival and mean service times are updated based on this value.

For this test the same techniques were used as for the previous test, i.e. the queue exit time predictor with $k = 24$, with and without context information, and both "Average" and "Last event" techniques. The error values results for these tests can be seen in Figure 6.6. Note that the number on the bottom indicates the amount of idle periods and not the length of the idle periods, i.e. the higher the amount of idle periods, the shorter their length.

The results in Figure 6.4 mostly conform to Hypothesis 6.2.2. As expected, the queue exit time predictor, independently of workload information, outperforms both other techniques for larger idle periods (and hence less idle periods). Unexpectedly, the baseline "Average" technique also performed relatively good for the larger idle periods. This is due to longer service times relative to idle periods giving a lower standard deviation between the predicted values. Another unexpected effect is that the snapshot predictor "Last event" performs worse the longer the idle periods become. This is due to an alternating effect that follows from larger idle periods combined with longer mean service times: longer durations are alternating with relatively shorter durations.

Arrival rate/service rate ratio test: The final synthetic experiment for the queue exit time predictor will be looking at the ratio between the mean service rate and mean arrival rate. One would expect that with no queueing, the "Last event" and "Average" predictor perform relatively well and that the queue exit time predictor shines when light queueing occurs. For heavy queueing it will be difficult for all techniques. These expectations are formalized in Hypothesis 6.2.3.

Hypothesis 6.2.3 *While keeping working and idle periods stable and increasing the ratio of arrival rate/service rate, the following will occur: For no queueing and heavy queueing, the queue exit time predictor will be inferior to the "Average" and "Last event" techniques. In between the queue exit time predictor will be better than the other techniques, independent of using workload information.*

To test this approach once more a single-queue model was used, with a single server. A week pattern as shown in Figure 6.7 will be used for defining the working and idle periods for this

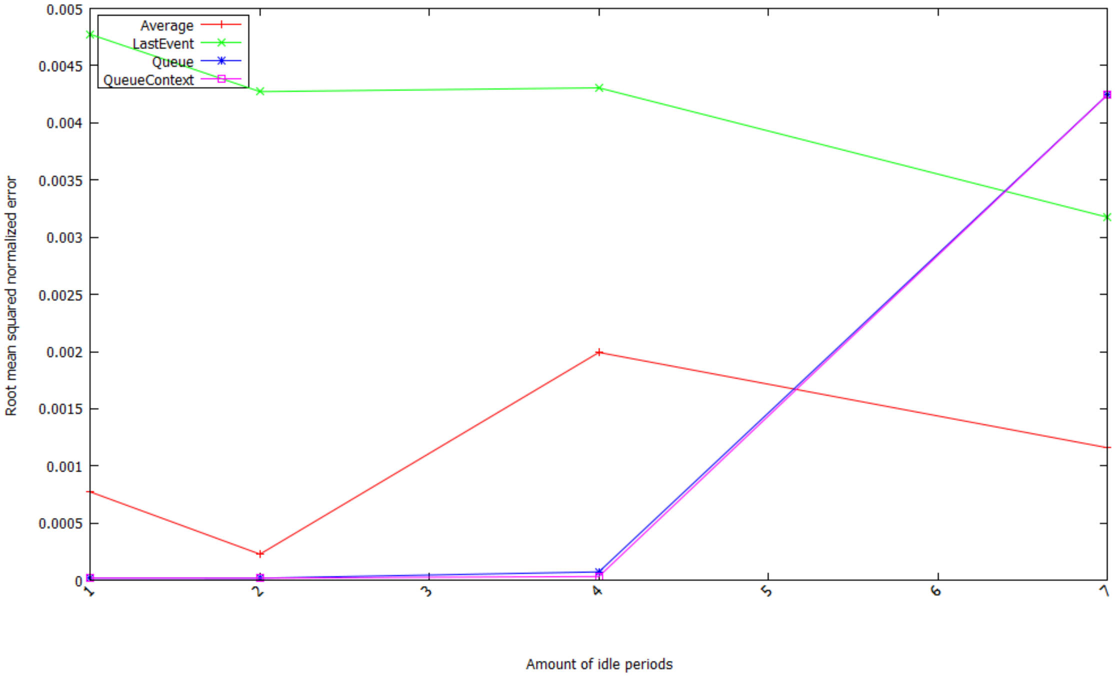


Figure 6.6: Idle ratio test results. The line for "Average" indicates the "Average all" method, the line "LastEvent" indicates the "Last event" predictor, "Queue" indicates the queue predictor with k =24 and no context information. "QueueContext" indicates the same method, but with context information. On the x-axis, seven idle periods corresponds to the top left time pattern in Figure 6.3. Four , two and one idle periods correspond to the top right, bottom left and bottom right pattern respectively.

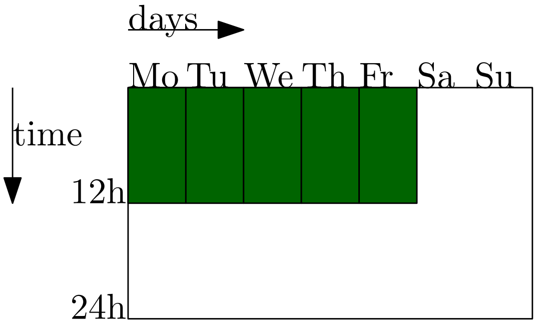


Figure 6.7: Standard week pattern. The green parts indicate which part of the day work is done.

experiment. One can observe that in the weekend days nothing is done and all the other days start with twelve hours of work, followed by twelve hours of idle time. The variation in this experiment will be the ratio between arrival and service rate, i.e. the factor r for which $\lambda = \mu/r$. The values used for r are in the set $\{3, 4, 6, 8, 10\}$.

The results of this experiment can be observed in Figure 6.8. As predicted in the hypothesis, the queue exit time predictor performs poorly in case of very heavy queueing (ratio = 3) and the other techniques suffer as well. Surprisingly, however, the queue exit time predictor with workload information does perform well for the case with heavy queueing. The reason for this is as follows: When heavy queueing occurs, the queue size keeps growing and growing and hence the amount of activities arriving and being executed grows compared to the starting state. For this reason, the queue exit time predictor with workload information can distinguish between a more quiet and a more busy period. Since the distributions for these periods are split up, a better prediction is obtained. In addition, the queue predictor performs well for very light queueing as well, which is due to the fact that the other techniques do not work well with the week pattern. It is noticeable that the snapshot predictor "Last event" is once more worse than the base line. This is due to at one side the week pattern and at the other the queueing behaviour, which lead to alternation between longer and shorter event durations.

Real-life data experiment: This experiment will focus on testing the queue exit time predictor on real-life data. A number of real-life data sets will be used for experimenting: The "BPI 2011" challenge log concerning a hospital process, the "BPI 2012" challenge log concerning the process of a financial institute, the "BPI 2013" challenge log on closed incidents of an incident management system, a log of a building permit application process for a Dutch municipality called "CoSeLoG", a log for the receipt handling phase of an environmental permit process which will be called "Receipt", and a real-life like synthetic log concerning a photo copier process called "synPhoto". All logs except the synthetic photo copier log consist of real data from companies and was used as a challenge input to test various process mining techniques. The synthetic photo copier log is included to see the difference between real life work patterns and artificially created working patterns. The source of all logs is the 3TU Datacenter[1] and all logs are described in Appendix C.

A number of techniques will be used for comparison in this experiment. The baseline for these experiments will once more be the "Average" predictor. The baseline will be used as an indication for whether the technique is worth using, i.e.: if it has a lower error than the baseline. The "Last event" snapshot predictor will be used for comparison as well. Finally, a number of configurations of the queue exit time predictor will be used. The different values for k will be one, four and 500. For each value of k, an experiment with and without workload information will be done.

Before a hypothesis will be made, the main characteristics of all the input logs will be summarized as a basis for this hypothesis:

[1] http://data.3tu.nl

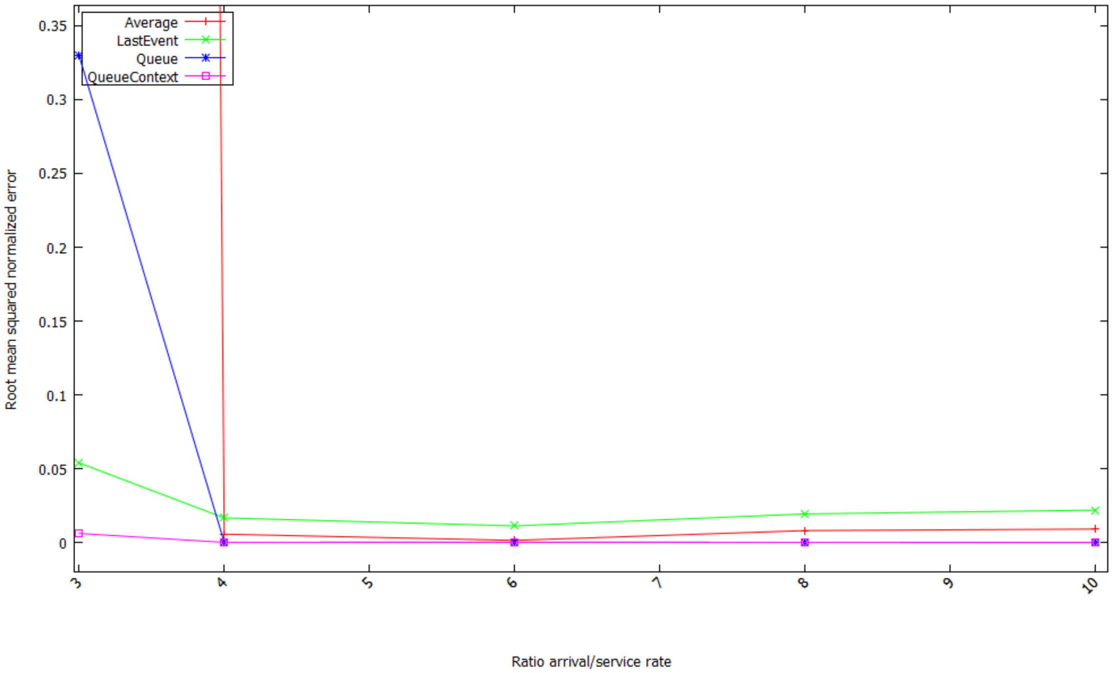


Figure 6.8: Arrival and service rate ratio test results. The line for "Average" indicates the "Average all" method, the line "LastEvent" indicates the "Last event" predictor, "Queue" indicates the queue predictor with k =24 and no context information. "QueueContext" indicates the same method, but with context information. The line for "Average" for a ratio of three performs relatively extremely poor and hence it was chosen to not show the complete line, as it would obfuscate the more interesting results.

- The BPI 2011 log was obtained from a Dutch academic hospital and consists of a very large amount of events (150291) and a relatively small amount of cases (1143), many of which take zero seconds of time and has in general long cases. There is a clear week pattern in this log, but there is not a clear workload difference pattern. There is a large number of different starting activities in this log.
- The BPI 2012 log was obtained from a Dutch financial institute and has a large number of cases (13087) and a large amount of events (262200). A large portion of the activities in this log take a very short time, e.g. a few seconds. There are two kinds of cases present in this log, one kind is almost immediately over after it opens and the other kind takes a longer time, e.g. a few weeks. This log has a clear week pattern, but no clear workload pattern.
- The BPI 2013 log ontains events from an incident and problem management system and has a relatively small number of cases (1437) and also a small number of activities (6660). Typical cases take a very long time, e.g. a year and have a relatively small number of activities. There is a clear week pattern and workload difference pattern in this log.
- The "Receipt" log considers the receipt phase of an environmental permit application and has in general very short cases, e.g. done within a day, with some exceptions. There is a relatively small number of cases (1434) and activities (8577). This log shows both week patterns and difference in workload patterns.
- The "CoSeLoG" log contains the records of the execution of a building permit application process and has a small amount of cases (937) which are in general very long, e.g. a year. Many cases have a large number of events (38944 events in total), of which many of which are very short and some are take a very long time. There are a lot of different events present in this log. This log shows a week, but no workload pattern.
- The synthetic photo copier log "synPhoto" is the only synthetic log modeling a photo copier and hence shows no week nor workload patterns. There is a small amount of cases (100), but a large number of events (40995). Most of these events take around ten minutes and most cases are around half a year long, but most activity in this case is present in only a few weeks of the total time.

The main intuition is that the queue exit time predictor performs well on logs with day patterns and logs that take a relatively long time. The expectation is that the snapshot predictor performs well on very short activities and when there is less week patterns. Hypothesis 6.2.4 formalized these expectations.

Hypothesis 6.2.4 *It is expected that for logs with a week and workload pattern, i.e. "BPI 2013" and "Receipt" , the queue predictor with $k = 24$ and workload information will perform the best. It is expected that for all cases with a week pattern, i.e. all log except "synPhoto", the queue predictor with $k = 24$ will outperform other methods and queue methods with other values for k will not be consistently better than the other techniques. It is expected that the "Average" predictor will be the worst for all techniques except "synPhoto" and that it will perform well on "synPhoto" since it is very regular. It is expected that the "Last event" predictor will perform well overall, except for the logs with a workload pattern, i.e.: "BPI 2013" and "Receipt".*

The results of the experiments can be observed in Table 6.1 and are mostly as expected. In general, note that errors of 0.0 indicate errors so small that they were indistinguishable within the evaluation environment. There are small differences between these errors, but they are negligible. Also, in general, scores between different variants of k and with or without workload for the queue exit time predictor are not exactly the same, but have a negligible difference that is removed when rounding. Another note is that the choice for the activity duration has a large impact on the set of events tested and their result. Different interpretations for duration might lead to different results. Lastly, the values for the version of the queue predictor with $k = 500$ and the "BPI2013" log could not be completed, since there was not enough memory.

Technique, Log	synPhoto	Receipt	BPI2011	CoSeLoG	BPI2012	BPI2013
Average	**2.39E-8**	9.44E-5	2.81E-5	0.0086	2.85E-5	0.0010
LastEvent	1.45E-7	7.22E-8	**0.0**	4.87E-5	**0.0**	**3.42E-6**
Queue24	1.26E-4	1.26E-7	**0.0**	1.48E-4	1.16E-5	1.16E-5
Queue24Context	1.26E-7	**2.17E-8**	**0.0**	**3.10E-5**	**0.0**	**3.42E-6**
Queue1	8.92E-4	2.25E-6	3.72E-4	1.48E-4	**0.0**	7.18E-6
Queue1Context	0.0049	2.91E-7	3.52E-4	**3.10E-5**	**0.0**	7.18E-6
Queue500	1.26E-4	1.31E-7	**0.0**	1.48E-4	1.83E-7	-
Queue500Context	7.12E-4	1.64E-7	**0.0**	**3.10E-5**	3.78E-10	-

Table 6.1: Mean normalized root square error for a multitude of event logs on the x-axis and techniques used for prediction on the y-axis. Bold values are the lowest error values for some log.

For the "BPI 2011" log, it is noticeable that many techniques score a very low error. The reason for this is that a very large number of events in this log take zero time. The same holds for all the zero error values in the "BPI2012" log. For this reason, many of the models often predict exactly zero for a large number of events and get it right. The "Average" predictor performed, as expected, quite well when there was no week or context pattern. Especially for the "synPhoto" log, since this had high regularity. In general, the "Last event" predictor performs poorer than expected for some cases, but on par with the best queue technique in others. This is as expected, since "Last event" performs quite well on logs with short cases and activities and does not perform well on cases where workload has a big impact. The queue predictor with value $k = 24$ and workload context information consistently performs very good, as was expected. Other variants of the queue predictor also show good results, but not much consistency. In general the queue approach works well, defeating the base line "Average" and often also the "Last event" snapshot predictor. To further show the worth of this approach, experiments with other state of the art techniques should be executed, which is left as future work.

6.3 Sojourn time prediction experiments

This section will cover the experiments showing the capabilities of the queue sojourn predictor plugin introduced in Subsection 5.4.2. Given that part of the sojourn time predictor is the queue exit time predictor, a focus will lie on the next event prediction part of this approach. This is why there is only one experiment based on synthetic logs. In addition, an experiment on real-life data will be executed, once again using a large set of real-life logs. All tests will be executed by doing k-fold validation where logs are split on time. The effectiveness of techniques will be measured by taking all traces in a testing set and comparing the actual duration of those traces to the predicted duration. The prediction methods will get the first event of the trace as an input and need to deduce the sojourn time. The normalized root mean square error of the difference between the prediction sojourn time and the actual sojourn time is the error rate.

For these experiments a number of techniques will be used for comparison. The first technique is the queue sojourn time prediction as discussed in Subsection 5.4.2. This technique will be tested for possibly a different number of input variables k and possibly with and without workload context information. Then, to provide a bottom-line comparison for this technique, a technique called "Average trace" is used. This technique takes the average over all the durations of traces in the training set and will predict this average in every case. Finally, the "Last trace" technique is used, which is a snapshot predictor that predicts the duration of the last trace of the same type as the duration for all incoming trace durations to be predicted.

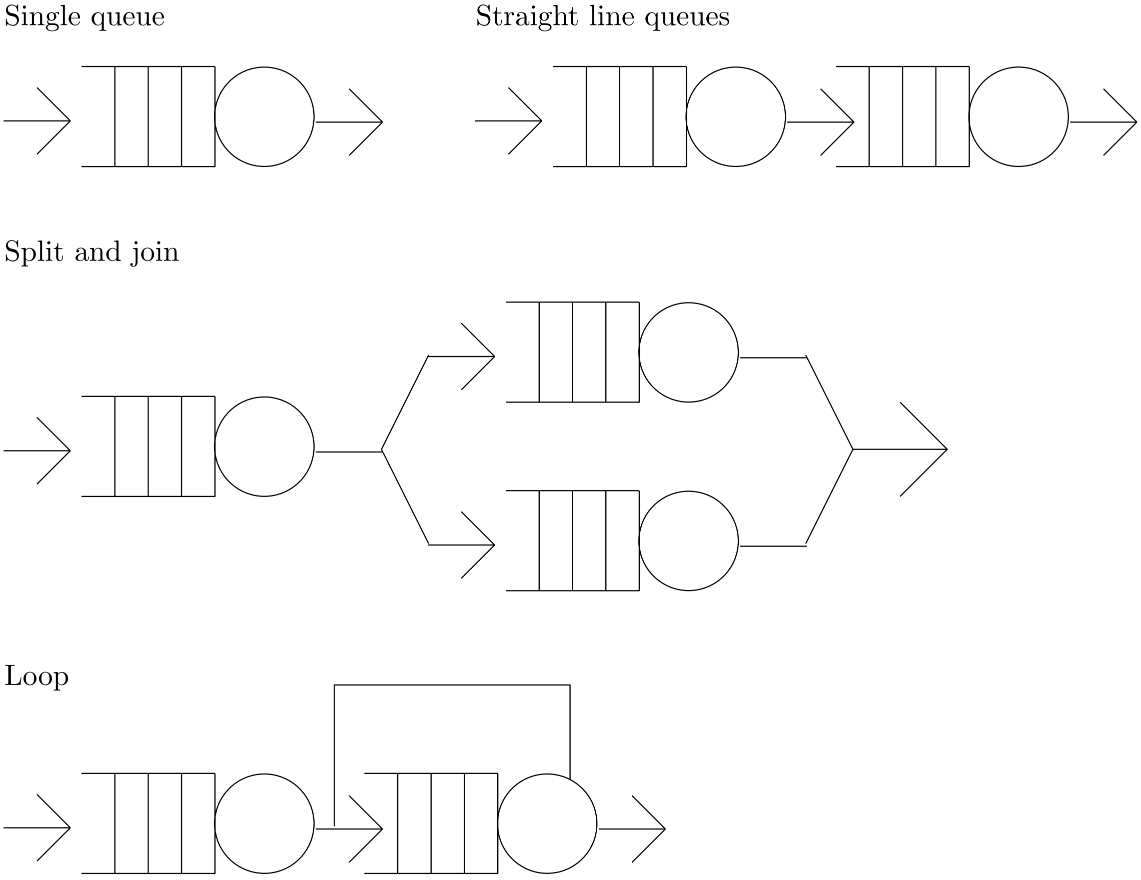


Figure 6.9: Basic queue collection patterns.

Basic process model pattern test: For this experiment four different basic patterns where implemented as CPN Tools models and used for log generation. A conceptual model of all four patterns can be found in Figure 6.9, which includes a single queue, a line of queues, a split and join and a loop. For this experiment the following techniques will be used: "Average trace", "Last trace" and the queue sojourn time predictor, with and without workload information for the following values of $k : \{5, 24, 500\}$. For each test the arrival rate was matched to the service rate such that mild queueing occurs, i.e. no queues that keep infinitely growing and no queues that are always empty. The amount of idle and work periods is described by a standard week pattern as in Figure 6.7. The expected results for all these techniques are formalized in Hypothesis 6.3.1.

Hypothesis 6.3.1 *The sojourn time predictor will perform on the following patterns in order of successful to less successful: Single queue, straight line, split and join, loop. The sojourn predictor will in general outperform the "Average trace" and "Last trace" techniques. It is expected that the sojourn time predictions with workload information will not perform better than those without and it is also expected that for $k = 24$ the results will be better than for $k = 5$ and $k = 500$.*

The results of this experiment can be seen in Figure 6.10 and do not completely conform to expectations. First, "Average trace" performs really well overall compared to the queue sojourn time predictions and "Last trace", which implies a low standard deviation on the activity durations. As expected, the predictor instances with workload information do not perform better than its counterparts without the workload information. This is due to the fact that there is no change in workload during these test setups which results in splitting up distributions on some arbitrary workload factor.

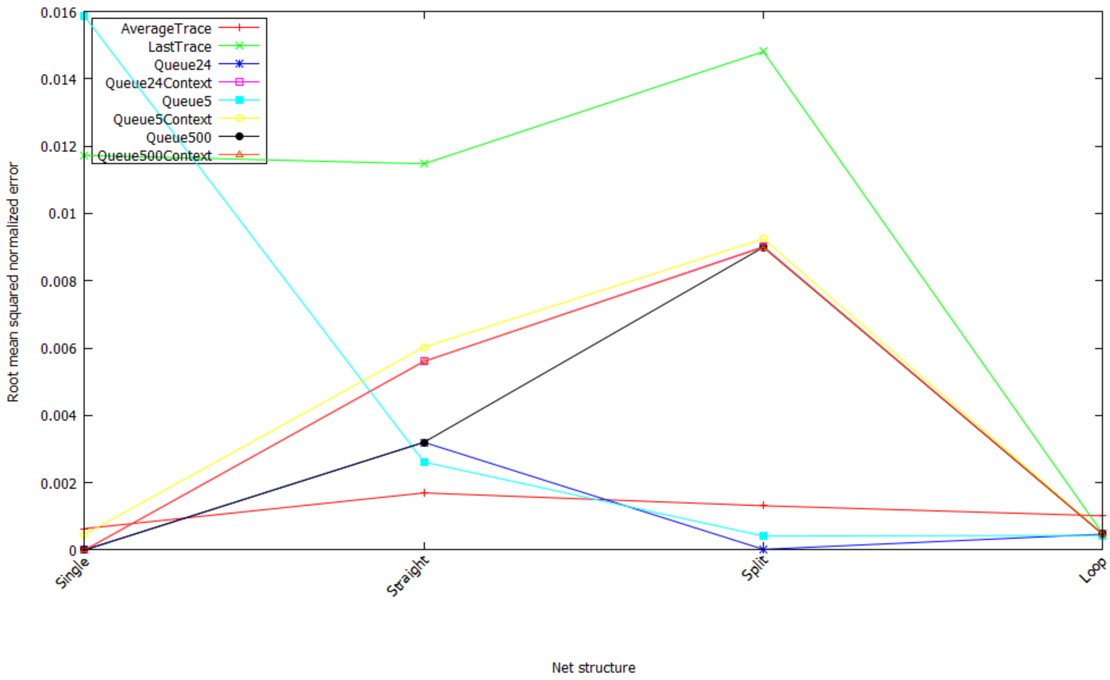


Figure 6.10: Sojourn prediction pattern test results.

It is apparent for this example that, as expected, having a good value for k makes a difference. All models except "Straight" show that $k = 24$ is the best choice. Prediction for the "Straight" model appears to be difficult for all techniques, given that "Average" outperforms the best. The reason for the queue sojourn time predictor is that there is a large amount of peaks of about the same height, but different duration, in the distributions for this technique, which make it very difficult to predict the exit time accurately. The other patterns do not show this effect as strongly.

Real-life data experiment: This experiment will focus on testing the queue sojourn time predictor on real-life data. A number of real-life data sets will be used for experimenting: The "BPI 2012" challenge log concerning the process of a financial institute, the "BPI 2013" challenge log on closed incidents of an incident management system, a log of a building permit application process for a Dutch municipality called "CoSeLoG", a log for the receipt handling phase of an environmental permit process which will be called "Receipt". All logs consist of real data from companies and was used as a challenge input to test various process mining techniques. The source of all logs is the 3TU Datacenter[2]. A number of logs was not included, because multiple test cases for these logs resulted in an out of memory error. The synthetic photo copier log was not included, since it did not contain enough traces to train a classifier with these traces as an input.

A number of techniques will be used for comparison in this experiment. The baseline for these experiments will once more be the "Average Trace" predictor. The baseline will be used as an indication for whether the technique is worth using, which is when it has a lower error than the baseline. The "Last trace" snapshot predictor will be used for comparison as well. Finally, a number of configurations of the queue exit time predictor will be used. The different values for k will be one, four and 500. For each value of k, an experiment with and without workload information will be done. The logs that are used in this experiment have been described in detail at the final experiment of Section 6.2. The expectations for these techniques are formalized in Hypothesis 6.3.2.

[2] http://data.3tu.nl

Technique, Log	Receipt	CoSeLoG	BPI2012	BPI2013
AverageTrace	0.0011	0.0086	0.0021	0.0080
LastTrace	2.79E-4	0.0028	0.0038	4.35E-8
Queue24	0.0012	9.90E-4	**1.06E-7**	**0.0**
Queue24Context	0.0109	0.0081	1.84E-6	5.90E-6
Queue1	5.17E-4	0.0020	1.09E-7	**0.0**
Queue1Context	**2.00E-4**	**0.0**	1.22E-4	1.25E-5
Queue500	0.0013	-	0.0032	-
Queue500Context	0.0015	-	1.27E-7	-

Table 6.2: Mean normalized root square error for a multitude of event logs and techniques tested with the sojourn time predictor. Bold values are the lowest error values.

Hypothesis 6.3.2 *In general, the "Average Trace" predictor is expected to perform poorly for cases where there is a large difference between case durations. This is in practice all input logs except the "Receipt" log. The snapshot predictor "Last Trace" is expected to perform well on short running cases, such as "BPI2012" and "Receipt". The queue predictor with* $k = 24$ *is expected to perform best for the logs without a workload pattern, e.g.: "BPI2012" and "CoSeLoG". The predictor with* $k = 24$ *and workload information is expected to be the best for the logs with workload information, e.g.: "BPI2013" and "Receipt". The queue predictors with* $k = 500$ *are expected to perform poorly, since data is split up in many parts for the sojourn predictor and it is expected this data will be too sparse to use for* $k = 500$*.*

The results of the experiments can be found in Table 6.2 and conform mostly to the expectations and also shows some surprising results. In general, note that errors of 0.0 indicate errors so small that they were indistinguishable within the evaluation environment. The "Average" technique performs badly overall but relatively good for the "Receipt" log, as expected. The "Last event" as expected, performs quite good for logs with short cases, especially "Receipt". Unexpectedly, the performance for "BPI2012" is quite bad and for "BPI2013" it is rather good. This is due to an alternating case length pattern in the "BPI2012" log and a rather stable change in case lengths for the "BPI2013" log. Finally, the queue predictor performs relatively well on not only the expected $k = 24$ ("BPI2012","BPI2013"), but also $k = 1$ often performs well ("Receipt","CoSeLoG"). The reason for $k = 1$ outperforming the others is that the sojourn predictor splits up data on a large number of aspects: unique prefix, relative time part, context and queue. This results in very sparse data for each split part and hence makes a poor predictor. The cases for $k = 1$ have an advantage due to less splitting compared to $k = 24$ and $k = 500$. This shows a weakness of the current sojourn predictor, as it needs a very large number of samples to work properly.

6.4 Bottleneck finding experiments

This section will cover the experiments done to show the capabilities of the bottleneck finder plugin as introduced in Subsection 5.4.3. For the bottleneck finder plugin there will be a single synthetic log experiment, showing that the bottleneck finder works as it should. Then, two experiments will be performed on real life data, namely the BPI 2012 challenge log and the BPI 2013 challenge log. All tests will be executed by doing k-fold validation where logs are split on time. The effectiveness of techniques will be measured by comparing the bottleneck finders results to known bottleneck information. For the real life data an argumentation based in log data was given to compare the bottleneck finder plugin's results.

Bottleneck finder basic test: This experiments is a sanity check to confirm the bottleneck finder method behaves as it should. For this experiment a model was constructed to generate log

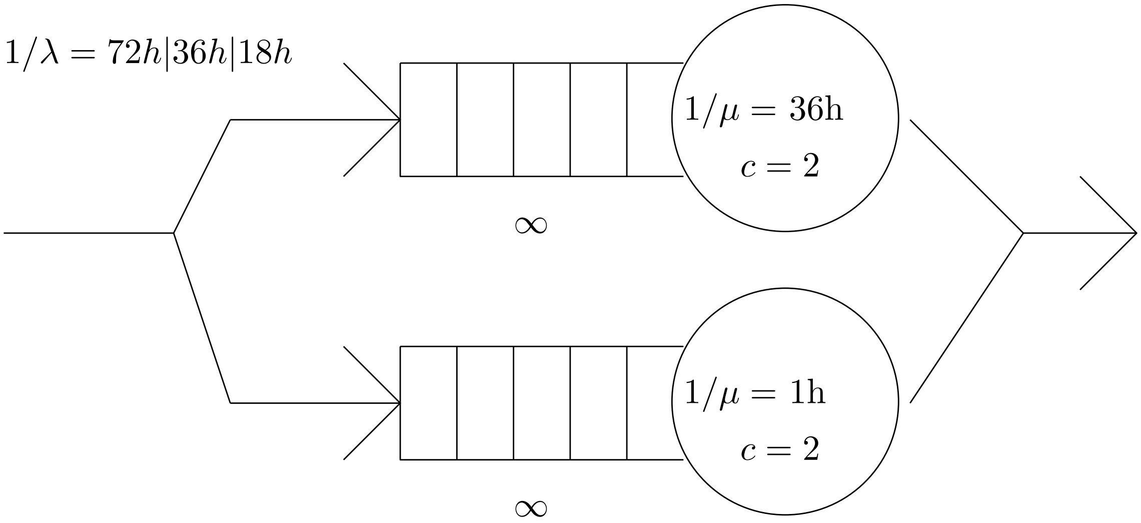
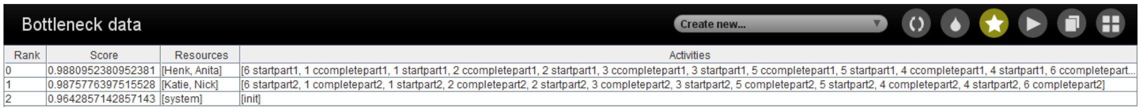


Figure 6.11: Queue collection for testing bottleneck finder. Work items arriving in the system split and arrive at both queues. Items arrive with an interarrival time of 72, 36 or 18 hours. The interarrival time changes every 504 hours. The top queue has a mean service time of 36 hours and the bottom queue has a mean service time of 1 hour.



Bottleneck data

Rank	Score	Resources	Activities
0	0.9880952380952381	[Henk, Anita]	[6 startpart1, 1 ccompletepart1, 1 startpart1, 2 ccompletepart1, 2 startpart1, 3 ccompletepart1, 3 startpart1, 5 ccompletepart1, 5 startpart1, 4 ccompletepart1, 4 startpart1, 6 ccompletepart...
1	0.9875776397515528	[Katie, Nick]	[6 startpart2, 1 completepart2, 1 startpart2, 2 completepart2, 2 startpart2, 3 completepart2, 3 startpart2, 5 completepart2, 5 startpart2, 4 completepart2, 4 startpart2, 6 completepart2]
2	0.9642857142857143	[system]	[init]

Figure 6.12: Bottleneck finder basic test results.

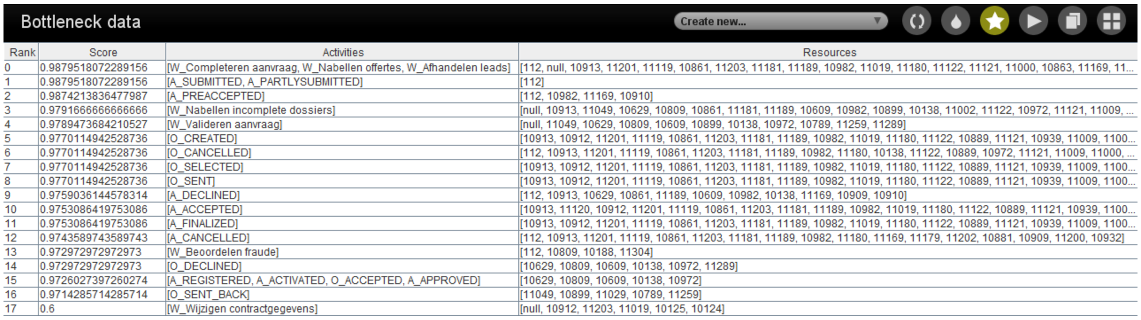
files in CPN Tools. The conceptual model for this implementation can be found in Figure 6.11. The interarrival times on the top left are 72, 36 and 18 hours. Every 504 hours the arrival rate switches to create artificial busy, regular and quiet periods. In the beginning of the model jobs are split to arrive at two separate queues. The service time of the first follows a Poisson process with a mean service time of 36 hours and the service time of the other queue follows a Poisson distribution with a mean service time of 1 hour. Both queues have two servers. The hypothesis for this model has been formalized in Hypothesis 6.4.1.

Hypothesis 6.4.1 *The queue with the higher mean service time of 36 hours will be predicted as a bigger bottleneck than the other queue with mean service time of one hour.*

The results of this experiment can be observed in Figure 6.12. All queues are ordered by their bottleneck ranking, i.e.: how much correlation there is between an increased arrival rate and an increased service time. The queue with a mean time of 36 hours was the queue with resources {“Henk”,“Anita”} and the other queue had resources {“Katie”,“Nick”}. Hence, it is expected that the former queue is a bigger bottleneck, which is confirmed by the results.

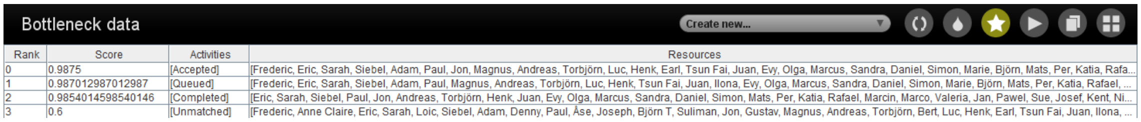
Bottleneck finder BPI 2012 challenge log test: To see the performance of the bottleneck finder on real data, the BPI 2012 challenge log was used as an input for this experiment. As input for the bottleneck finder a value of $k = 24$ was used. The expected results for all these techniques is formalized in Hypothesis 6.4.2.

Hypothesis 6.4.2 *By looking at the BPI 2012 challenge log it becomes apparent that many activities often have a very small duration, because they are executed by an automatic system. Any activities executed in this way that take only seconds are not expected to be part of a big bottleneck.*



Rank	Score	Activities	Resources
0	0.9879518072289156	[W_Completeren aanvraag, W_Nabellen offertes, W_Afhandelen leads]	[112, null, 10913, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11019, 11180, 11122, 11121, 11000, 10863, 11169, 11...
1	0.9879518072289156	[A_SUBMITTED, A_PARTLYSUBMITTED]	[112]
2	0.9874213836477987	[A_PREACCEPTED]	[112, 10982, 11169, 10910]
3	0.9791666666666666	[W_Nabellen incomplete dossiers]	[null, 10913, 11049, 10629, 10809, 10861, 11181, 11189, 10609, 10982, 10899, 10138, 11002, 11122, 10972, 11121, 11009, ...
4	0.9789473684210527	[W_Valideren aanvraag]	[null, 11049, 10629, 10809, 10609, 10899, 10138, 10972, 10789, 11259, 11289]
5	0.9770114942528736	[O_CREATED]	[10913, 10912, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11019, 11180, 11122, 10889, 11121, 10939, 11009, 1100...
6	0.9770114942528736	[O_CANCELLED]	[112, 10913, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11180, 10138, 11122, 10889, 10972, 11121, 11009, 11000, ...
7	0.9770114942528736	[O_SELECTED]	[10913, 10912, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11019, 11180, 11122, 10889, 11121, 10939, 11009, 1100...
8	0.9770114942528736	[O_SENT]	[10913, 10912, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11019, 11180, 11122, 10889, 11121, 10939, 11009, 1100...
9	0.9759036144578314	[A_DECLINED]	[112, 10913, 10629, 10861, 11189, 10609, 10982, 10138, 11169, 10909, 10910]
10	0.9753086419753086	[A_ACCEPTED]	[10913, 11120, 10912, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11019, 11180, 11122, 10889, 11121, 10939, 1100...
11	0.9753086419753086	[A_FINALIZED]	[10913, 10912, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11019, 11180, 11122, 10889, 11121, 10939, 11009, 1100...
12	0.9743589743589743	[A_CANCELLED]	[112, 10913, 11201, 11119, 10861, 11203, 11181, 11189, 10982, 11180, 11169, 11179, 11202, 10881, 10909, 11200, 10932]
13	0.972972972972973	[W_Beoordelen fraude]	[112, 10809, 10188, 11304]
14	0.972972972972973	[O_DECLINED]	[10629, 10809, 10609, 10138, 10972, 11289]
15	0.9726027397260274	[A_REGISTERED, A_ACTIVATED, O_ACCEPTED, A_APPROVED]	[10629, 10809, 10609, 10138, 10972]
16	0.9714285714285714	[O_SENT_BACK]	[11049, 10899, 11029, 10789, 11259]
17	0.6	[W_Wijzigen contractgegevens]	[null, 10912, 11203, 11019, 10125, 10124]

Figure 6.13: Bottleneck finder BPI 2012 challenge log results.



Rank	Score	Activities	Resources
0	0.9875	[Accepted]	[Frederic, Eric, Sarah, Siebel, Adam, Paul, Jon, Magnus, Andreas, Torbjörn, Luc, Henk, Earl, Tsun Fai, Juan, Evy, Olga, Marcus, Sandra, Daniel, Simon, Marie, Björn, Mats, Per, Katia, Rafa...
1	0.987012987012987	[Queued]	[Frederic, Eric, Sarah, Siebel, Adam, Paul, Magnus, Andreas, Torbjörn, Luc, Henk, Tsun Fai, Juan, Ilona, Evy, Olga, Marcus, Sandra, Daniel, Simon, Marie, Björn, Mats, Per, Katia, Rafael, ...
2	0.9854014598540146	[Completed]	[Eric, Sarah, Siebel, Paul, Jon, Andreas, Torbjörn, Henk, Juan, Evy, Olga, Marcus, Sandra, Daniel, Simon, Mats, Per, Katia, Rafael, Marcin, Marco, Valeria, Jan, Pawel, Sue, Josef, Kent, Ni...
3	0.6	[Unmatched]	[Frederic, Anne Claire, Eric, Sarah, Loic, Siebel, Adam, Denny, Paul, Åse, Joseph, Björn T, Suliman, Jon, Gustav, Magnus, Andreas, Torbjörn, Bert, Luc, Henk, Earl, Tsun Fai, Juan, Ilona, ...

Figure 6.14: Bottleneck finder BPI 2013 challenge log results.

This concerns most activities starting with A_ or O_. Any work which is done by people and often takes a longer time is expected to be a bottleneck. This concerns activities starting with W_.

The results of this experiment can be observed in Figure 6.13. The two biggest bottlenecks consist of activities {"W_Completeren aanvraag","W_Nabellen offertes", "W_Afhandelen leads"} and {"A_SUBMITTED","A_PARTLYSUBMITTED"}. The former was expected, since the activities starting with $W\_$ indicate people executing tasks. The latter is unexpected, but the reason quickly becomes apparent when looking at the log: These activities are the automatic system's way of starting the case, after which some human resource needs to pick up this task. Tasks are being picked up with a longer delays once things get busier, explaining the bottleneck behaviour. A similar argument can be made for the queue with "A_PREACCEPTED". Two exceptions exist in this list, which are the queues with activity "W_Beoordelen fraude" and "W_Wijzigen contractgegevens". The reason is on one side that these activities occur a very small amount of times in the log compared to the others and on the other side that all these activities on average take only a few minutes and are hence hardly influenced by a higher workload.

Bottleneck finder BPI 2013 challenge log test: The final experiment is another real data experiment which is based on the BPI 2013 challenge log. As input for the bottleneck finder a value of $k = 24$ was used. The expected results for all these techniques is formalized in Hypothesis 6.4.3.

Hypothesis 6.4.3 *It is expected that the starting activities, i.e. with activity "Accepted", have the biggest bottleneck, since cases are started more slowly when the workload increases. It is also expected that the ending activities, i.e. with activity "Completed" have a lower bottleneck rating, since they are often at the end of traces.*

The results of this experiment can be observed in Figure 6.14. These results conform to the expectations in the sense that activity "Accepted" has a high bottleneck factor and activity "Completed" has a lower factor. The "Unmatched" activity has a very low bottleneck factor, since it only occurs five times in the entire log and hence it can be disregarded.

Chapter 7

Conclusion

Within this master thesis a new area useful for providing operational support was proposed named *queue mining*. The strength of queue mining lies in combining the strengths of queueing theory and process mining to perform high quality predictions and analysis. A new technique for providing operational support and finding bottlenecks was proposed with a queue mining focus. The goal of this thesis was to answer the question: *Given an event log of a process, how can one combine queueing theory and process mining to gain insight in the bottlenecks of the process and provide operational support for the process?* The problems arising when answering this question have been discussed and a number of example techniques within the scope of queue mining have been proposed. Stating the main problem, in combination with the development, implementation and experimental evaluation of an example solution is the main contribution of this work. The contributions in this work have been summarized in Section 7.1. The area of queue mining is relatively new and has much potential for expanding. Possible future work in this area is covered in Section 7.2.

7.1 Summary of contributions

The core contribution of this work is a queue collection that accurately describes a business process and allows for predictions and analysis. The innovation of this model over previous techniques is that it uses both the strength of queueing theory by representing a process model as queues and the strength of process mining by using real life data patterns inside these queues. Instead of finding an average arrival rate and service rate, this technique proposes a model in which the arrival and service rate are extracted based on the parts of the log in some relative time and in some context. This queue collection is the core component that opens the way to a number of techniques.

Supporting contributions include the possibility for users to obtain, merge and use context information to enrich event logs with added information. Context information can be used generically in any technique that uses the XES format. In addition, a visualizer was created to allow a user to view the week and day patterns that are apparent within event logs. This allows the user to both gain insight in their business process and to provide the input for the techniques described below, i.e. the granularity of the week, or: the amount of parts a day should be split up in.

Using the queue collection result as an input, three example techniques were developed to provide operational support. The first contributed technique is that of exit time prediction. Queue exit time prediction has been implemented within the ProM framework and allows users to easily and quickly obtain predictions on the completion times of events within a business process. Everything is fully automatic and only requires a single input from the user: the granularity for the queue collection data.

The second contributed technique is that of sojourn time prediction. Sojourn time prediction was implemented in the ProM framework as well and allows users to predict the sojourn times for incomplete traces in a simple and fast way. The only input necessary is the preferred granularity and everything else is fully automatic.

The final contributed technique concerns finding the most important bottlenecks in a system. Bottleneck finding was also implemented within the ProM framework and allows users to, given an event log of some process, obtain a list of the most likely bottleneck queues in this process. Once more, this approach is fully automatic except for a granularity parameter.

To verify and validate the proposed techniques, an experimental evaluation was done comparing the proposed techniques with a baseline and a snapshot predictor. To validate and discover the properties of the proposed techniques, a number of synthetic logs were generated and their effect on the different techniques was analyzed. In addition, a number of experiments were run on real-life data to show the capability of handling real data. All experiments have shown that in general these techniques perform well compared to a snapshot predictor, as long as there are significant day and week patterns in the business process being analyzed.

This work has shown a novel set of techniques based on queue collections to provide operational support for business processes. This is achieved by finding bottlenecks in a business process and making predictions on the completion time of individual work items and complete cases. The techniques proposed have been implemented in the ProM framework and are publicly available and easy to use for analysts. All methods have been experimentally evaluated, yet a large number of challenges are still in the future. There is a large number of potential and possibilities within the scope of queue mining which is as of yet unexplored.

7.2 Future work

As stated before, there is a lot of queue mining left unexplored with this work and there are many possibilities for expanding this area. This chapter will discuss future work topics in the area of queue mining. In Subsection 7.2.1 the possibilities of incorporating queue mining techniques in other operational support and guidance frameworks will be discussed. Next, in Subsection 7.2.2 the possibilities for adapting other queueing models to fit this approach will be discussed. Then, Subsection 7.2.3 will discuss how simulation could help in providing an analysis tool. Next, Subsection 7.2.4 will discuss alternatives and possible improvements for the example techniques presented in this thesis. Finally, Subsection 7.2.5 touches upon how the experimental evaluation can be extended to make it stronger.

7.2.1 Direct operational support and guidance

This section will discuss the possibilities of incorporating queue mining techniques in other operational support and guidance systems. The plugins developed in ProM already provide a convenient method for doing predictions on completion times of events and traces and to find bottlenecks in a business process. An opportunity to improve the usability of predicting sojourn times and queue exit times in practice lies in a more real-time support and guidance based approach.

From the perspective of the current implementation, the most straightforward way to achieve this is to use the infrastructure in ProM introduced by Nakatumba et al. [31] that allows for testing of operational support algorithms. Embedding the current queue mining techniques in this infrastructure would give a valuable opportunity to test validate these methods.

Alternatively, such an approach could be realized in the following way. Given a business process management system with a certain system state, provide an interface adaptor to communicate with a set of queue mining implementations, possibly still within ProM. This interface adaptor then has to provide the following functionality:

- Given the historical data of a business process management system, an input XES Log training set should be generated.
- Given the system state of a business process management system, an input XES Log testing set should be generated.
- Given some training and testing set XES Log, the adaptor should be able to instantiate the queue miner and let it do sojourn and/or queue exit time predictions.
- Given one or more lists of prediction data, the adaptor should be able to either: convert the data to a format such that predicted data can be shown in the business process management system, or: visualize the prediction data in some other way externally.

Given an adaptor with such functionality it becomes possible to get real-time estimations on when jobs and processes will be completed. This can then be improved by adding a guidance system, for which an expansion of the sojourn time predictor would need to be developed. At this point the sojourn time predictor can predict accurately what the end time for the most likely path will be. If one would expand it such that it looks at the predicted end time of multiple or even all paths, it can give advice on which path would be the one that leads to the quickest completion time.

7.2.2 Adapting other queueing models to this framework

This section will discuss the possibility of using other queuing models for a queue mining approach. The current queueing model only consists of the most basic queue characteristics: The kind of events that arrive, the servers that operate the queue, the mean arrival rates and the mean service time distributions. In addition, the amount of servers is not used in the current approach, but could be in an alternative one. Developing and imbedding a more complex queue method could help improve the performance of predictions and might be achieved in the following ways:

- Extending the queueing model to use the amount of servers in it's prediction, by using e.g. Markov chains.
- Develop a technique which derives the queue capacity (if any) of queues within a process.
- Introduce a method for estimating how long it will take to process a certain number of items in a given queue.
- Develop a technique which derives the queueing discipline(s), such as FIFO, LIFO or priorities for certain traces.

Developing any of these possible extensions will result in a more detailed queueing model. The next challenge is developing a technique which uses some or all of the previously mentioned extra queue information to make better queue exit time predictions, sojourn time predictions and to enable a better way of finding which queues are bottlenecks.

7.2.3 Simulation as a means of analyzing a queue collection

The approach for predictions and analysis in this work focuses on considering historical data and a single system state for which information is to be extracted. It would be very informative as well, however, to look at a simulation model based on a queue collection. The work by Senderovich et al. [36] is shows promise for this direction. A list of possible valuable simulation setups will now be presented:

- Using the service rates of queue collection as a predictor, and some system state as input, one should be able to simulate the expected behaviour based on this current state to gain insight in the process and predict completion times for currently relevant cases.
- Using the service rates of the queue collection as a predictor and using the arrival rates of the queue collection in combination with some parameters to generate an input, one can see the effects of the current steady week state and try to identify bottlenecks.
- Using the same setup as the previous example, but allowing the user to tweak arrival and service rates, to allow for analysis of the system and gaining insight in the effects of for example increased workload or a shift of workload over different times or different queues.

7.2.4 Alternatives for the presented queue mining approach

This section will discuss possible improvements and alternatives for the techniques presented in this work.

First of all, the metric for workload in a system in Subsection 3.1.1 is not necessarily the best. A number of alternatives will now be presented. One could try to find the workload based on existing queueing theory techniques concerning workload, given some service rate and arrival rate. Additionally, there exist a number of techniques for measuring workload, some of which could possibly be adapted to fit the context of a queue collection. Having a better measure for workload also has a big impact on the bottleneck finder implementation, which builds upon this definition. Alternatives for implementing a bottleneck finder also include simulating historical data to find the bottlenecks. To validate the current technique and workload measure one would need to see if there is a strong correlation between workload and service time.

Next, the basic queue finder as introduced in Subsection 3.1.3 can be improved. The current metric for queue costs and fairness thresholds has a limitation. Considering the example in Table 3.4, the resource “Henk” is not considered a good match for “Sort package”, even though it he intuitively executes this activity a fair amount of times. A number of different metrics are possible for this fairness threshold, but none seem to work for all cases. Nevertheless, a list of alternatives will be presented here:

- Taking a fraction of the maximum value as a threshold.
- Taking the mean value over all resources, instead of only the resources participating in that activity and setting that value as the threshold.
- Alternatively for the previous idea, do not use the mean, but instead use the mean and some factor times the standard deviation to set the threshold.

In the end, however, it seems that an entirely different metric might provide better behavior for all cases.

Concerning the queue collection finder, there is a point that could be improved. The current implementation bins the service time distribution data using bins the size of the relative time parts in the week, e.g. one hour for $k = 24$. It might be better, however, to be able to separately config the relative time part size and the time distribution bin size, as activities might be completed in a matter of seconds, while the interesting patterns are still in the range of hours.

Then, there is a possibility to improve the algorithm that clusters activities and resources based on their cluster costs. The current algorithm is greedy and is rather naive. It would be interesting to search for an algorithm that is efficient, but returns a higher quality clustering. Of course, the definition of what is a good clustering is disputable as well, and finding which clusterings are good representations is another interesting topic to look into.

Additionally, the way the activity duration has been extracted is quite naive. One could research the possibilities of finding a better estimation of how long activities take, how long they are in the queue and how long they are in the work station. Optimally, one would like to obtain EPT's (as discussed in Subsection 3.2.1) automatically or semi-automatically to allow a better estimation of activity durations and possibly help with describing a better queueing model. A semi-automatical approach would involve users choosing which activities correspond to starting, completing and queueing actions. In the same spirit, letting a user have control over which parts of the XES XEvent attributes denote the activity might give better results for the predictions techniques that were previously discussed. For example the BPI 2013 log has a small number of activities when looking only at the "concept-name" attribute, but might have a finer grained and better queueing model if one would look at multiple attributes.

Next, there is a way to improve the queue exit time predictor. Considering the waiting time of some arriving event and some service time distribution, one can give a better prediction on when a job is done by taking the most likely value after the input waiting time. The current approach would predict the most likely overall time no matter how long the an event has already waited. This change also improves part of the sojourn time predictor, as it uses the exit time predictor as a module.

Finally, there exist a number of alternatives for implementing the temporal router presented as a part of the example sojourn time predictor. The weakness of the current router is that a large number of samples is necessary for the technique to work. One way to change this is to choose another way of presenting the prefixes in the current approach such that less unique prefixes exist. Alternatively, an entirely new sort of router based on temporal and contextual data could be devised or a routing technique could be used that does not build upon the queue collection, but rather tries to find workflow patterns in a more advanced way. In addition, the same could be said for the entire sojourn predictor approach. The current technique considers what the next queue is one step at a time. An alternative way would be to try and find the entire future path at once and use other forms of prefixes, transition systems, etc.

7.2.5 Extending the experimental evaluation

The experimental evaluation in this thesis has shown comparisons with a baseline predictor and a simple snapshot predictor. However, no comparison has been made to state of the art techniques for sojourn predictions and bottleneck finding. A valuable extension of this work would be to perform experiments and compare with some of the following techniques. This list is not exhaustive and other interesting prediction techniques will surely appear in the future.

- The cycle time prediction proposed by Crooy [9].
- The short-term simulation methods proposed by Rozinat et al. [34].
- The cycle time prediction techniques proposed by Schellekens [35].
- Deadline transgression prediction by Pika et al. [32].
- The Context-based prediction method by Eren [12].

In addition, it is possible to research the effect of using different kinds of context information and their effectiveness on helping predictions for a certain process.

Bibliography

[1] Rakesh Agrawal, Dimitrios Gunopulos, and Frank Leymann. *Mining process models from workflow logs.* Springer, 1998.

[2] Sren Asmussen and OnnoJ. Boxma. Editorial introduction. *Queueing Systems*, 63(1-4):1–2, 2009.

[3] Carlos A Bana e Costa, Leonardo Ensslin, Émerson C Cornêa, and Jean-Claude Vansnick. Decision support systems in action: integrated application in a multicriteria decision aid process. *European Journal of Operational Research*, 113(2):315–335, 1999.

[4] G. Bolch, S. Greiner, H. de Meer, and K.S. Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications.* Wiley, 2006.

[5] E Brockmeyer, HL Halstrm, Arne Jensen, and Agner Krarup Erlang. The life and works of ak erlang. 1948.

[6] Toly Chen. A fuzzy back propagation network for output time prediction in a wafer fab. *Applied Soft Computing*, 2(3):211–222, 2003.

[7] Steven I-Jy Chien and Chandra Mouly Kuchipudi. Dynamic travel time prediction with real-time and historic data. *Journal of transportation engineering*, 129(6):608–616, 2003.

[8] Jonathan E Cook and Alexander L Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(3):215–249, 1998.

[9] Ronald Crooy. Predictions in information systems. 2008.

[10] Anindya Datta. Automating the discovery of as-is business process models: Probabilistic and algorithmic approaches. *Information Systems Research*, 9(3):275–301, 1998.

[11] AK Alves De Medeiros and Christian W Günther. Process mining: Using cpn tools to create test logs for mining algorithms. In *Proceedings of the sixth workshop on the practical use of coloured Petri nets and CPN tools (CPN 2005)*, volume 576, 2005.

[12] Can Eren. *Providing Running Case Predictions based on Contextual Information.* PhD thesis, EINDHOVEN UNIVERSITY OF TECHNOLOGY, 2012.

[13] LFP Etman, CPL Veeger, Erjen Lefeber, Ivo JBF Adan, and Jacobus E Rooda. Aggregate modeling of semiconductor equipment using effective process times. In *Proceedings of the Winter Simulation Conference*, pages 1795–1807. Winter Simulation Conference, 2011.

[14] Erol Gelenbe. G-networks with triggered customer movement. *Journal of Applied Probability*, pages 742–748, 1993.

[15] Donald Gross, John F Shortle, James M Thompson, and Carl M Harris. *Fundamentals of queueing theory.* John Wiley & Sons, 2013.

[16] Christian W Günther and Wil MP van der Aalst. A generic import framework for process event logs. In *Business Process Management Workshops*, pages 81–92. Springer, 2006.

[17] Christian W Günther and Wil MP Van Der Aalst. Fuzzy mining–adaptive process simplification based on multi-perspective metrics. In *Business Process Management*, pages 328–343. Springer, 2007.

[18] Joachim Herbst. A machine learning approach to workflow management. In *Machine Learning: ECML 2000*, pages 183–194. Springer, 2000.

[19] Wallace J Hopp and Mark L Spearman. Factory physics: foundation of manufacturing management. *Chicago: Irwin*, 1996.

[20] James R Jackson. Networks of waiting lines. *Operations research*, 5(4):518–521, 1957.

[21] JH Jacobs, LFP Etman, EJJ Van Campen, and JE Rooda. Characterization of operational time variability using effective process times. *Semiconductor Manufacturing, IEEE Transactions on*, 16(3):511–520, 2003.

[22] JH Jacobs, PP Van Bakel, LFP Etman, and JE Rooda. Quantifying variability of batching equipment using effective process times. *Semiconductor Manufacturing, IEEE Transactions on*, 19(2):269–275, 2006.

[23] FJA Jansen, LFP Etman, JE Rooda, and Ivo JBF Adan. Aggregate simulation modeling of an mri department using effective process times. In *Proceedings of the Winter Simulation Conference*, page 82. Winter Simulation Conference, 2012.

[24] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007.

[25] David G. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *The Annals of Mathematical Statistics*, 24(3):338–354, 09 1953.

[26] Maurice G Kendall. A new measure of rank correlation. *Biometrika*, pages 81–93, 1938.

[27] Sander JJ Leemans, Dirk Fahland, and Wil MP van der Aalst. Discovering block-structured process models from event logs containing infrequent behaviour. In *Business Process Management Workshops*, pages 66–78. Springer, 2014.

[28] L Mayhew and D Smith. Using queuing theory to analyse the governments 4-h completion time target in accident and emergency departments. *Health care management science*, 11(1):11–21, 2008.

[29] AK Medeiros, AJ Weijters, and WM Aalst. Genetic process mining: an experimental evaluation. *Data Mining and Knowledge Discovery*, 14(2):245–304, 2007.

[30] Daniel A Menasce, Virgilio AF Almeida, Lawrence W Dowdy, and Larry Dowdy. *Performance by design: computer capacity planning by example.* Prentice Hall Professional, 2004.

[31] Joyce Nakatumba, Michael Westergaard, and Wil MP van der Aalst. An infrastructure for cost-effective testing of operational support algorithms based on colored petri nets. In *Application and Theory of Petri Nets*, pages 308–327. Springer, 2012.

[32] Anastasiia Pika, Wil MP van der Aalst, Colin J Fidge, Arthur HM ter Hofstede, and Moe T Wynn. Predicting deadline transgressions using event logs. In *Business Process Management Workshops*, pages 211–216. Springer, 2013.

[33] Anne Vinter Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Søren Christensen, and Kurt Jensen. Cpn tools

for editing, simulating, and analysing coloured petri nets. In *Applications and Theory of Petri Nets 2003*, pages 450–462. Springer, 2003.

[34] Anne Rozinat, Moe Thandar Wynn, Wil MP van der Aalst, Arthur HM ter Hofstede, and Colin J Fidge. Workflow simulation for operational decision support. *Data & Knowledge Engineering*, 68(9):834–850, 2009.

[35] B Schellekens. Cycle time prediction in staffware. *Master's Thesis, Eindhoven University of Technology, Eindhoven*, 2009.

[36] Arik Senderovich, Matthias Weidlich, Avigdor Gal, and Avishai Mandelbaum. Queue mining–predicting delays in service processes. In *Advanced Information Systems Engineering*, pages 42–57. Springer, 2014.

[37] Wil Van Der Aalst, Arya Adriansyah, Ana Karla Alves de Medeiros, Franco Arcieri, Thomas Baier, Tobias Blickle, Jagadeesh Chandra Bose, Peter van den Brand, Ronald Brandtjen, Joos Buijs, et al. Process mining manifesto. In *Business process management workshops*, pages 169–194. Springer, 2012.

[38] Wil Van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *Knowledge and Data Engineering, IEEE Transactions on*, 16(9):1128–1142, 2004.

[39] Wil MP Van der Aalst. *Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[40] Wil MP van der Aalst, Vladimir Rubin, Boudewijn F van Dongen, Ekkart Kindler, and Christian W Günther. Process mining: A two-step approach using transition systems and regions. *BPM Center Report BPM-06-30, BPMcenter. org*, 2006.

[41] Wil MP van der Aalst, M Helen Schonenberg, and Minseok Song. Time prediction based on process mining. *Information Systems*, 36(2):450–475, 2011.

[42] Wil MP van der Aalst, Boudewijn F van Dongen, Christian W Günther, Anne Rozinat, Eric Verbeek, and Ton Weijters. Prom: The process mining toolkit. *BPM (Demos)*, 489, 2009.

[43] Wil MP van der Aalst, Boudewijn F van Dongen, Joachim Herbst, Laura Maruster, Guido Schimm, and Anton JMM Weijters. Workflow mining: A survey of issues and approaches. *Data & knowledge engineering*, 47(2):237–267, 2003.

[44] Wil MP Van der Aalst, Mathias Weske, and Dolf Grünbauer. Case handling: a new paradigm for business process support. *Data & Knowledge Engineering*, 53(2):129–162, 2005.

[45] Jan Martijn EM van der Werf, Boudewijn F van Dongen, Cor AJ Hurkens, and Alexander Serebrenik. Process discovery using integer linear programming. In *Applications and Theory of Petri Nets*, pages 368–387. Springer, 2008.

[46] Boudewijn F van Dongen, Ana Karla A de Medeiros, HMW Verbeek, AJMM Weijters, and Wil MP Van Der Aalst. The prom framework: A new era in process mining tool support. In *Applications and Theory of Petri Nets 2005*, pages 444–454. Springer, 2005.

[47] CPL Veeger, LFP Etman, Joost Van Herk, and JE Rooda. Generating cycle time-throughput curves using effective process time based aggregate modeling. *Semiconductor Manufacturing, IEEE Transactions on*, 23(4):517–526, 2010.

[48] HMW Verbeek, Joos CAM Buijs, Boudewijn F Van Dongen, and Wil MP Van Der Aalst. Xes, xesame, and prom 6. In *Information Systems Evolution*, pages 60–75. Springer, 2011.

[49] AJMM Weijters and JTS Ribeiro. Flexible heuristics miner (fhm). In *Computational Intelligence and Data Mining (CIDM), 2011 IEEE Symposium on*, pages 310–317. IEEE, 2011.

[50] AJMM Weijters, Wil MP van Der Aalst, and AK Alves De Medeiros. Process mining with the heuristics miner-algorithm. *Technische Universiteit Eindhoven, Tech. Rep. WP*, 166:1–34, 2006.

[51] Lisa Wells. Performance analysis using cpn tools. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, page 59. ACM, 2006.

[52] Branimir Wetzstein, Philipp Leitner, Florian Rosenberg, Ivona Brandic, Schahram Dustdar, and Frank Leymann. Monitoring and analyzing influential factors of business process performance. In *Enterprise Distributed Object Computing Conference, 2009. EDOC'09. IEEE International*, pages 141–150. IEEE, 2009.

[53] Moe Thandar Wynn, Marlon Dumas, Colin J Fidge, Arthur HM Ter Hofstede, and Wil MP Van Der Aalst. Business process simulation for operational decision support. In *Business Process Management Workshops*, pages 66–77. Springer, 2008.

Appendix A

Glossary

(Event) log Log of events executed within some (business) process. A log consists of traces.

Trace One unique trace from an event log describing the path of a single execution of the process, discerned by some unique trace id. A trace consists of events.

Event One event within a trace, indicating some work done by some resource at some point in time.

Resource Any entity, human or machine, that performs a task in a process.

Basic Queue A queue for which only the servers and type of arriving activities are known.

Queue Object representing a queue in a business process. Contains a list of activities that arrive at the queue and a list of servers that can execute arriving activities. Queues have rates for the arrival of activities and a distribution of the service time of finishing activities.

Queue collection A collection of queues describing the process obtained by analyzing an event log.

Arrival rate Amount of tasks that arrive at a queue in some time interval.

Service rate Amount of tasks that are handled at a queue in some time interval.

Primetree Tree that consists of the multiples of a number of pre-defined primes. Every node has children with numbers that are a multiplication of the node number and a pre-defined prime. See subsection 3.2.2.

Prefix A representation of the activities that happened in the past of some trace.

Appendix B

Reference to plugins used

This appendix will provide a list of the ProM plugins that are available to an end user and give a short description of the purpose and the way to use this plugin. Any plugins specifically used for testing or intermediate results have not been described.

B.1 Start up period remover

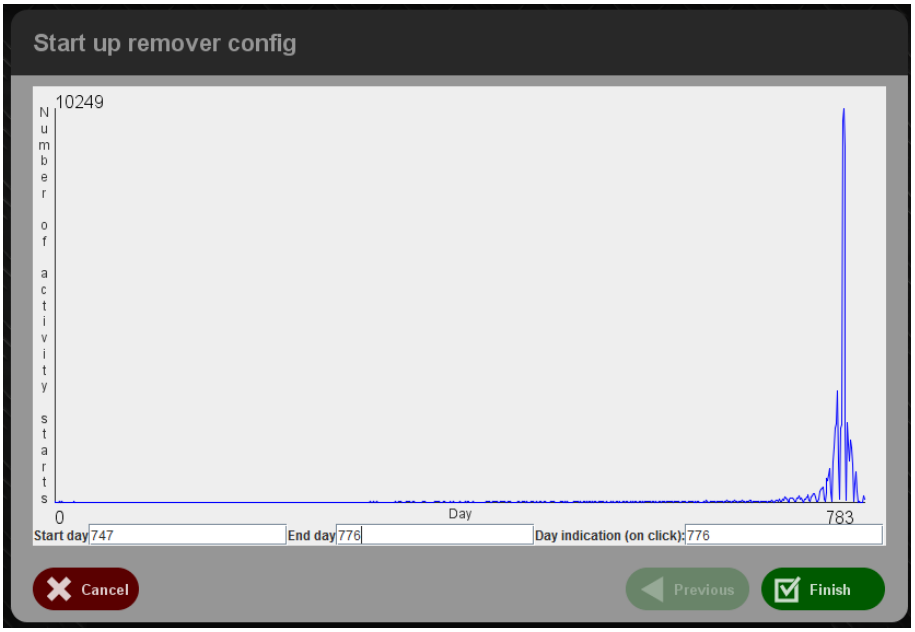
Plugin name: Start up period remover
Input: XES Log
Output: XES Log

The *Start up period remover plugin* was made to remove the start up and cool down periods from a log. The user gets a graph of activity as a number of events started per day and has to choose the starting and ending day of the relevant period (i.e.: the period excluding start up and cool and cool down). The log is split on event level, based on a given starting and ending time. Any event that does not fall within the time frame made by the starting and ending time specified is omitted. Any other events remain.

To use the Start up period remover, start the plugin with any XES Log. For this example, the 2013 BPI Log (as seen in Appendix C.3) will be used. The screenshot in Figure B.1 shows the interface in which the user can indicate two fields: The *Start day* and *End day* indicating the period in which events should be kept in the log. The graph assists the user in seeing where a lot of activity is happening and where none is happening. In the example of the 2013 BPI Log, before about day 700 there is a there is a relatively small percentage of events occurring. To get a more precise feel for which day should be chosen for start or end, the user can click inside the graph, which will put the number of selected day in the “Day indication” field. When the start and end day are chosen, the user can press Finish and the events outside the range will be removed, resulting in a new XES Log.

B.2 Week pattern visualizer

Plugin name: PrimeLogTree Viewer
Input: XES Log
Visualizer plugin The purpose of the PrimeLogTree Viewer is to find the variable k which indicates in how many pieces a week should be split up to best fit a business process. A user can access the PrimeLogTree Viewer through the visualizer menu with any XES Log and use the


Start up remover config

Start day 747

End day 776

Day indication (on click): 776

Figure B.1: Start up remover user interface.

visualizer to gain insight in how many parts is reasonable. More information on the implementation of the PrimeLogTree Viewer can be found at the end of Subsection 5.3.3.

The functionality of the PrimeLogTree Viewer will be explained. Figure B.2 shows the interface one gets when opening the visualizer on the BPI 2012 Log. There is a number of options in the interface, which will now be described from top to bottom. The “Begin new tree” button on top starts an entirely new tree from scratch. Depending on the field “Length of biggest time unit(days)” the amount of total days which are considered is changed. The standard value for this field is seven, a week. A new tree needs to be built to see the effect of changing this field. The second button states “Update tree” and it takes into account all the fields below it.

When pressed, the tree is updated with respect to those fields. The first field is the “Metric”, which indicates the metric used for visualizing the distributions. The metrics that can be chosen are the distribution size, distribution mean or distribution variance. In addition to changing the pattern metric, it is also possible to tweak the sizes of nodes and the space between nodes, to find a better picture of the tree, e.g.: fitting more nodes on the screen or making nodes larger to have a better understanding of their pattern. The fields that support this are “Node width” which lets the user set the width of all nodes, “Node height” which allows for setting the height of all nodes and “Layer spacing” with which one can customize the amount of pixels between layers of the tree. The final input fields are for “Zoom modus”. There are three different zoom modi: “None” which shows the standard tree from the root node, “Single node” in which the field below “Zoom modus” indicates the root of the subtree that will be shown and for the “Multiple nodes” mode a number of nodes indicated by a number of comma separated values in the field below will be shown aligned to make comparisons easier.

In addition, in the main screen of the visualizer on the left side, using the zoom modus of “None” or “Single node”, a user can click nodes to generate its children nodes. This process can be repeated to traverse the tree as the user would like. An example execution of this plugin can be found in Subsection 5.3.2.

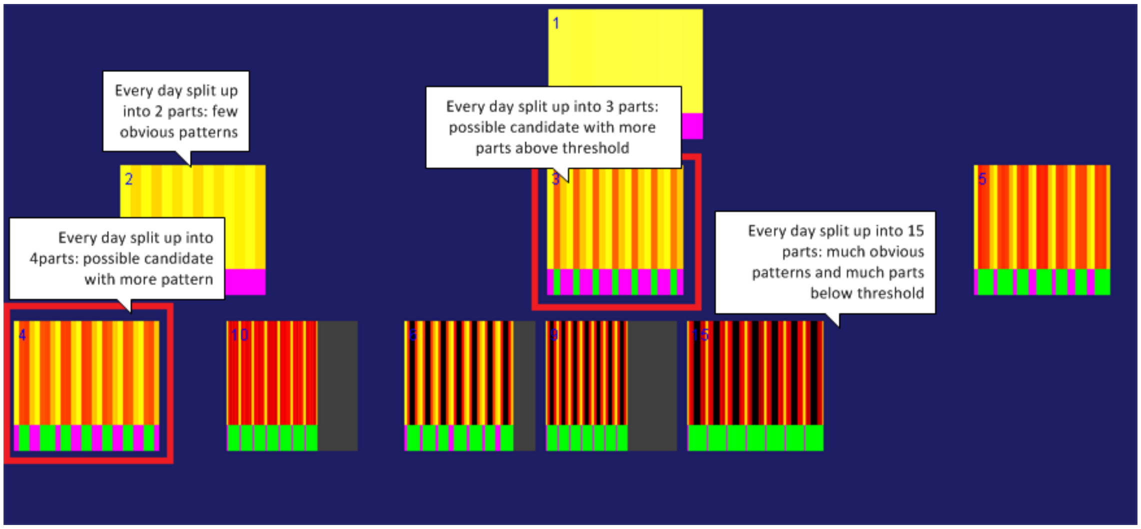

1
Every day split up into 2 parts: few obvious patterns
Every day split up into 3 parts: possible candidate with more parts above threshold
2
3
5
Every day split up into 4parts: possible candidate with more pattern
Every day split up into 15 parts: much obvious patterns and much parts below threshold
4
10
6
9
15

Figure B.2: PrimeLogTree Viewer basic user interface.

B.3 Context plugins

This section will discuss all context information related plugins.

Plugin name: Context model from XML file
Input: XML file
Output: TimeContext

The goal of this import plugin is to convert XML context information files into the TimeContext object format. The TimeContext object can be used for the other context plugins. To import, a user needs to select the "Context model from XML file" option when using the standard importing functionality of ProM on an XML file, as illustrated in Figure 5.3.

Plugin name: Export context model (XML)
Input: TimeContext
Output: XML file

The goal of this export plugin is to convert the internal TimeContext object into an XML file. To export, a user needs to use the standard ProM export functionality and export as an XML file, as shown in Figure 5.5.

Plugin name: Merge context info
Input: TimeContext, TimeContext
Output: TimeContext

The purpose of this plugin is merge context information from multiple sources. A user can specify two context information objects and merge them into a single TimeContext object. Figure 5.4 shows an example execution.

Plugin name: Enrich log with context info
Input: XLog, TimeContext
Output: Xlog

The "Enrich log with context info" plugin was created to allow adding context information to event logs. The TimeContext input will be used to set the context attributes for all events in the input XLog. Figure 5.7 shows an example execution of this plugin.

B.4 Queue exit time prediction

Plugin name: Make exit time predictions based on a queueing model (All-in-one)
Input: XLog, XLog
Output: PredictionDataContainer The goal of this plugin is to, given a training and testing set XLog, make predictions of the queue exit time for all events in the testing log. For each event the predicted end time will be saved in a PredictionDataContainer.

Figure 5.19 shows an example execution of the exit time predictor. The only input that has to be given is a parameter k, indicating the granularity of the queue collection distributions in the form of the amount of parts a day should be split up in. This is shown in Figure 5.17.

Like the context plugins, the resulting PredictionDataContainer is supported by a visualizer to directly inspect results as seen in Figure 5.20 and an export plugin to convert the resulting data to a CSV file as seen in Figure 5.21. These shall not be discussed in detail, but are nevertheless available for use.

B.5 Queue sojourn time prediction

Plugin name: Make sojourn time predictions based on a queueing model (All-in-one)
Input: XLog, XLog
Output: PredictionDataContainer The goal of this plugin is to, given a training and testing set XLog, make predictions of the sojourn time for all traces in the testing log. For each trace the predicted end time will be saved in a PredictionDataContainer.

Figure 5.23 shows an example execution of the sojourn time predictor. The only input that has to be given is a parameter k, indicating the granularity of the queue collection distributions in the form of the amount of parts a day should be split up in. A similar input field is shown in Figure 5.17.

The resulting PredictionDataContainer is supported by a visualizer to directly inspect results as seen in Figure 5.24 and an export plugin to convert the resulting data to a CSV file s seen in Figure5.25. These shall not be discussed in detail, but are nevertheless available for use.

B.6 Queue Bottleneck Finder

Plugin name: Find bottleneck in queue clustering (All-in-one)
Input: XLog
Output: BottleneckData The goal of this plugin is to, given an XLog describing some process, find the most important bottlenecks of this log by assigning a bottleneck factor. The bottleneck factor for each queue in the process will be saved within a BottleneckData object.

Figure 5.27 shows an example execution of the bottleneck finder plugin. The only input that has to be given is a parameter k, indicating the granularity of the queue collection distributions in the form of the amount of parts a day should be split up in. A similar input field is shown in Figure 5.17. The resulting PredictionDataContainer is supported by a visualizer to directly inspect results as seen in Figure 5.28.

Appendix C

Input and output log reference

This appendix serves as a reference for the real-life and synthetic data logs used. The source and characteristics of all logs will be discussed. All logs can be found in the 3TU Datacenter[1].

C.1 BPI 2011 challenge log

The BPI 2011 challenge log[2] is a real-life log, obtained from a Dutch academic hospital. This log contains 1.143 cases and 150.291 events. Apart from some anonymization, the log contains all data as it came from the hospital's systems. Each case is a patient of a gynaecology department.

C.2 BPI 2012 challenge log

The BPI 2012 challenge log[3] is a real-life log, obtained from a Dutch financial institute. This log contains some 262.200 events in 13.087 traces. Apart from some anonymization, the log contains all data as it came from the financial institute. The process represented in the event log is an application process for a personal loan or overdraft within a global financing organization. The log is split up into three distinct types: Activities starting with A_ imply states of the application, activities starting with O_ imply states of the offer belonging to the application and activities starting with W_ indicate a work item belonging to the application, i.e. a person doing some work. The tasks have information on whether they are starting, completing or queueing, but this is not used in this work to adhere to the assumption of handling a general log where no EPT's (see Subsection 3.2.1) are known.

C.3 BPI 2013 challenge log

The BPI 2013 challenge log[4] is a real-life event log from Volvo IT Belgium. The log contains events from an incident and problem management system called VINST and contains about 7.554 traces which contain 65.533 events. The log has a period where relatively few actions occur and a large peak of a few weeks in which most of the activity of the log occurs.

[1] http://data.3tu.nl

[2] http://www.win.tue.nl/bpi/2011/challenge

[3] http://www.win.tue.nl/bpi/2012/challenge

[4] http://www.win.tue.nl/bpi/2013/challenge

C.4 Receipt phase of an environmental permit application process (WABO), CoSeLoG project

This event log[5] contains the records of the execution of the receiving phase of the building permit application process in an anonymous municipality. This log contains 1.434 cases and 8.577 activities.

C.5 Environmental permit application process (WABO), CoSeLoG project

This event log[6] contains the records of the execution of a building permit application process. The log contains 937 cases and 38.944 events.

C.6 Artificial Digital Photo Copier Event Log

This event log[7] is an artificial event log for a digital photo copier. The log is used in a demo paper accepted at the CAiSE 2011 Forum. The log consists of 100 cases and 40.995 events.

[5] http://data.3tu.nl/repository/uuid:a07386a5-7be3-4367-9535-70bc9e77dbe6

[6] http://data.3tu.nl/repository/uuid:c45dcbe9-557b-43ca-b6d0-10561e13dcb5

[7] http://data.3tu.nl/repository/uuid:f5ea9bc6-536f-4744-9c6f-9eb45a907178