Process Mining for Systems with Shared Resources and Queues

Process Mining for Systems with Shared Resources and Queues

Process Modeling, Conformance Checking, and Performance Analysis

V.V. Denisov

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Denisov, V.V.

Process Mining for Systems with Shared Resources and Queues- Process Modeling, Conformance Checking, and Performance Analysis by V.V. Denisov. Eindhoven: Technische Universiteit Eindhoven, 2023. Proefschrift.

A catalogue record is available from the Eindhoven University of Technology Library

ISBN 978-90-386-5742-4

Keywords: Process mining, Performance analysis, Conformance checking, Log repair, Modeling, Process model, Material handling system, Event data, Partial order, Performance pattern, Performance spectrum, Predictive performance monitoring, Root cause analysis

ProefschriftMaken || www.proefschriftmaken.nl

The work in this thesis has been partially funded by Vanderlande Industries B.V., Vanderlandelaan 2, 5466 RB Veghel, the Netherlands, under the “Process Mining in Logistics” project.

Process Mining for Systems with Shared Resources and Queues

Process Modeling, Conformance Checking, and Performance Analysis

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr. S.K. Lenaerts, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op donderdag 1 juni 2023 om 13:30 uur

door

Vadim Vladimirovich Denisov

geboren te Leningrad, Sovjet-Unie

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

Voorzitter:	prof.dr. E.R. van den Heuvel
Promotoren:	dr. D. Fahland
	prof.dr.ir. W.M.P. van der Aalst (RWTH Aachen University)
Promotiecommissieleden:	prof.dr.ir. I.J.B.F. Adan
	prof.dr. M. Montali (Free University of Bozen-Bolzano)
	prof.dr. S. Rinderle-Ma (TU Munich)

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Dedicated to my family and parents.

Abstract

Organizations transform digitally to improve their processes, implement new business models, and develop new capabilities. Information systems execute their processes and store detailed data about the execution progress and outcome. Process mining is a field of data science that exploits such data for identifying process improvements and providing operational support. It is achieved through the *tasks* of *data-driven discovery* of *process models*, *conformance checking*, *performance analysis*, and so on. Historically, most process mining techniques address the analysis of *process instances*, or *cases*, in *isolation*, i.e., assuming that various cases do not affect each other. However, this assumption does not hold if cases *interact* on *limited shared resources*. In this case, applying many existing process mining techniques is infeasible as it would lead to poor or even inaccurate results.

In this dissertation, we study *material handling processes* of Material Handling Systems (MHSs) in logistics, such as Baggage Handling Systems (BHSs) of airports or warehouse solutions. In MHSs, cases are not isolated. For example, bags in BHSs interact on *conveyors* of *finite capacity* while competing for *shared machines*. The primary concern of MHS operators is to keep the MHS performance at the desired level. It makes improving material handling processes and providing operational support an actual problem. However, existing process mining techniques fail to capture interactions between cases. This dissertation aims to bridge this gap by adapting existing techniques and creating new ones, primarily targeting MHSs.

The proposed techniques aim at enabling the main process mining tasks for MHSs.

- We propose a generic technique for process performance description and analysis, called *performance spectrum*. It can reveal case interactions and various performance phenomena, which we describe in a taxonomy of *performance patterns*.
- We identify the key concepts for modeling MHSs: queues, resources, and routing functions by investigating existing queueing theory approaches and ma-

terialize them in *proclets* (i.e., sub-models) of the *Process-Queue-Resource system* (PQR-system), which is a dedicated *synchronous proclet system*, tailored for modeling MHSs.

- We adopt the concept of *generalized conformance checking* for the PQR-system by decomposing it into simpler tasks, for which existing approaches can be used. We propose a novel method for inferring missing events with timestamps for the log repair task of generalized conformance checking to address the problem of MHS data incompleteness.
- We propose a method for *root-cause performance analysis*, which aligns the performance spectrum with the PQR-system and uses the latter to discover the origins of undesirable performance scenario *propagation* observed in the performance spectrum. The performance spectra of the queue and resource dimensions are used to determine the root causes at the origins.
- Finally, we exploit the ability of the performance spectrum to capture the system dynamics to formulate a large class of *predictive performance monitoring* problems as a generic regression problem over the performance spectrum. We suggest a PQR-system-based method for selecting features relevant to learning the corresponding regression models.

The proposed techniques have been implemented as ProM plugins and stand-alone tools and evaluated on real data.

Contents

List of Figures

List of Tables

Acknowledgments

First of all, I would like to thank my first promoter Dirk Fahland for his endless patience, support, and guidance throughout the entangled unmarked paths of process mining research. Next, I wish to thank my second promoter Wil M.P. van der Aalst for directing my research toward tangible results and for his invaluable feedback. By discussing research with both, I could learn their ways of approaching scientific problems and reasoning about their solutions. Hope I was able to train myself a bit in their way of reasoning. Thank you both so much for helping me grow from an engineer to a researcher.

Next to my supervisors, I would like to thank the other committee members: Stefanie Rinderle-Ma, Marco Montali, and Ivo Adan, for making time to read the thesis, provide feedback, and attend the ceremony.

I would like to express my sincere gratitude to Vanderlande for making this project possible and for providing access to various material handling systems, invaluable datasets, and knowledgeable domain experts. Hilda Fabiola Bernard, Joost van Montfort, Erik Blokhuis, and Liam McMenamin – thank you for steering my research from the industry's side.

While at TU/e, I had the privilege to be part of the Process Analytics group. It gave me a unique opportunity to learn from the group's researchers. I'm grateful to Boudewijn van Dongen, Natalia Sidorova, Hajo Reijers, Marwan Hassani, Renata Medeiros de Carvalho. Undoubtedly, nothing could be accomplished without the help of Ine van de Moosdijk and Eric Verbeek.

I am grateful to my fellow Ph.D. students (many of whom are doctors already) Mitchel, Ayda, Rashid, Eva, Yorik, Azadeh, Gungming, Mohammadreza, Bart, Marcus, Xixi, Felix, Bas, Mike, Sander, Niek, Maikel, Eduardo, Cong for adding so much fun to the (sometimes unbearable) life of a Ph.D. student. Thanks should also go to the PADS, RWTH Aachen University, where I was welcome many times while meeting Wil.

I am incredibly grateful to Julia Kiseleva for inspiring me to become a researcher.

It took some time to become ready for the defense after my Ph.D. ended. All this time, I have been with Process Optimization at ServiceNow, working on cutting-edge

process mining software. I am grateful to Jochen Pohle, Andrei Vlad Oteanu, Jeroen van Gassel, Siddhant Sinha, Bojan Tomic, and the NowX team for their support and guidance.

Last but not least, no research could be possible without the unconditional support of my family, Lena, Anya, and Eva, and my parents, Liudmila Vasilevna and Vladimir Vasilevich. Lena looked after the kids, Anya entertained me in my free time, and Eva (impatiently) observed how the thesis was finalized. Meantime, my parents could support each of us when it was needed. I am enormously grateful to them.

Vadim Denisov
Amsterdam, April 2023

Chapter 1

Introduction

Performance analysis is an essential element in process management, relying on precise knowledge about the actual process behavior and performance. It can be done postmortem on historic data for enabling process improvements, and in real-time for detecting and mitigating emerging performance problems [1]. In logistics, Material Handling Systems (MHSs) provide short-distance movement (e.g., within a building) of discrete units according to *material handling processes*. For instance, an airport Baggage Handling System (BHS) accepts passengers *hold baggage* at check-in counters, move it throughout the terminal building(s), stores it temporarily in case of early check-in, and off-loads the baggage toward the aircraft. In this type of system, which this thesis is focused on, poor performance can lead to various *undesirable performance scenarios*, such as congestion, inefficient management of manual operations, materials mishandling (e.g., baggage being late for a flight), and results in lower customer satisfaction and higher operational costs. The following tasks, when executed systematically, can help to keep the performance at a desired level:

1. postmortem root cause analysis of undesirable performance scenarios for enabling improvements of material handling processes,
2. real-time detection and root cause analysis of undesirable performance scenarios for eliminating their root causes and preventing further performance deterioration,
3. predicting undesirable performance scenarios for preventing them from happening.

Event data emitted by machines and sensors in MHSs [2, 3], as well as in manufacturing [4] and the Internet of Things (IoT) [5, 6], offer opportunities for *process mining*, whose *event data-driven* techniques address problems of process analysis and monitoring. In turn, process mining techniques can help in approaching the creation of

AI-augmented business process management systems [7], and *digital twins* of business processes based on historical and current data [8, 9, 10].

However, MHSs is a challenging and perhaps one of the most interesting domains for process mining because it combines the physical world (e.g., systems with machines, conveyors, and so on) and human work [11]. Prior to this thesis, existing process mining techniques had limited applicability to those problems in the MHS domain. The understanding of this situation, when existing process mining techniques could not realize opportunities provided by the availability of event data, triggered the materialization of our project, called *"Process mining in logistics"*, and research presented this thesis.

In this chapter, we briefly introduce our project and MHSs in Section 1.1.1 and Section 1.1.2 respectively. We discuss how keeping the MHS performance on a desired level is one of the main concerns of the system operators and why it is a challenge for process analysis techniques in Section 1.1.3. We consider the event data MHSs generate in Section 1.1.4. We overview established and recent techniques for analysis of material handling processes and identify a gap in these techniques affecting MHS analysis in Section 1.2. We formulate our research questions in Section 1.3. Finally, we outline the approaches of this thesis in Section 1.4 and summarize our contributions in Section 1.5.

1.1 Challenges and Opportunities of Material Handling System Analysis

In this section, first, we briefly describe our project and Vanderlande[1] — the company that funded it together with the Dutch Research Council (NWO)[2] and participated in steering our research toward the goals. Then, we introduce MHSs by the example of an airport BHS, discuss the challenges that the MHS operators face, and consider why their analysis and improvement are difficult. We consider how MHSs usually record information about their operations in the form of event data, and how it becomes an opportunity for their performance analysis.

Most information in this section comes from our *practical experience* obtained at Vanderlande. During the project years, this information has been collected from interviews with domain experts of both the manufacturer and its customers' sides, project documentation, experiments with simulation models of MHSs, analysis of real event datasets, and evaluation of newly created techniques.

[1] `https://www.vanderlande.com/`
[2] `https://www.nwo.nl/`

1.1.1 Process Mining in Logistics

The project "Process mining in logistics" was organized for enabling research collaboration between the Eindhoven University of Technology and Vanderlande.

Vanderlande is a *material handling* and *logistics automation* company based in Veghel, Netherlands; it supplies MHSs for airports, warehousing, and parcel distribution industry [12]. For example, their BHSs and related passenger solutions can move over four billion pieces of baggage worldwide annually in more than 600 airports, including 12 of the world's top 20 [13]. As a BHS records dozens of events while handling each piece of baggage, i.e., a bag, all the Vanderlande-built BHSs record around a trillion events per year. Such an amount of data potentially provides a great variety of opportunities for applying process mining to analyze material handling processes. However, back to the project start time, process mining was not frequently used in the industry, and no studies were done regarding its applicability to material handling processes.

To shed light on it and enable process mining for the analysis of MHSs, "Process mining in logistics" included two Ph.D. projects, targeting research for materializing the following three "big rocks":

1. descriptive process mining,
2. predictive process mining, and
3. prescriptive process mining

of MHSs. This thesis is the result of one of these Ph.D. projects, which is completed first.

Next, we introduce MHSs by the example of a BHS.

1.1.2 Material Handling Systems in Logistics

We start by explaining how a BHS of a large airport handles passenger baggage and then discuss what other MHS types (besides BHSs) exist.

Material Handling Systems. An MHS handles Transport and Storage Units (TSUs) and delivers them to their destinations within a facility the system serves on time. For BHSs, security standards must be additionally observed. An MHS handles TSUs according to some *material handling process*, in order to provide the service reliably and fast. The process ensures that each TSU goes through a sequence of required process steps. Although these steps and their possible sequences vary from system to system, most MHSs have much in common. In the following, we consider an example of a BHS and then summarize the commonalities of MHSs and their processes.

Example. Now, let us consider a BHS at an airport as an MHS example. In a nutshell, a BHS is a network of connected conveyors that can transport bags. In the MHS

domain, a *Material Flow Diagram* (MFD) is typically used for documenting the system logical layout (i.e., its material flows), and key equipment in the system *areas*.

An MFD of a large BHS is shown in Figure 1.1. It has several areas:

- a *check-in* area with *check-in counters* $q_1^1,\ldots,q_{nz1}^1,\ldots,q_1^n,\ldots,q_{nzn}^n$,
- *transfer in/our* area for *transfer baggage*,
- connection to *another airport terminal* (for baggage),
- *preliminary sorting* area with *sorting loops* P^1 and P^2, X-ray hold baggage *scanners* $S_1 - S_k$, *cameras* a, x, *manual stations* m_1, m_2, and *dump area* u,
- *early bag storage*, and
- *final sorting* area with final sorters Y^1, Y^2 and *laterals* (exits) $y_1^1,\ldots,y_{nl}^1, y_1^2,\ldots,y_{nl}^2$.

The areas and their equipment are connected by *conveyors*. In Figure 1.1, each thin line represents a flow built from a single conveyor *link*, and each thick line represents multiple parallel links. A conveyor itself is either a series of *belts* or a series of *trays*. For the former, no permanent "lots" for placing TSUs are preserved on the belts, i.e., they can be placed anywhere on the belt. For the latter, each bag must be placed on a single free tray. Various equipment is installed *over* the conveyors for performing various process steps. For example, X-ray scanners $S_1 - S_k$ *screen* bags for detecting prohibited items.

Now, we consider an example of bag handling.

- A bag enters the system at check-in counter q_1^1.
- On check-in, the system immediately assigns it an *internal bag identifier*. This is the very first *process step* performed over the bag.
- Afterward, the bag travels to preliminary sorting loop P^1 via the link between the check-in and preliminary sorting areas. On the way, the system *tracks* its locations when it passes optical *sensors*, installed at various places over conveyors. As the system knows the exact speed of each conveyor, it can infer which bag passes a sensor. We consider passing a system location as a process step.
- Eventually, the bag reaches location b_1.
- At this location, the bag is *identified* by a camera installed at b_1. It reads the attached baggage tag with a bar code to recognize the *global bag identifier*. Using this identifier, the system associates the bag's internal identifier with the passenger information available for the global identifier (flight number, destination, class, etc.).
- Then, the bag is *merged* onto P^1 by *merge unit* b_1.
- After merging, the bag is *screened* by scanner S_1 and obtains *clearance* to continue to the aircraft. When prohibited or suspicious items are detected in a bag, it is offloaded out of the system to dump area u via exits D_1 and D_2 for further examination.

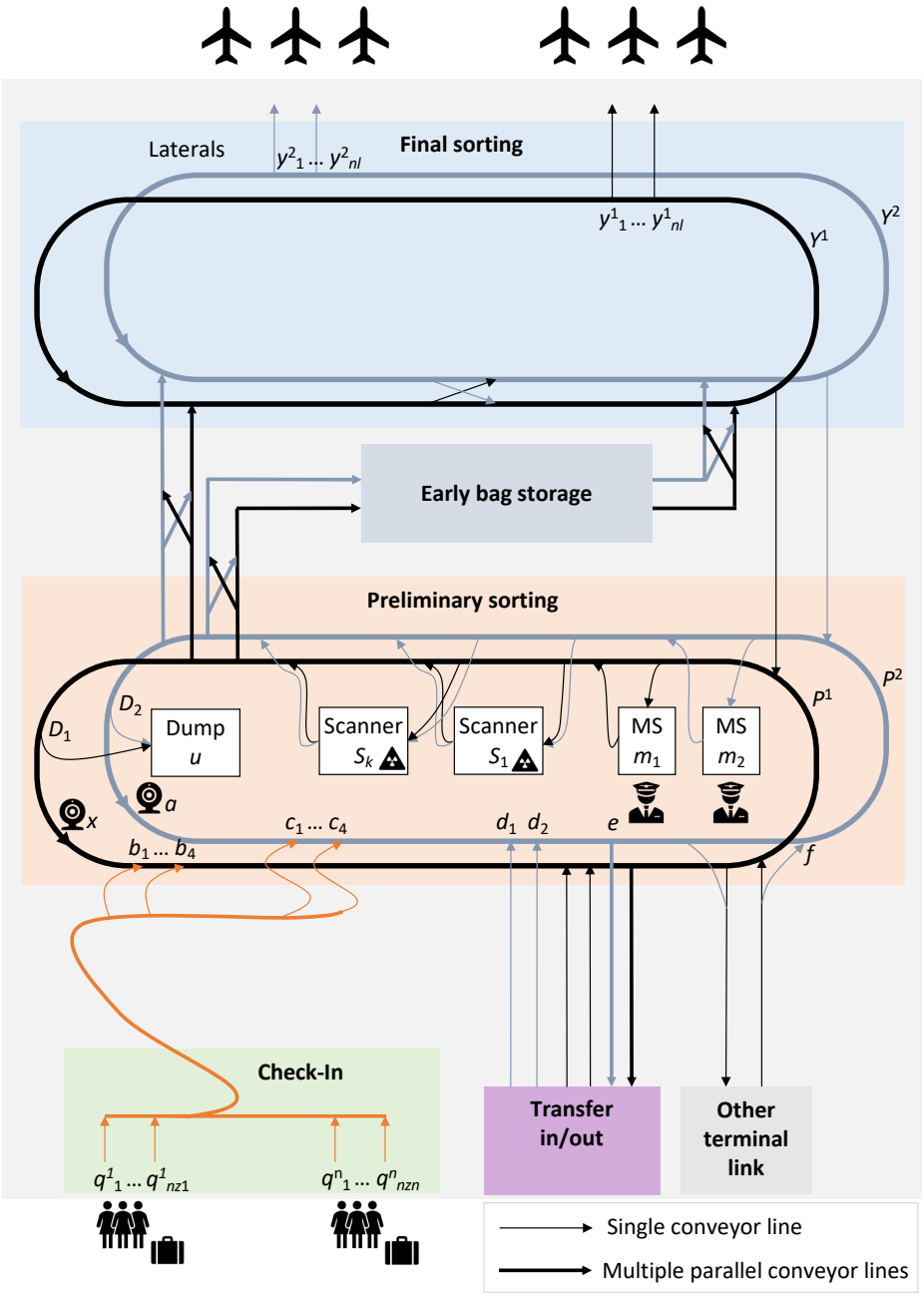


Figure 1.1: Baggage handling system at an airport.

- The bag is *diverted* via a *diverting unit* toward final sorter Y^1, where it leaves the system via lateral y_1^1, to be transported toward the aircraft. This is the last process step for this bag.

This is an example of a so-called *happy* (shortest) path throughout the system to the final destination. However, other paths are possible as well. For example,

- a bag is sent to a manual station for manual identification if it cannot be identified automatically,
- a bag can recirculate on a preliminary sorting loop if paths to its destination (e.g., scanners, manual stations, diverting units) are unavailable,
- a bag must be identified again if it is *lost in tracking*, i.e., if it was tracked at some location at an unexpected time,
- and so on.

Moreover, a bag can be sent to its destination via a longer path when the shortest path is unavailable (e.g., due to a mechanical failure or congestion).

To summarize, a BHS usually performs multiple process steps over a bag, such as registering, tracking at different locations, identification, screening, merging, etc. The next process step depends on both the bag status (e.g., whether it is identified) and the system state, i.e., equipment availability, load on system areas, etc..

To provide the required performance and the possibility to perform process steps in different orders, the system design and material handling process must

- be able to handle the maximum expected load,
- provide high availability, i.e., eliminate single points of failure,
- ensure alternative paths to destinations.

For example, the Heathrow Terminal 3 Integrated Baggage Facility is capable of handling 7.200 bags per hour. The BHS of Istanbul Airport (IST) handles up to 22.000 bags per hour and provides an availability of 99.99%. For details about BHSs of other large airports where Vanderlande-built systems are installed, e.g., Amsterdam Airport Schiphol, Las Vegas McCarran Airport, Hong Kong International Airport, Vancouver International Airport, and so on, we refer to Vanderlande's website.

In the BHS in the Figure 1.1, these requirements affect

- the number of check-in counters q and laterals y,
- the number and throughput of parallel conveyors in links between the system areas, e.g., the link between check-in and preliminary sorting areas,
- the number and throughput of sorting loops P^1, P^2, Y^1, Y^2, including their additional equipment (e.g., scanners $S_1 - S_k$, manual stations m_1, m_2),
- capacity of the other system areas, e.g., the early check-in storage,
- the number of alternative paths between areas,
- etc.

Redundant equipment (e.g., multiple scanners) helps both handle load peaks and provide high availability. As a result,

- a BHS usually has a complex network of conveyors with various equipment installed at different locations,
- the material handling process is large and complicated, and
- bags follow a great variety of paths throughout the system.

So far, we introduced BHSs. Next, we consider other types of MHSs.

Warehouse Systems, and Systems with Batching. General merchandise, fashion, and food retail companies use *warehouse* systems to handle TSUs (e.g., cartons or trays) carrying goods by means of conveyors.

For example, cartons come to a warehouse system from supplier trucks at the warehouse gates, and go to *picking stations*, either directly, or with an intermediate stop at short-term storage. At a picking station, *customer orders* are assembled from goods available in cartons and handed over for *packing* and *delivery*.

Often, such systems handle TSUs one by one, i.e., not consolidating them in *batches*, similar to the BHS we consider above. To refer to such systems in this thesis, we introduce a sub-class of MHSs that we call *MHSs without batching*.

However, there is another sub-class of MHSs, capable of handling TSUs grouped in *batches*. For example, a BHS can have an area where multiple bags are consolidated into large containers to be loaded onto the aircraft faster and in a more compact way. Some warehouse systems can distribute goods from pallets among cartoons, or consolidate multiple pieces of goods together on a *tray* to fulfill a customer order. We call this sub-class of systems *MHSs with batching*.

Thus, Vanderlande's warehousing solutions are the first choice for many of the largest global e-commerce players and retailers in food, fashion, and general merchandise across the globe. The company helps them to fulfill their promise of same-day delivery for billions of orders, with nine of the 15 largest global food retailers relying on its efficient and reliable solutions [14]. For example, Vanderlande has a fully automated distribution center in Zaandam for Albert Heijn, the largest supermarket chain in the Netherlands. It has an area of 48.000 m2, and there are 1.800 trucks that drive to and fro every day to supply more than 330 Albert Heijn stores [15]. Among Vanderlande's warehouse solutions customers, Lidl's distribution center in Køge, Denmark, Zalando in Sweden, Vegalsa-Eroski, Bosch in Karlsruhe, Asda in Warrington, and many others.

In the next section, we consider the challenges of material handling process analysis, actual for both sub-classes.

1.1.3 Challenges for Analysis of Material Handling Processes

In this section, we discuss problems that the MHS operators face daily, and why these problems are difficult to solve.

The MHS performance is crucial for customer satisfaction, so keeping it on a desired level is one of the main concerns of the system operators. Therefore, the performance is typically estimated through Key Performance Indicators (KPIs), such as:

1. the throughput of individual system areas, e.g., the throughput of hold baggage scanners $S_1 - S_k$ in Figure 1.1, in bags per hour,
2. the throughput of the whole system, e.g., the number of bags offloaded from laterals $y_1^1 - y_{\text{nl}}^1, y_1^2 - y_{\text{nl}}^2$,
3. the number of outliers, e.g., the number of TSUs that did not follow the optimal path, or could not reach their destination on time,
4. the equipment availability time, e.g., the availability time of manual stations m_1, m_2,
5. and so on.

Keeping these KPIs on the desired level requires avoiding undesirable performance scenarios that deteriorate the MHS performance.

Undesirable Performance Scenario Example. Let us consider an example of such a scenario for the BHS shown in Figure 1.1. For that, we consider its preliminary sorting loops P^2 in more detail. Its MFD is shown in Figure 1.2. Bags come onto P^2 from the check-in links via merge units $c_1 - c_4$, transfer links via d_1, d_2, and from the other terminals via link f. After merging, bags are screened at scanners $S_1 - S_k$, and diverted toward their destinations. If a bag cannot be diverted toward a scanner or its destination, it keeps circulating on P^2.

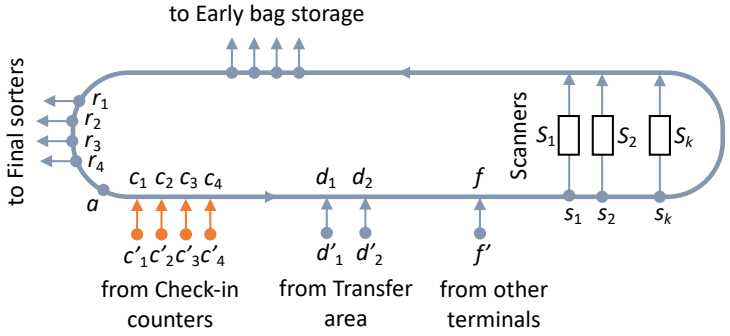


Figure 1.2: Sorting loop P^2 of the system in Figure 1.1.

Initially, the system operated normally. Suddenly, at time t_1, a bag got stuck at location s_1 during diverting toward scanner S_1, so the system stopped the entire belt

of P^2 to avoid piling bags before s_1. As a result, all bags on P^2 stopped moving, and the other bags, coming to P^2 through incoming conveyors $c'_1 - c_1$, $c'_2 - c_2$, $c'_3 - c_3$, $c'_4 - c_4$, $d'_1 - d_1$, $d'_2 - d_2$, and $f' - f$, started accumulating before merge units $c_1 - c_4, d_1, d_2$, and f, unable to merge onto P^2. Later, at time t_2, engineers arrived and fixed the problem, so P^2 resumed. Immediately, the bags, accumulated before the merge units, started merging onto P^2, causing load peaks for the scanners. As the scanner's throughput is limited and designed to handle some average load, many bags started recirculating on P^2 waiting for a chance to be scanned. Later, the peak was eventually handled, and the load on P^2 returned to normal. However, many bags were delayed due to:

1. the sorter halt during time interval $[t_1, t_2]$, and
2. recirculation on P^2 (due to waiting for an available scanner).

As a result, part of these bags arrived at their destination on final sorters Y^1, Y^2 too late and could not make it to the flight. So, this undesirable performance scenario worsened all of the aforementioned KPIs, caused additional costs on bag delivery, and lowered the satisfaction of the affected passengers.

Such scenarios can be potentially (1) prevented, or at least (2) mitigated quickly. For the former,

- the post-mortem scenario analysis can reveal flaws in the system and/or process design, the need for more frequent maintenance, inefficiencies in manual operation processes, etc., so the corresponding processes can be improved, or
- predictive performance monitoring can raise an alarm early enough to prevent an undesirable performance scenario from happening.

For the latter, determining the root causes of an undesirable performance scenario immediately after its detection (in real-time settings) can reduce the reaction time and minimize the impact. However, such prevention and mitigation are difficult to do because of the complexity of material handling processes. In the following, we consider the aspects of MHSs that cause this complexity.

Non-Isolated Cases. In processes, cases can be executed in isolation or interact with other cases on shared resources. As TSUs are not usually handled in isolation in MHSs, we consider the example of a Road Traffic Fine Management (RTFM) process instead to illustrate the former. Let us consider only a part of this process, responsible for registering fine tickets in the system. When a road camera detects a vehicle whose speed is above a certain limit, the RTFM process automatically

1. creates a ticket in a database, and
2. generates a notification for the driver.

The system behavior is never affected by the number of drivers violating a speed limit at the same moment in time. The system handles each case (ticket) as if it were the only case in the system, i.e., *in isolation*. Indeed, it is difficult to imagine that

a camera, installed, for example, at a ring road near the Eindhoven University of Technology, fails to create new tickets because too many drivers had been already fined there in the last 15 minutes. We call this type of process *processes with isolated cases*, i.e., process instances can be considered in isolation.

Conversely, in other processes, cases remind vehicles on roads rather than tickets. That is, the cases, like vehicles in heavy traffic, affect each other. For example, in the undesirable performance scenario example of this section, the bag stuck at s_1 effectively stopped handling the other bags on the same sorting loop and its incoming conveyors, which later caused also load peaks for the scanners. That is, this case caused multiple performance issues affecting many other cases through interactions on shared resources (conveyors, merge units, and so on). We call this type of process *processes with non-isolated cases*.

It comes as no surprise, the non-isolation of cases drastically impacts the way how the process behavior can be represented and analyzed. For example, ticket registration in the RTFM system can be analyzed one by one (i.e., case by case) without losing any information about the process behavior. However, the analysis of the undesirable performance scenario in a BHS is much more difficult because it requires considering all the cases (bags) *together* for revealing their interactions and inferring the scenario's root causes.

Resource availability, spatial limitations, and queues. MHS resources can be temporarily *unavailable* due to TSU handling problems (as we showed above), mechanical failure, or maintenance. They also have *finite capacity*. Some "simple" pieces of equipment, e.g., scanners $S_1 - S_k$, or merge units, have the capacity of just one TSU. In contrast, conveyors can usually move multiple TSUs at the same time. However, an MHS conveyor has a finite length and can accommodate only a finite number of TSUs. Moreover, its capacity depends on the length of TSUs (e.g., passenger bags usually have different dimensions), and the minimum allowed distance between the neighboring TSU. Thus, the actual conveyor capacity is a function of

- the length of each TSU,
- their *spatial configuration*, i.e., where they are located on the conveyor, and
- the minimum allowed distance between the neighboring TSUs.

So, the conveyor's actual capacity constantly varies while baggage is transported by the conveyor. When the maximum capacity is reached, no more TSUs can be added, so it becomes effectively (temporarily) unavailable for the other TSUs, and can trigger an undesirable performance scenario (e.g., massive recirculation).

As a result, the capacity and availability of each piece of equipment are not static but dynamically depend on TSUs handled by the system and constantly change over time. Knowing the exact equipment capacity and availability at each moment is valuable for understanding the system's behavior. However, it is difficult because many

factors, such as variable conveyor capacity, must be meticulously inferred from, for example, recorded event data.

Process complexity. MHSs are often large and complex. For example, a large BHS consists of thousands of interconnected conveyors and hundreds of other equipment pieces. This allows for many possible variants of TSU handling. Different variants are engaged under different load conditions, in case of equipment failure, during maintenance, and so on. As a result, the process that describes all these situations is large and difficult for understanding.

External factors. Some external factors can affect MHSs, for example, an unexpected flight schedule or supplier truck timetable change, traffic jams near the facility, weather conditions making bags wet and slippy, and so on. They may invalidate the analyst's (or model's) assumptions about probabilities of certain events, or stationarity of the process.

Batching. Last by not least, batching adds an additional level of complexity because (1) TSU consolidation behaviors and (2) additional types of TSUs (consolidated TSUs in a tray, empty trays, and so on) must be additionally analyzed.

The aforementioned factors make the MHS behavior overwhelmingly complicated, thereby making its analysis a challenging and painstaking problem. Next, we consider the event data that MHSs usually record to facilitate their analysis and monitoring.

1.1.4 Event Data Collected by Material Handling Processes

An MHS has numerous machines and workstations, which we consider as resources, performing process steps. An MHS can have sensors detecting TSU movement throughout the system, cameras identifying tags attached to the TSUs, devices that measure, weigh, and screen the TSUs, and workstations for manual operations. When a resource performs a process step, an *event* is generated and either recorded to the system database or dropped, depending on the logging architecture in place. In this section, we consider these events in more detail and show how they form traces, event logs, and event tables.

Event Data and Event Tables. Typically, an event generated by an MHS resource on a TSU handling has the following *attributes*:

1. a unique TSU identifier,
2. a timestamp of the process step start or completion,
3. the location of where the process step was performed,
4. the outcome of the *process step* execution.

Each machine and workstation is permanently installed for performing a particular process step. As a result, the location attribute value unambiguously identifies both the process step and the resource that performs it. That is, the information about the

resources performing process steps is typically presented in MHS events. Last but not least, different *process steps* may have a different outcome, including a *flag* whether the process step succeeded or failed due to an error. For example, the result of a bag identification in a BHS is

- the bag's unique identifier read from the attached sticker with a bar code,
- associated flight number,
- passenger class, and
- final destination.

Table 1.1 shows the example of events of bags with identifiers 1 and 2, which were handled by the sorting loop P^2 of the BHS in Figure 1.2 (the events generated at the locations before and after P^2 are not shown for clarity). In these events, each process step and resource is encoded through the process step execution location, e.g., merging is encoded by the location of the merge unit c_1, and so on. Bag 1 was handled as follows.

1. **Event e_1 (time t_1).** The bag was merged by unit c_1 onto P^2 with the empty ("-") process step outcome and flag "Success".
2. **Event e_3.** The bag was diverted by unit s_1 ("-", "Success") toward S_1.
3. **Event e_4.** The bag obtained clearance ("Level 1 clearance", "Success") at scanner S_1.
4. **Event e_7, e_9, e_{11} and e_{12}.** The system tried to divert bag 1 onto each link going to the final sorters but failed due to the unavailability of units $r_1 - r_4$ ("-", "Unavailable"). As a result, it started another round on P^2.
5. **Event e_{13}.** The bag was identified by camera a, that (re-)identifies all bags recirculating on P^2. It recognized the bar code attached to the bag on check-in, and associated the bag with encoded attributes "KL812;Economy;LED", i.e., flight "KL812" with the final destination in the airport "LED", the passenger class was "Economy". As this step succeeded, the resulting flag is "Success".
6. **Events e_{14}, e_{15}, time t_3.** Finally, bag 1 passed s_1 and was successfully diverted by r_1 ("-", "Success").

Another bag 2 entered P^2 later (e_2) and followed a similar path but was successfully diverted at r^2 without an extra round on P^2 (the events in Table 1.1 shows the events ordered by time). That is, bag 2 entered P^2 later than bag 1 but left it earlier, effectively overtaking bag with $id = 1$ due to temporal unavailability of $r_1 - r_4$. These two sequences of events show how the bags *competed* for shared resources $r_1 - r_4$, and how the bag that entered P^2 last "won" the competition. Note, although such interpretation is possible assuming some domain knowledge of the system, the event data in Table 1.1 do not describe TSU competition explicitly.

Note, not all events generated during the bag handling are shown in Table 1.1. For example, events from locations $c_2 - c_4, d_1, d_2$, and f are missing. In MHSs, events can

Event ID	Bag ID	Timestamp	Location	Outcome	Flag
e_1	1	t_1	c_1	-	"Success"
e_2	2	t_2	c_1	-	"Success"
e_3	1	t_3	s_1	-	"Success"
e_4	1	t_4	S_1	"Level 1 clearance"	"Success"
e_5	2	t_5	s_1	-	"Success"
e_6	2	t_6	S_1	"Level 1 clearance"	"Success"
e_7	1	t_7	r_1	-	"Unavailable"
e_8	2	t_8	r_1	-	"Unavailable"
e_9	1	t_9	r_2	-	"Unavailable"
e_{10}	2	t_{10}	r_2	-	"Success"
e_{11}	1	t_{11}	r_3	-	"Unavailable"
e_{12}	1	t_{12}	r_4	-	"Unavailable"
e_{13}	1	t_{13}	a	"KL812;Economy;LED"	"Success"
e_{14}	1	t_{14}	s_1	-	"Success"
e_{15}	1	t_{15}	r_1	-	"Success"

Table 1.1: Events generated by the handling of bags with identifiers 1 and 2.

be not recorded due to various reasons that we discuss later. These events are called *unobserved*. They introduce *incompleteness* of recorded event data, which usually affects the quality of analysis and monitoring, based on such data.

Next, we consider how events, recorded by various processes, are usually organized into data structures convenient for the process analysis, and what terms are used to refer to these data structures and their elements.

Process Instances, Traces, and Event Logs. Process steps are usually executed over individual entities, often referred to as *cases*. The processing of a single case, e.g., the handling of a particular bag by a material handling process, is referred to as a *process instance*. During its execution, each executing process step generates an event. For example, handling of bag 1 in the example above corresponds to a process instance, where process steps $c_1, s_1, S_1, r_1, \ldots, r_4, a, s_1, r_1$ were executed over the case of bag 1, and generated the event sequence

- $\langle e_1, e_3, e_4, e_7, e_9, e_{11}, \ldots, e_{15} \rangle$.

Such a sequence is called a *trace*. In a trace, the events preserve the execution order of the process steps that generate them. To organize events into traces, the following attributes are usually recorded for the events:

1. *case identifier*, in order to group the events into traces by the value of this identifier;
2. *timestamp*, for ordering events in traces.

No additional attributes are required to distinguish the traces of different cases. However, the information of the performed process step called an *activity label*, is usually

needed for most process analysis techniques and is often considered as a trace mandatory attribute. In Table 1.1, column "Bag ID" contains case (bag) identifiers, column "Timestamp" contains the event timestamps, and column "Location" contains locations that can be interpreted as activity labels.

When particular attributes of a set of events are considered as the case identifier and timestamp, these event data can be seen as the set of traces for a *case notion* defined by the chosen case identifier attribute. Such a set is called an *event log*. For example, the event table in Table 1.1 can be seen as the event log for case notion *Bag ID*, containing traces

- $\sigma_1 = \langle e_1, e_3, e_4, e_7, e_9, e_{11}, \ldots, e_{15} \rangle$, and
- $\sigma_2 = \langle e_2, e_5, e_6, e_8, e_{10} \rangle$.

Note different event attributes can be chosen as different case notions for the same events, thereby materializing the traces of different entities in the event data. For example, choosing attribute "Location" as a case notion for events in Table 1.1 results in traces describing how machines at each location handled bags. For example, the trace of the scanner at location S_1 (i.e., a trace with case identifier S_1 for case notion "Location") is an event sequence $\langle e_4, e_6 \rangle$. Event data in different forms are extensively used as input for various kinds of process analysis and monitoring techniques, which we consider next.

1.2 Techniques for Analysis of Material Handling Processes

As MHSs have been used for decades, there are established techniques for their performance analysis. These techniques include the design and analysis of MHS simulation models, the creation and use of dashboards with statistics describing TSU handling, and the modeling and analysis of MHS parts using queueing theory. The recent rapid progress of storage technologies made it affordable to store large amounts of system event data describing material handling. As a result, data-driven techniques started to be used. Nowadays, analysis of recorded system events using SQL and spreadsheets like Microsoft Excel is a common practice, while various data science techniques are applied as well. In the following, we consider the established and new techniques driven by data and then identify the knowledge gap they need to bridge when applied in the MHS domain.

1.2.1 Established Techniques

MHSs are classically studied using *queueing networks* and *simulation*. We consider both in the following.

Queueing Theory. Queueing theory allows modeling MHSs in the terms of *queue models*. *Jobs* arrive at a server according to an *arrival process*. If the server is busy, the jobs not processed yet wait in the server's *queue* until it is available. Various queue models have different parameters of their arrival process, server *service time*, *queue discipline*, *queue capacity*, etc., thereby comprising a variety of queue model types. However, a single queue model is capable of describing only a simple system [16]. *Queueing networks* are used for modeling complex systems. A queueing network comprises multiple queue models (nodes), and a *routing function*, defining the *probability* of transferring jobs from one node to the others, and outside the network. If a queue has a *finite capacity*, i.e., it can be full, so no jobs can be routed to it. If a queueing network has such queues, it is called a queueing network with blocking, and the blocking behavior [17] is additionally defined. Further, queueing networks are defined either as *open* or *closed*. The former is for studying the system for various scenarios of arrival processes, while the latter is for studying the system under certain load conditions when the job number in the system is constant.

Let us show how these concepts can be used for modeling sorting loop P^2 (Figure 1.2). In this BHS,

- the system resources, such as scanners $S_1 - S_k$, merge units $c_1 - c_4$, divert units $s_1 - s_k$, etc. can be seen as servers,
- conveyors can be seen as queues with FIFO ordering (as TSUs cannot overtake each other on a conveyor), and
- the system layout and routing can be seen as a routing function, thereby defining a queueing network.

In this network, each node represents a conveyor with a resource operating at its end, for example, a conveyor (c_1, c_2) (of capacity 10 bags) with merge unit c_2 at the end. As the conveyor capacity is always finite, it is a network with blocking, e.g., if conveyor (queue) (c_1, c_2) is full, it cannot receive new TSUs, and blocks all the TSUs waiting for merging onto it on conveyors (a, c_1) and (c_1', c_1).

Among works addressing the MHS performance analysis using queueing theory, we identified recent works supporting queueing network with blocking [18, 19, 20]. They exploit the analogy between TSUs on conveyors and vehicle traffic, and the *speed-density effect*, observed on roads [21] (vehicle speed drops when traffic becomes heavier). The same effect is observed in MHSs as well, i.e., TSU travel time increases as the conveyor load increases [18]. It is caused by spatial limitations, dictating how TSUs can be placed on conveyors, e.g., the need for free *protective* space between the neighboring TSUs on the same belt. To model this effect, each conveyor is modeled as a state-dependent queue [18], i.e., a queue whose waiting time depends on the number of jobs in the queue such that a greater job number causes a longer service time.

Even assuming that a state-dependent queue can accurately model an MHS conveyor, other challenges for modeling MHSs with queueing networks still remain. For example, routing between BHS conveyors is driven by complicated algorithms considering the whole system state, i.e., all bags in the system and all equipment, and it can be hardly modeled through routing probabilities. Additionally, external factors, like changes in the flight schedule, affect it dynamically. These and other challenges make the use of queueing networks for the analysis of MHS notoriously hard and infeasible in practice. So, simulation-based analysis is typically used instead.

Simulation. The MHS performance can be studied in the *fully controllable environment* of a *simulation model*. For example, problem-oriented simulation models allowed the identifying of bottlenecks and critical operations for inbound baggage handling [22] and automatic security screening [23], learning dependencies between security policies and time characteristics of manual baggage screening [23], and so on. While such simulation models can be precise, their design requires in-depth knowledge of a system design and proved to be time-consuming. Moreover, the number of replications needed for covering various variants of each scenario of interest exponentially grows with the system size. As a result, in practice simulation of MHSs is commonly used for a limited number of scenarios, and for either relatively small systems or for particular areas (parts) of larger MHSs. As a result, simulation does not fully solve the problems of MHS performance analysis.

In the next section, we discuss how data-driven techniques aim to overcome the drawbacks of queueing theory and simulation.

1.2.2 Process Mining

Recently, the possibility to store and analyze large amounts of data due to decreased costs of data storage solutions triggered the rapid growth of data-driven technologies, including *process mining* studying business processes through the event data they generate [1]. In the following, we consider the main types of process mining.

Across various industries, business processes are the subject of improvements, compliance checking, and automation activities [1, 24, 25]. Process mining is the field of data science that addresses the two former problems directly, and helps to automate processes in a conjunction with other techniques. Process mining has two key characteristics that distinguish it from the other disciplines studying processes [1]:

1. process mining is event data-driven, i.e., it studies processes from event data generated by execution of process steps of the system or process of interest rather than assuming its behavior from domain knowledge,
2. most process mining techniques are process model-centric, i.e., they take a process model as input to reason about the information captured in event data.

These properties allow process mining techniques to obtain insights into the process behavior, and various findings that (1) reflect the real behavior, and (2) are unbiased, i.e., they are not affected by existing assumptions of the domain experts. The obtained results, in turn, are used for making the process compliant, and efficient, and to identify process steps, and/or sub-processes for potential automation.

Figure 1.3 shows the core process mining types and artifacts, comprising a typical process mining architecture, as follows.

- A system or process *records* (or *streams*) event data, that are typically represented in the form of an event log. An event log usually contains information about process steps execution, their observed sequences, and performance. Most process mining techniques assume a complete event log, i.e., a log where all events generated by executed process steps are recorded.
- Various event data-driven *process discovery* algorithms take an event log as input to create a generalized process description in the form of a *process model*.
- *Conformance checking* relates event data to the process model to detect outliers, which are used for
  - taking countermeasures to address frequent deviations,
  - *model repair* if the model is considered as *descriptive*,
  - *log repair* if the model is considered as *prescriptive* [24], or
  - outlier analysis.
- A process model can be *enhanced* by projecting various performance information, derived from event data.
- The process performance can be *analyzed* using the discovered process model and given event log.
- Last but not least, an event log can be used for deriving features for training various predictive models.

Obtained analysis results and outliers can be used for process improvements.

Similarly, predictions are used by an analyst (operator) or operating software to prevent undesirable scenarios. The circle arrows in Figure 1.3 indicate that these process mining tasks are performed iteratively, gradually enabling a better process, more accurate event log, process model, and predictive models, i.e., this architecture allows for continuous process improvement through repeating iterations [26]. However, the described process mining architecture is highly generic. Each process requires the use of concrete process mining and process modeling techniques to solve actual problems.

In Section 1.1.3, we considered what challenges MHS operators are facing for operational support, and why it is crucial to keep the MHS performance on a desired level. Process mining types, shown in Figure 1.3, can be potentially used for addressing these challenges. However, historically process mining considers processes with isolated cases. As a result, applying process mining techniques that assume

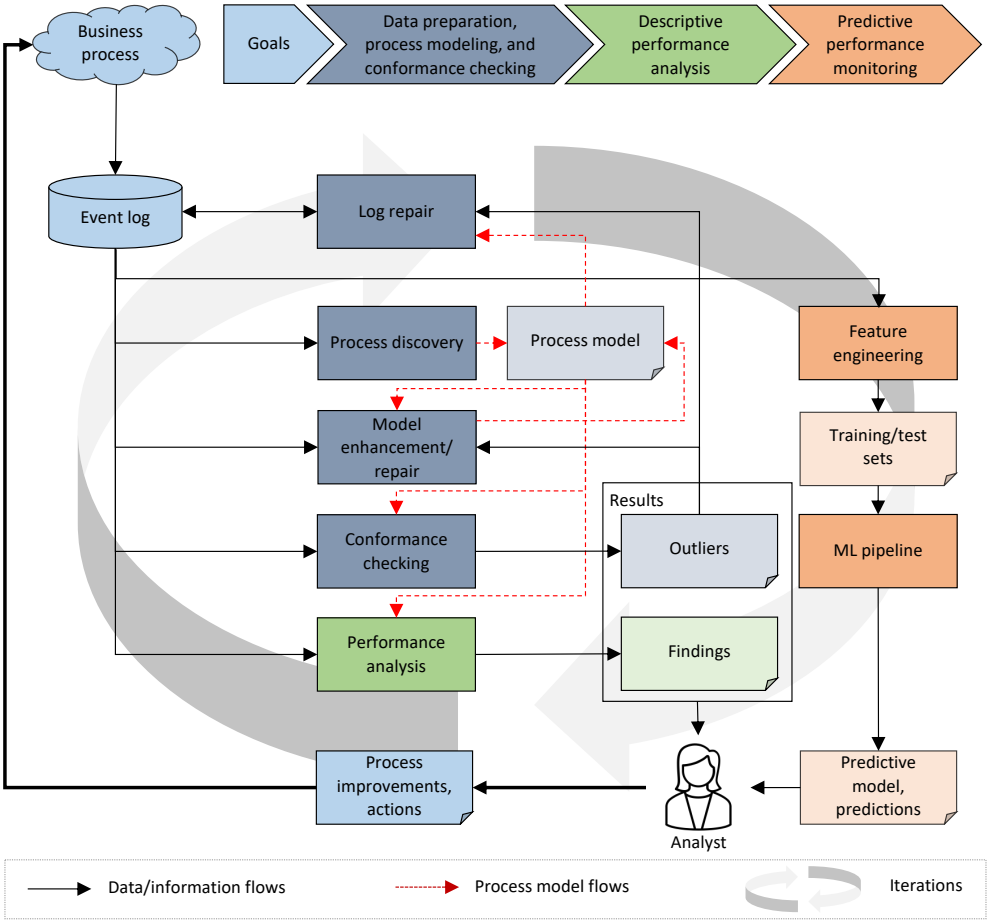


Figure 1.3: Main types of process mining.

processes with isolated cases fails to obtain tangible results for processes with non-isolated cases, like material handling processes.

In the next section, we discuss why the state-of-the-art queueing theory and process mining techniques are not currently applicable for the analysis of MHS processes in detail.

1.2.3 Knowledge Gaps in Techniques for Analysis of Processes with Non-Isolated Cases

In this section, we consider queueing theory and process mining, which we identified as the primary choice for MHS analysis. We discuss what is still missing in their tech-

niques to address the challenges we formulated previously and conclude by choosing process mining for research in this thesis.

Queueing Theory. Previously, we considered the state-of-the-art queueing theory techniques for MHS analysis capable of modeling cases interacting through queues and their networks with blocking [18, 19, 20], and a speed-density effect caused by spatial limitations [21, 27]. Although these approaches effectively model simple MHSs under certain conditions, they have serious limitations for practical use. For queueing networks, the possession of a *product-form solution* is the key characteristic [17]. It is an analytical formula for the *queue-length distribution* at the network nodes. Its existence enables various techniques, for example, aggregation and decomposition results for product-form queueing networks yield Norton's theorem for queueing networks [28, 29], and the arrival theorem implies the validity of mean value analysis for product-form queueing networks [17, 30]. Conversely, if there is no product-form solution for a queueing network available, its non-analytic analysis becomes notoriously difficult and usually infeasible. For the works we identified, the product-form solution is available under the assumption that the arrival process in *each* node is a *Poisson* arrival process. However, as we show in Chapter 4, this is not usually the case, as the routing in MHSs is not random but depends on the overall system state. That makes this approach inapplicable for the performance analysis of the real-world MHSs used in logistics.

Besides that, most queueing theory approaches, including the one we consider [18], are designed for the analysis of systems in the *equilibrium state* rather than in *transient state* [31]. An equilibrium state means that the number of jobs in the system does not depend on time, otherwise, the system is in a transient state. This assumption does not hold for MHSs due to flight schedule changes (in BHSs), load peaks, and many other factors. Last but not least, queueing theory techniques cannot use event data, recorded from a process execution, for the postmortem performance analysis. For example, we cannot "replay" an event log, recorded from an undesirable performance scenario in a BHS, over a queueing network to analyze individual paths of each bag, bag interactions over time, availability of resources, etc.

Next, we consider what is missing in process mining, whose techniques, unlike queueing theory ones, are designed to be driven by recorded (or streamed) event data.

Process Mining. The process mining architecture, shown in Figure 1.3, comprises different process mining types for the fulfillment of various process analysis objectives. Among them, descriptive and predictive analysis directly match the challenges of MHS analysis formulated in Section 1.1.2. However, process mining techniques mainly consider processes with isolated cases. For example, in Section 1.1.2, we discussed how cases (tickets) of the RTFM process *do not* interact during the initial

process steps, i.e., how they are isolated. However, when they *do* interact during *later* process steps, existing process mining methods fail to describe such case interactions.

For example, aggregate durations of process steps, projected on the process model in Figure 1.4(a), are unaware of the case dependencies and fail to describe the complex behavior, shown in Figure 1.4(b) in the form of a *performance spectrum* that we propose in Chapter 3. This performance spectrum reveals how tickets were accumulated in batches

- to be printed out and sent by post to the violators (hourglass-shaped patterns in the spectrum), and
- to be sent to a credit collection company if a violator refuses to pay (triangle patterns at the spectrum bottom).

Within the RTFM process, this batching is needed to save the working time of costly human resources handling the paperwork. As a result, cases (tickets) usually wait till these limited shared resources become available. Similarly, existing methods fail to reveal much more complicated behaviors of TSUs in MHSs.

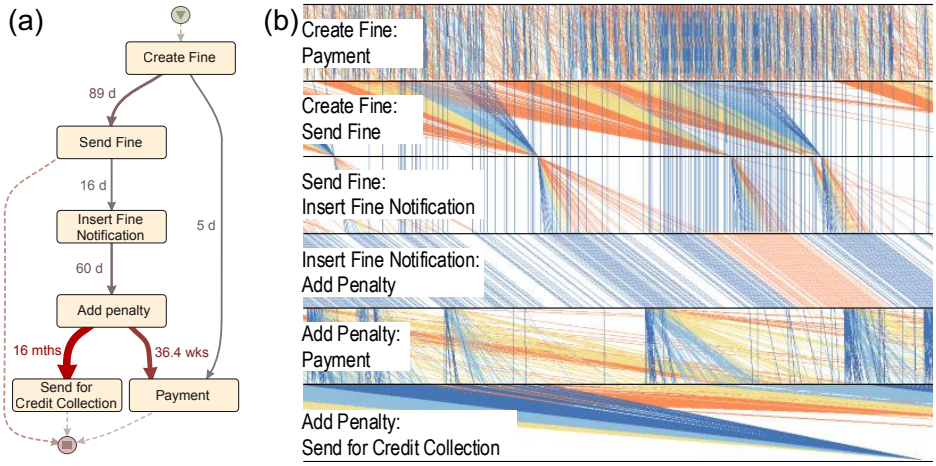


Figure 1.4: Performance analysis using a graph-based model (a), and the performance spectrum (b).

For predictive analysis, the typical problems of predicting the *next case process step*, and *case remaining time* have been studied in many works [32, 33]. To address them, various ways of identifying and extracting features, relevant to training predictive Machine Learning (ML) models, are suggested in process mining works [34, 35, 36, 37, 38]. With process mining, such features are identified using information about the actual process behavior over time. Thus, for predicting the next case process step,

the control-flow perspective of processes significantly contributes to identifying and extracting relevant features.

For example, the trace *prefix*, i.e., a *sequence* of already performed (observed) process steps for a case, can help predict the next process step [39]. Then, for predicting the remaining case time, dependencies between process step durations and trace prefix variants can be effectively used to train corresponding models. However, in processes with non-isolated cases, this information is much less relevant than for processes with isolated cases. For example, the remaining case time in the RTFM process significantly depends on which batch a case is handled in [40, 41]. It depends on the dynamics of case accumulation. In MHSs, both the next process step and the remaining case time depend on the load in system areas, resource availability, TSU spatial configuration, etc. Moreover, these conditions continuously change over time. So, current process mining techniques essentially ignore most phenomena caused by inter-case dynamics and are not helping much in the analysis of such processes.

Last but not least, a process model is a central artifact of the process mining architecture. First, a process model itself provides insights into the process it models. Various visualization techniques *replay* recorded, or project ongoing cases on the model for providing insights into case execution over time. Then, conformance checking techniques relate data recorded from the process execution to a process model for outlier analysis, model enhancement, or data preparation tasks like log repair. These techniques usually expect a Petri net [42] as a process model. A Petri net initial marking (state) is typically defined as a single token representing a single case, i.e., a Petri net in process mining is typically used to describe the execution of a single case at a time [1].

For example, for a BHS, such a model usually describes the system layout (the map of conveyors) with nodes corresponding to the equipment locations. Although it is capable of correct modeling and visualizing possible paths of bags throughout the system, it does not describe any resources, and how cases interact on them. As a result, even a single BHS conveyor cannot be described by such a model in a way that captures the behavior of more than one bag on it at a time, while visualization techniques are incapable of helping a human to spot the root causes and development of undesirable performance scenarios.

However, can a Petri net model with a single token in its initial marking still be valuable for conformance checking, model enhancement, and data preparation in the MHS domain? Yes, but in a very limited way. For example, trace alignment [43], the state-of-the-art technique for conformance checking, does not consider any constraints related to resource characteristics, and the ways in which cases interact. As a result, physically impossible behaviors can be seen in traces aligned with a process model. Moreover, incomplete logging, typical for modern MHS architectures, dramatically increases the chances of such errors. For example, traces with some unobserved (unrecorded) events after alignment can describe the number of bags on a conveyor

that cannot be physically fitted there, or even bags overtaking each other while they are being moved on the same conveyor. These errors can dramatically impede the conformance checking outcome, and model/log repair. Alternatively, an appropriate process model and trace alignment algorithm must consider, for example, conveyor capacity constraints while reducing the trace alignment search space, to avoid a priori impossible alignments.

To summarize, both queueing theory and process mining have the potential for the analysis of processes with non-isolated cases but certainly have a gap in knowledge of how to address the challenges we previously discussed. As a result, their state-of-the-art techniques yield relatively modest practical value when applied to processes of this type. All things considered, process mining covers these challenges better, and the gap seems to be smaller and easier to bridge. The following section discusses how this thesis is designed for that.

1.3 Research Questions and Solution Approaches

In this section, we formulate Research Questions (RQs) and outline our methods for solving them. However, let us first scope the research of this thesis.

As this is the first systematic study of material handling processes (with non-isolated cases) using process mining,

- we focus on MHSs without batching (and not on MHSs with batching) to limit the scope,
- we assume the process model itself to be known from the system design documentation, so we do not aim to create a process model discovery technique.

To structure this thesis, we design the RQs to "customize" the problems for the main types of process mining, shown in the diagram of Figure 1.3, for processes with non-isolated cases. These RQs assume analyzing from scratch, i.e., nothing but the system's informal description, and analysis goals are known a priori. The version of the diagram Figure 1.3, "customized" for MHS analysis, is shown in Figure 1.5. From now, we consider only processes with non-isolated cases interacting on shared resources.

In Figure 1.5, the general analysis is shown as a path from the initial goals, via descriptive performance analysis toward predictive performance monitoring. Further, a process with non-isolated cases, i.e., a material handling process to be analyzed, is presented as a cloud. It generates an event log (or event table that we define later) that is the input for all data-driven approaches of this thesis, starting with **RQ-1** formulated as follows.

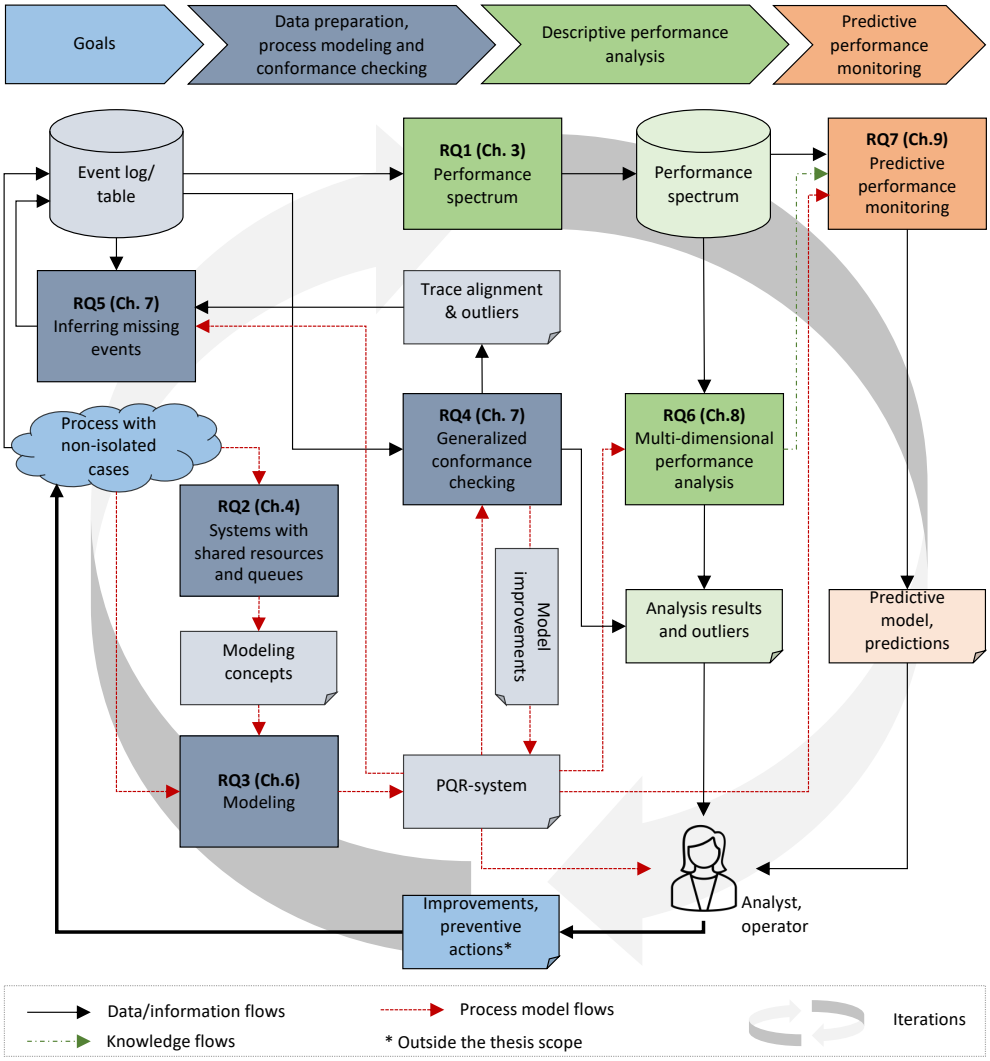


Figure 1.5: Overview of research questions and methods.

- **RQ-1**. *Given an event log of a process, how to describe the performance information the log contains in a way that reveals both the performance of individual cases and how these cases interact over time?*

To approach **RQ-1**, we propose the *performance spectrum* in Chapter 3 (see Figure 1.5), which is a fine-grained process performance description showing how cases interact both during the same process step and across different ones. It is a model-less technique, which can be used as a generic stand-alone visual analytics tool for the

performance analysis of processes from event data. In the following, we use the performance spectrum as input for other approaches because it captures well interactions of non-isolated cases.

The process mining architecture in Figure 1.3 shows that most process mining techniques are model-centric, i.e., they *benefit* from a given process model. The problem of modeling processes with non-isolated cases we approach in two steps, as the path from **RQ-2** to **RQ-3** shown in Figure 1.5.

In Step 1, we identify key concepts for such a model using

- actual analysis questions,
- domain knowledge about MHSs, and
- findings about MHS performance presented in Chapter 3 (**RQ-1**),

to address the following RQ in Chapter 4.

- **RQ-2**. *Given the informal description of a process with non-isolated cases, and the questions of its* descriptive *and* predictive *performance analysis, how to identify the key entities and concepts required for modeling case interactions in a way facilitating answering these questions?*

Additionally, we introduce the term *systems with shared resources and queues* to refer to the class of systems/processes that we study in this thesis.

In Step 2, we define the process model in Chapter 6, using the outcome of Chapter 4, by addressing an RQ that reads as follows.

- **RQ-3**. *Given the modeling concepts obtained by answering **RQ-2**, and the problems of descriptive and predictive performance analysis of systems with shared resources and queues, how to design a process model facilitating answering these questions?*

We approach **RQ-3** by proposing a Process-Queue-Resource System (PQR-system) that describes not only the control-flow (process) perspective but also MHS resources and queues for modeling case interactions. We also define the PQR-system *replay semantics*. Note, **RQ-3** does not aim at the creation of a process model *discovery* technique but assumes that existing system documentation can be used to generate the corresponding PQR-system.

The PQR-system enables the core model-centric data-driven process mining techniques for processes with non-isolated cases. Thus, to relate event data to the PQR-system, we address the problem of conformance checking in Chapter 7 as follows.

- **RQ-4**. *Given event data generated by a process with non-isolated cases in the form of an event table, and the PQR-system, how to relate the data and model to determine if the model correctly describes both the process for individual cases and case interaction observed in the event data, and if the event table fits into the behavior described by the PQR-system?*

However,

- most process mining techniques assume (and benefit) *complete* event logs, but MHSs usually generate incomplete ones, and
- if MHS documentation is outdated, the generated PQR-system is incorrect.

Because such errors must be detected before the system analysis, we go beyond classical conformance checking when addressing **RQ-4**. We adopt the concept of *generalized conformance checking* [44] for PQR-systems and material handling processes, which allows for doing both model and log repair tasks, instead of choosing whether a model is descriptive or predictive, as with "classical" conformance checking. Additionally, we address the problem of data incompleteness via log repair in Section 7.3 by inferring unobserved events with timestamp information. The corresponding **RQ-5** reads as follows.

- **RQ-5**. *Given event data generated by a system with shared resources and queues in the form of an event table where some events are* unobserved *(missing), and the PQR-system, how to reconstruct the unobserved events, including their timestamps, so that the resulting event table can be replayed over the given PQR-system?*

The resulting method transforms a given incomplete event log (table) into a complete one, which is shown in Figure 1.5 by the path from the event table via **RQ-5** back to the event table. The resulting complete event table can be successfully replayed over the PQR-system (according to its replay semantics).

The (repaired) PQR-system and complete event table, obtained after generalized conformance checking, are used as input for the methods answering the consequent RQs, which address performance analysis problems.

Thus, the descriptive performance analysis problem is addressed in Chapter 8 as follows.

- **RQ-6**. *How to relate a PQR-system, describing the process (control-flow), queue and resource dimensions of a process/system, and the corresponding performance spectra computed from complete and correct event tables?*

The method of Chapter 8 serves to improve processes through their post-mortem analysis. However, no improvements cannot guarantee the absence of undesirable performance scenarios. Our final RQ addresses the problem of predictive performance monitoring (Chapter 9) to prevent such scenarios, as follows.

- **RQ-7**. *Given a PQR-system, a complete event table, an aggregate process performance indicator, and a prediction horizon, how to predict this indicator so that both its gradual and sudden changes are predicted?*

We discuss these methods in more detail and summarize the thesis contributions in the next section.

1.4 Thesis Overview

Most results, discussed in this thesis, have been published in peer-reviewed conferences, journals, and workshops. The remaining chapters of this thesis are structured along the process mining workflow, shown in the diagram of Figure 1.5, which groups the related RQs, chapters, and methods in the boxes as follows.

Preliminaries in Chapter 2. We start by introducing the basic notations and key concepts, such as sets, multisets, partial orders, Petri nets, and colored Petri nets. We define events, traces, and event logs. We also define event tables, which can be seen as multi-case notion event logs. These notations, concepts, and definitions are extensively used throughout the subsequent chapters.

Performance Spectra in Chapter 3. We address **RQ-1** by introducing and defining the *performance spectrum*, a data structure and visual analytics technique for describing a system or process performance. It is computed from an event log whose directly following events in traces form *segments*. Each segment is a pair of activities (a, b) that represents a transition from activity a to b, handover of work from resource a to b, or movement of materials from location a to b. A performance spectrum comprises one or multiple *segment* spectra. In a nutshell, the spectrum of a segment shows the aforementioned along the time axis for all the process instances presented in the given event log, thereby describing their performance.

Let us show an example of the performance spectrum built from the log of the undesired performance scenario on the sorting loop P^2 (Figure 1.2), discussed in Section 1.1.2. This spectrum, shown in Figure 1.6(a), has two segments. Segment (c_3', c_3) corresponds to the conveyor of an incoming check-in link that connects the check-in area with sorting loop P^2 (see also Figure 1.1). Segment (c_3, s_1) corresponds to the path from the merge unit c_3 to diverting unit s_1. In the spectrum, each line, called a *segment occurrence*, shows the transition from the first process step of the segment to the second one for a single case along the time axis. Longer lines correspond to longer durations, and shorter ones correspond to shorter ones.

Additionally, each segment occurrence has an assigned *performance class* to make reading the performance information easier for humans. In Figure 1.6(a), these performance classes are {*normal speed*, *2 times slower*, *3 times slower*, and *very slow*}. Each class shows how fast a bag was moved from location c_3' to c_3 (top segment), or from c_3 to s_1 (bottom segment). For segment (c_3, s_1), we see a period of regular processing (blue lines), then a period of slow processing (yellow lines) that ends after several minutes (see the timeline), when regular processing resumes. In our example, the slow processing period corresponds to the time when sorting loop P^2 was stopped due to the bag that got stuck at s_1. As a result, bags on P^2 were not moving at a constant speed as before but standing still until P^2 resumed. Yellow lines clearly show this situation. In the meanwhile, bags on (c_3', c_3) were waiting for the resume of P^2 as

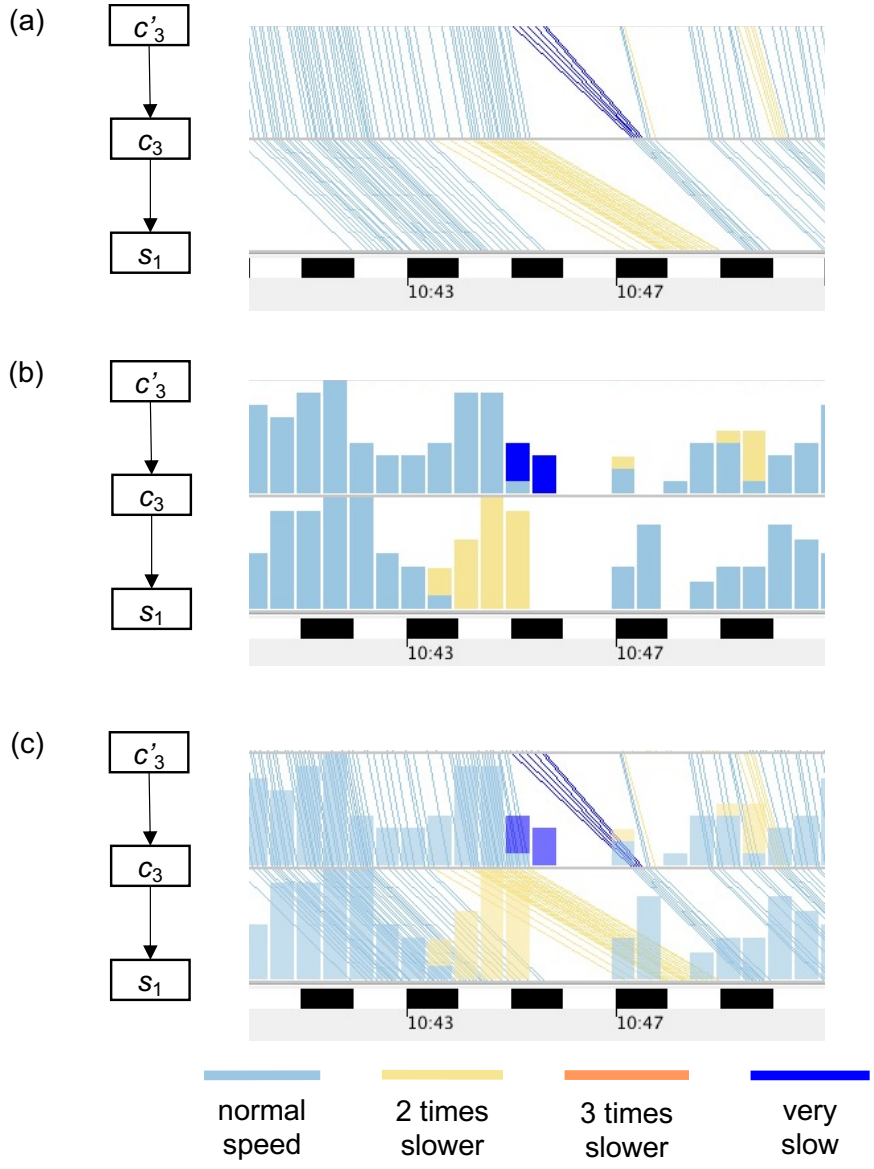


Figure 1.6: "Regular" performance spectrum (a), aggregate performance spectrum (b), and combined performance spectrum (c).

they could not merge onto the stopped sorting loop (dark blue lines corresponding to performance class *very slow*.

Interestingly, this example shows that each case (bag) can be followed along its full path from c_3' to s_1 because segment (c_3, s_1) directly follows (c_3', c_3). That is, it is

possible because the end of any segment occurrence line of (c_3', c_3) is the beginning of the segment occurrence line of (c_3, s_1) for the same case (bag). It also reveals the *interactions of non-isolated cases*, e.g., slow yellow cases on (c_3, s_1) delay dark blue cases on (c_3', c_3).

While the performance spectrum describes each segment occurrence in a detailed manner, the *quantified* information about these occurrences can be useful for analysis as well. For that, we propose an *aggregate* performance spectrum, which is a sequence of bins per segment. Each bin represents a histogram for a time window. In each histogram, a bar height shows how many occurrences started or ended within the corresponding time window or intersected it for the corresponding performance class. The way in which segment occurrences are assigned to a bin, meaning start, end, and intersection, is called *grouping*. The aggregate performance spectrum of Figure 1.6(a) is shown in Figure 1.6(b). If a performance spectrum is visualized over its aggregate (or the other way around), it is called a *combined* performance spectrum (see Figure 1.6(c)).

Both types of the performance spectrum separately and/or when combined reveal various *performance patterns* in processes, such as ordering, batching, various load patterns, etc. For example, the non-intersecting lines of segment occurrences in Figure 1.6(a) describes the FIFO order. We provide a comprehensive taxonomy for these patterns in Section 3.3.2.

The aggregate performance spectrum can serve as a building block for more complex data structures, containing more information about the process performance. Thus, we introduce the aggregate performance spectrum built for the same segments but different performance classifiers and/or grouping, comprising different *channels*, called a *multi-channel* performance spectrum. The details are provided in Chapter 3, which is based on [45, 46, 47].

The nature of MHSs in Chapter 4. We address **RQ-2** by investigating how MHSs can be modeled using queueing networks, and the limitations of such approaches. For modeling, the starting points are

1. a typically available description of an MHS in the form of an MFD (see Figure 1.7) that describes the system stations, servers, conveyors and their layout, and
2. the dynamics of interest, consisting of the blocking behavior of TSUs on the system conveyors, aspects of the TSU placement policy, including the TSU spatial configurations, and the need to analyze the behavior of each individual TSU rather than their aggregate characteristics.

As we focus on processes with non-isolated cases, it is crucial that a resulting model captures inter-case dynamics that can be revealed and explored using the performance spectrum. So, our approach is to

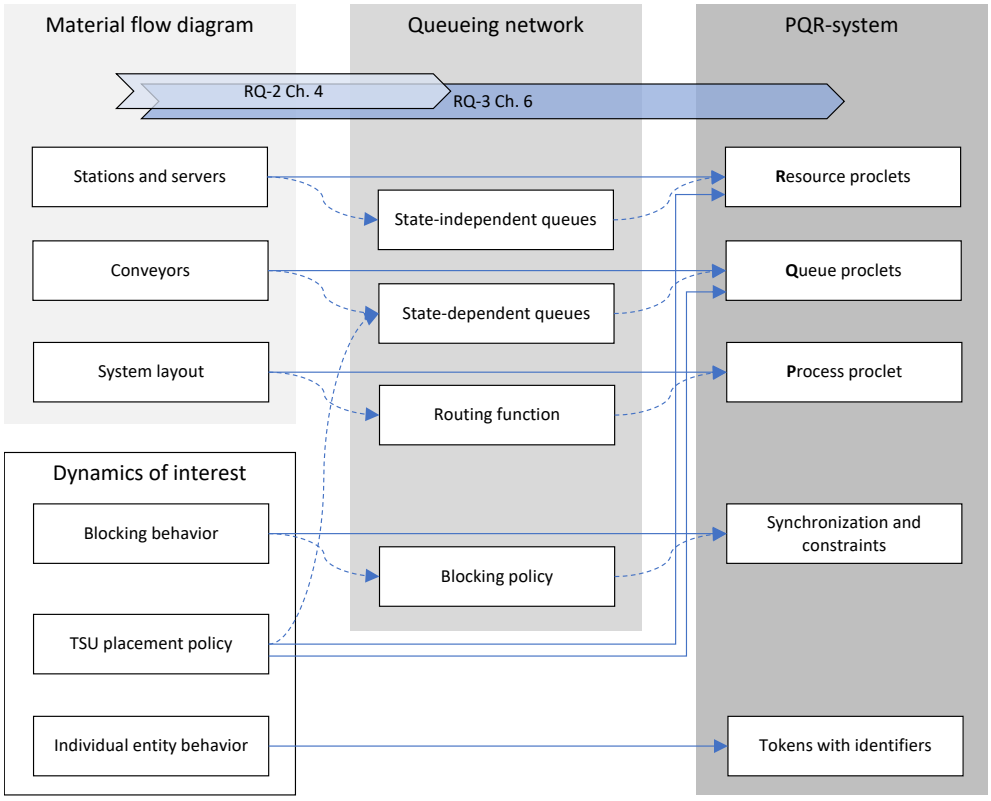


Figure 1.7: Modeling approach in Chapter 4 and Chapter 6.

1. investigate these dynamics with performance spectra built from real-world MHSs,
2. map the MHS entities and performance phenomena learned from the performance spectra, whenever it is possible, to the queues of different types, a routing function, and blocking policies of queueing networks (Figure 1.7),
3. check whether the state-of-the-art approach for modeling MHSs as queueing networks [18, 20] can capture it.

For example, the queueing network modeling the BHS fragment consisting of conveyors $(c_2', c_3), (c_3', c_3)$ and (c_3, c_4) accordingly to this approach [18, 20] is shown in Figure 1.8. In this model, units c_2', c_2, c_3, and c_4 are modeled as classical *M/G/1/K* nodes, and the conveyors are modeled as state-dependent *M/G/c/c* queues. This queueing network represents:

- the system layout,

- conveyor final capacities (parameter c of *M/G/c/c* queues) and blocking behavior caused by them,
- service time of resources (units) c_2', c_2, c_3 and c_4,
- spatial limitations on conveyors $(c_2', c_3), (c_3', c_3)$ and (c_3, c_4) through the speed-density effect modeled by the state-dependent *M/G/c/c* queues.

However, this model works only under conditions that do not hold for most real-world MHSs.

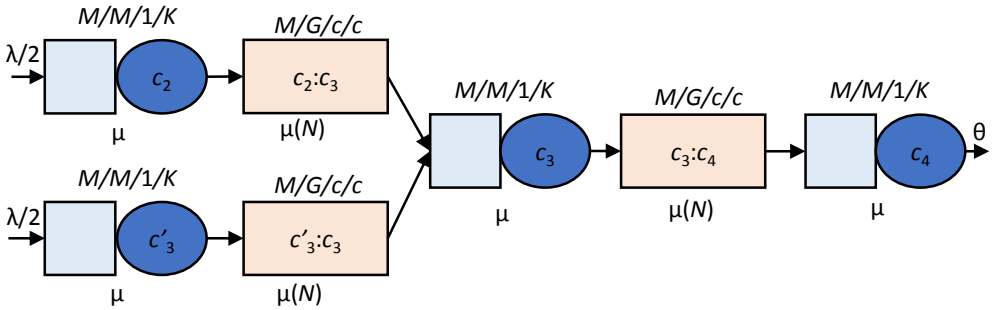


Figure 1.8: Combination of *M/G/1/K* and *M/G/c/c* queues models the BHS conveyors of Figure 1.2.

As a result, we identify the concepts for modeling MHSs (see the mapping in Figure 1.7), but conclude that no queueing network type can really capture the dynamics of interest. Additionally, we introduce the term *systems with shared resources and queues* to refer to the class of systems/processes we study in this thesis.

The PQR-System in Chapter 6. We answer **RQ-3** by modeling systems with shared resources and queues as a *Process-Queue-Resource system* (PQR-system). This dedicated synchronous proclet system [48] is tailored for the needs of MHS modeling and consists of colored Petri net models [49] called *proclets* whose transitions can synchronize via *synchronization channels*. These proclets and channels represent a system using the concepts identified in Chapter 4 as follows:

- the routing function is modeled as a *process proclet*,
- queues are modeled as *queue proclets*, and
- servers are modeled as *resource proclets*.

The blocking behavior due to TSU placement policies, which is not fully captured by the queueing networks we considered, is described on the level of proclet synchronization and in the models of the queue and resource proclet. Additionally, the individual entity behavior, which cannot be modeled through queueing networks, is modeled in the proclet system by introducing tokens with unique identifiers.

A PQR-system example, which models the same BHS fragment as the queueing network in Figure 1.8, is shown in Figure 1.9. In this figure,

- the process proclet (red) describes the system layout and routing, i.e., how bags can enter the system at c_2 and c_3', merge at c_3, and exit at c_4,
- the resource proclet (green) models merge unit c_3, and
- the queue proclets (blue) model queues (conveyors) $c_2 : c_3$, $c_3' : c_3$, and $c_3 : c_4$.

Resource proclets of c_3', c_2, and c_4 are not shown for simplicity. The process, resource, and queue proclets *synchronize* through synchronization channels defined between the proclet transitions. They limit the ways in which cases (e.g., bags) can interact on the resources and queues throughout the system.

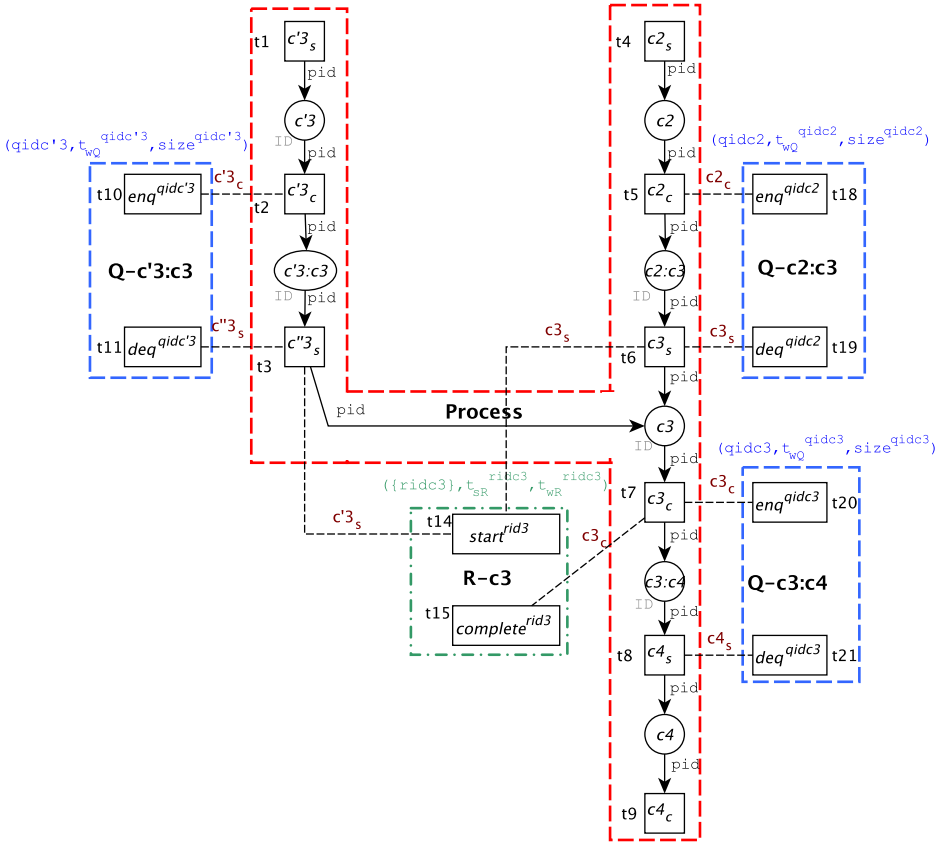


Figure 1.9: PQR-system example.

In Chapter 6, we define the PQR-system formally, as well as its replay semantics. This chapter is based on [50, 51].

Generalized Conformance Checking in Chapter 7. We address the problem of generalized conformance checking for systems with shared resources and queues, which gathers the tasks of conformance checking, model repair, and log repair under the common roof [44], in **RQ-4** and **RQ-5**. First, we show how to adopt this concept for our setting. Then we consider conformance checking and model repair tasks using the PQR-system and its semantics. For addressing **RQ-4**, we divide the problem of PQR-system conformance checking into two parts:

1. *trajectory conformance checking* of the P-, Q- and R-proclets,
2. and *synchronization conformance checking* on the proclet system level.

We reduce the problem of P-proclet trajectory conformance checking to the well-studied problem of conformance checking of Petri nets with black tokens [24]. For trajectory conformance of the Q- and R-proclets and synchronization conformance checking, the extension of data-aware Petri nets [52] is considered, in order to apply the existing approach [53].

To address **RQ-5**, we propose a novel method for inferring unobserved events with the timestamp information. For that, we first reconstruct unobserved events (without timestamps yet), using trace alignment [43] with the P-proclet. Then, we consider event tables with the case notions of the P-, Q- and R-proclets as a family of sequential event logs, and as a partial order. These views allow formulation constraints over timestamps of the unobserved (and reconstructed) events for defining a linear program, using

- ordering of events along the traces of the different case notions, and
- temporal parameters of the Q- and R-proclets.

As a result, the timestamp intervals of the unobserved (and reconstructed) events are inferred for these events.

The BHS, whose fragment is shown in Figure 1.2, records incomplete logs. All events generated by merge unit $c_1 - c_4, d_1, d_2$, and f are unobserved (not recorded) if a bag comes from the sorter and not from the incoming conveyors (c_1', c_1), (c_2', c_2), (c_3', c_3), (c_4', c_4), (d_1', d_1), (d_2', d_2) and (f', f). As a result, the recorded event table shows that bags often "skip" various process steps. For example, in the performance spectrum in Figure 1.6(a), bags going from c_3' via c_3 toward s_1 "skipping" process steps c_4, d_1, d_2 and f. As a result, this event table, as well as the performance spectrum, does not show the real bag behavior accurately. Such incomplete event data drastically impede any further analysis. Our log repair method reconstructs unobserved events and their possible timestamps. A performance spectrum built from the reconstructed complete event table is shown in Figure 1.10, where segments "swallowing" unobserved intermediate segments (like (c_3, s_1)) do not exist, but real segments (c_3, c_4), (c_4, d_1), (d_1, d_2), (d_2, f) and (f, s_1) are presented. For instance,

1. a bag bag_1 came from c_2 to c_3,
2. was delayed on (c_4, d_1),
3. continued at normal speed at (d_1, d_2) and (d_2, f), and
4. was delayed again on (f, s_1).

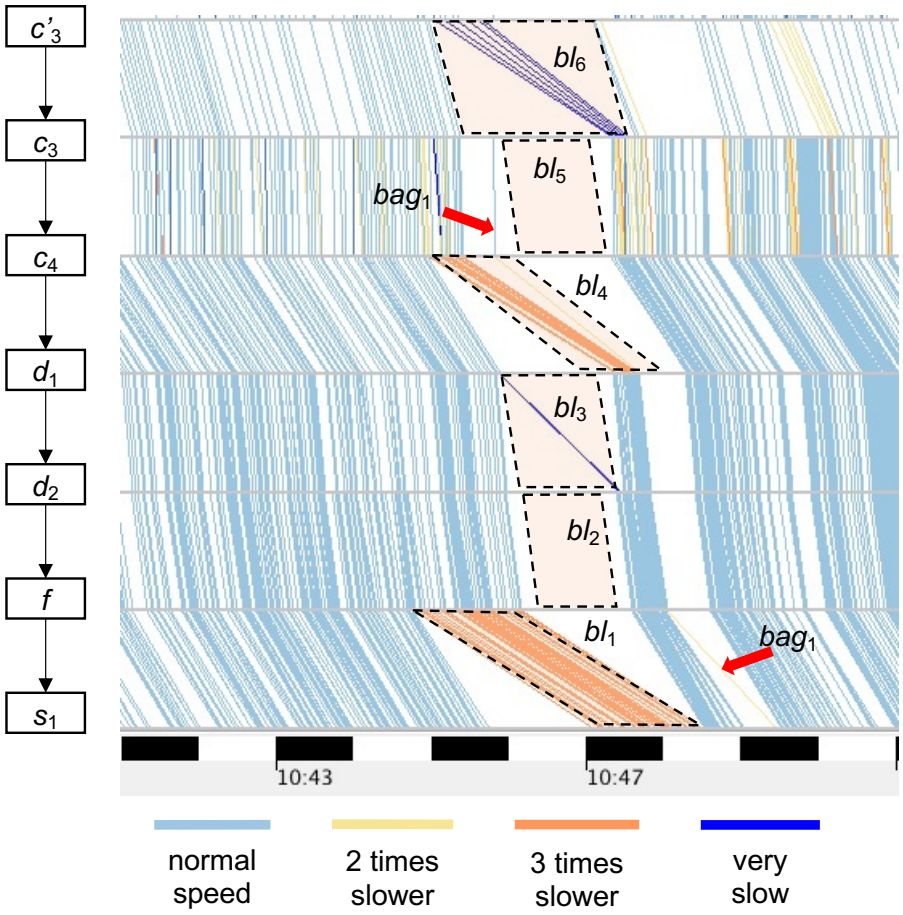


Figure 1.10: Performance spectrum built from a complete event table, i.e., an event table containing known missing events.

Finally, we organize these two approaches into a framework for generalized conformance checking, which can be used for addressing **RQ-4** and **RQ-5** in different real-world scenarios. This chapter is based on [50, 51].

Multi-Dimensional Performance Analysis in Chapter 8. For addressing **RQ-6**, we propose a PQR-system-based method for the performance analysis of systems with shared resources and queues using performance spectra. First, we relate performance spectra to the PQR-system. We define the performance spectra of the queue and resource dimensions, in addition to the performance spectrum of the control flow, and also show how performance spectra can be used to analyze synchronization among the process, queue, and resource proclets.

Afterward, we propose our method for multi-dimensional (meaning the process, queue, and resource dimensions) performance analysis of systems with shared resources and queues. It uses the PQR-system and performance spectra of the process, queue, and resource *together* to detect undesirable performance scenarios, like one we described in Section 1.1.2, and identify their root causes. We show that such a scenario is usually a *chain* of instances of performance patterns, proposed in Chapter 3 (**RQ-1**). The following patterns are frequently observed in the performance spectra of undesirable performance scenarios in MHSs:

1. *slow performance* called also *blockage*,
2. *high amount of workload* called also *high load*.

A blockage instance is a piece of a segment spectrum where cases were handled slower than usual. For example, blockage instance bl_6 in Figure 1.10 shows slower bags. A high load instance is a piece of a segment spectrum when a greater number of cases than usual were handled. Instances of this type are easier observed in an aggregate performance spectrum. In Figure 1.11, higher bars show high load instances hl_{1-8}.

Using these observations, we design the following steps for our method.

1. Detect blockage and high load instances in the performance spectrum of the control-flow dimension.
2. Discover their propagation chains and identify their initial segments (where the chains originate) using the PQR-system.
3. Detect blockage instances in the queue and resource performance spectra of the initial segments.
4. Map the combination of the detected instances to the root cause.

Note that the detection of performance patterns does not require a process model. However, the PQR-system is needed for understanding how instances caused each other to discover their propagation chains. Thus, a high load instance (e.g., a group of densely placed TSUs on an MHS conveyor) moves *forward* in the control flow dimension described by the P-proclet, dividing on split nodes (diverting units). For example, one part of bags in hl_8 in Figure 1.11 can continue on the sorting loop, and another part can be diverted toward scanner S_1.

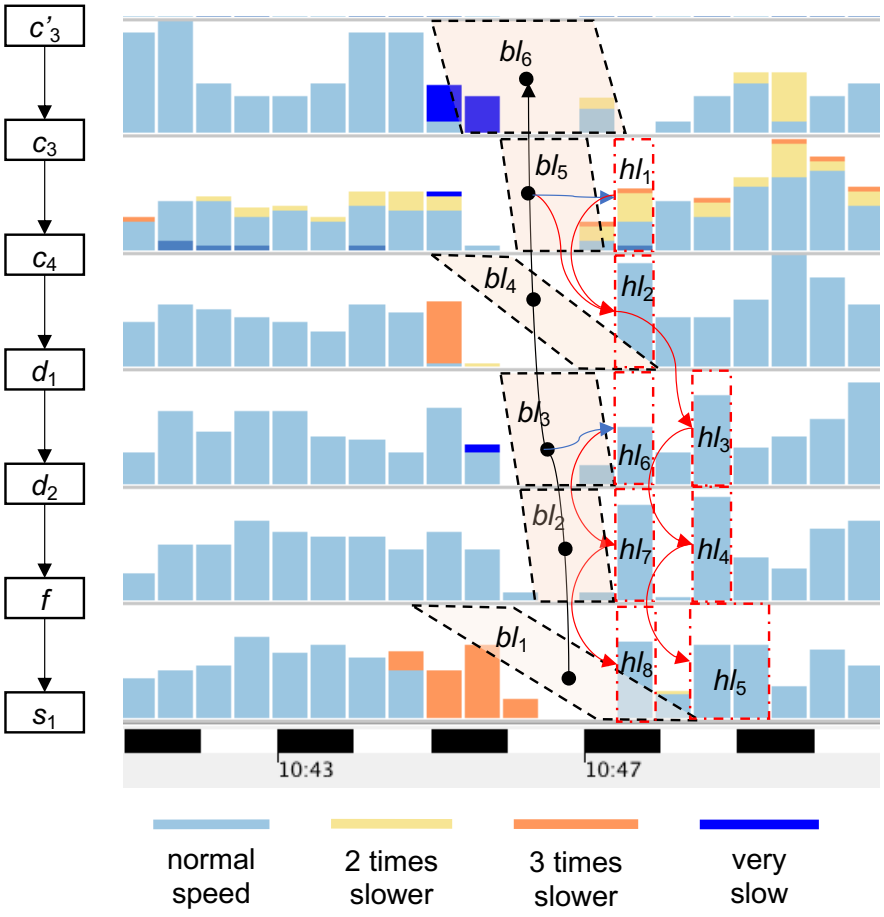


Figure 1.11: Aggregate performance spectrum built from a complete event table.

Conversely, a blockage instance corresponds to a longer waiting time in a queue of the corresponding Q-proclet. As soon as the queue reaches its maximum capacity, all incoming queues become blocked (this behavior is explained in Chapter 4, **RQ-2**). As a result, the segments corresponding to these incoming queues get blockage instances as well. To describe this situation, we say that blockage instances propagate *backward* in the control-flow direction.

Additionally, we consider how blockage instances can cause high load instances. For example, TSUs, accumulated during a blockage, can cause a high load instance when the blockage instance ends because the TSUs are handed over to available queues (conveyors) in a batch. We use such observations to merge propagation chains

into larger ones and identify the origins of undesirable performance scenarios more accurately.

The propagation chain, discovered for the undesirable performance scenario of Section 1.1.2, is shown in Figure 1.11. It is discovered using the following steps.

1. Blockage and high load instances $bl_1 - bl_6$, $hl_1 - hl_8$ were detected.
2. The chain of blockage instances (shown by black arrows) was discovered using the P-proclet (Figure 1.11(left)). Blockage instance bl_1 triggered this chain by propagating from (f, s_1) backward in the control-flow direction toward (c_3', c_3).
3. Two chains of high load instances, consisting of $hl_1, \ldots, hl_5$ and $hl_6, \ldots, hl_8$ (shown by red arrows) were discovered. Instances hl_1 and hl_6 triggered these chains by propagating from (c_3, c_4) and (d_1, d_2) respectively in the control-flow direction toward (f, s_1).
4. However, hl_1 and hl_6 did not appear by their own. Actually, ending of blockage instances bl_5 and bl_3 triggered them. So, we merged these chains accordingly (blue arrows).
5. As a result, one large chain, comprising all the instances, was discovered. It originated in segment (f, s_1) (bl_1).

Next, the performance spectra of the queue and resource spectra of (f, s_1) are to be investigated, to identify what queue or resource caused the blockage. We refer to Chapter 8.3.9 for the detailed description.

Last but not least, we also discuss what is required for doing the same in the real-time setting for *monitoring*. Chapter 8 is based on on [47, 54].

Predictive Performance Monitoring in Chapter 9. To address **RQ-7**, we

1. show how the performance spectra are capable of capturing *dynamics* of systems with shared resources and queues, and why it can be used as the source of both rich performance-related features and information required to derive the values of PPIs,
2. formulate a *regression problem* of predicting a given aggregate Process Performance Indicator (PPI) over the multi-channel performance spectrum (instead of an event table), and
3. propose a method to identify relevant performance-related features in the performance spectrum and extract them for training ML predictive models.

Let us illustrate how we formulate the problem, using the running example of Section 1.1.2. Given a target PPI showing the load on segment (f, s_1) in bags that enter this segment per 30 seconds, we want to predict it within a *prediction horizon* t_{ph} of one minute. This PPI can be derived from the aggregate performance spectrum of segment (f, s_1) directly as the bin height, which shows exactly the number of bags entering the segment in a 30-second interval. We call the performance spectrum required to derive the target PPI a *target spectrum* (shown in red in Figure 1.12).

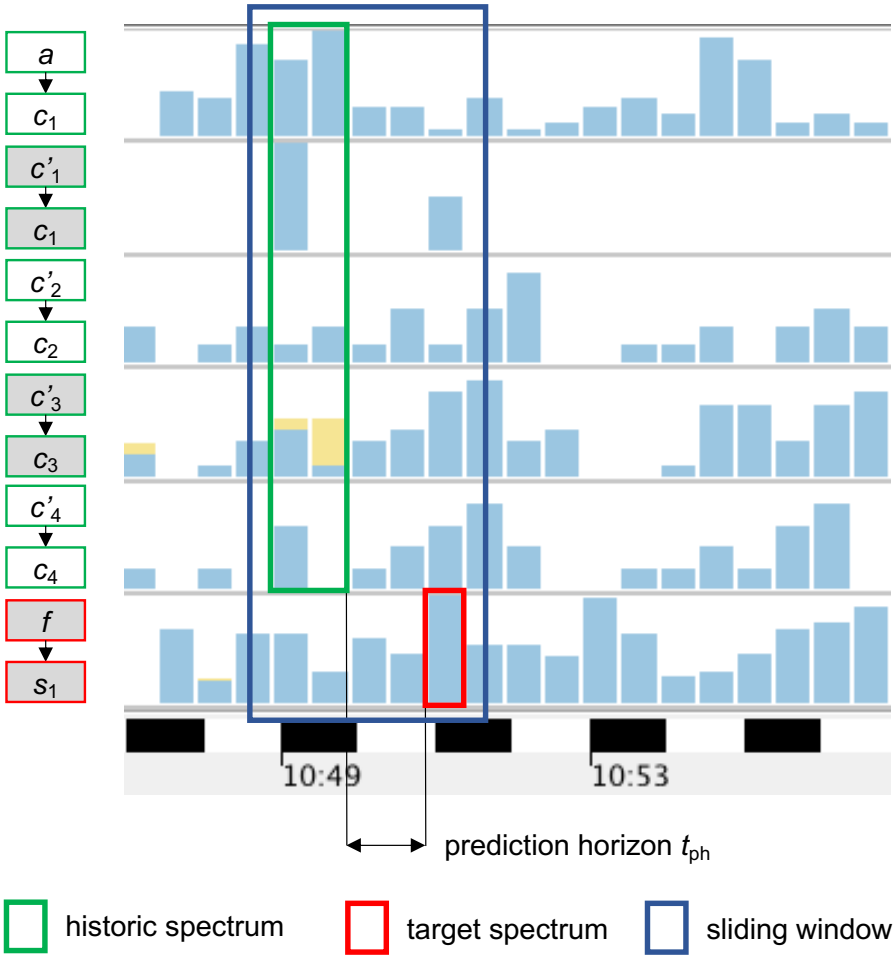


Figure 1.12: Problem of predictive performance monitoring over the performance spectrum.

Then, we introduce a *historic spectrum* (shown in green in Figure 1.12) that is needed to estimate (predict) the target spectrum within t_{ph}. The regression problem is to predict the target spectrum, using the observed historic spectrum.

The historic and target spectrum can be materialized for each required time *now*, using the sliding window technique [55]. The main challenge is to identify features that are relevant for predicting the target spectrum. To do it, we build on observations about the propagation of performance pattern instances (considered in Chapter 8, **RQ-6**) as follows.

1. We show that the propagation of any load can be discovered in the same way as high load propagation is discovered in Chapter 8.
2. We estimate the propagation time (in addition to propagation paths) using the minimum queue waiting and minimum resource service time, known from the Q- and R-proclet parameters.
3. We identify the historic spectrum, using the identified propagation paths and time.
4. We use the sliding window technique to extract features, and the standard ML pipeline to train a regression model.

Thus, we again combine the PQR-system and performance spectrum, as in Chapter 8. Note, we use *multi-channel* performance spectra to capture all required dynamics aspects in the historic spectrum.

In our example, we identify that segments (a, c_1), (c_1', c_1), (c_2', c_2), (c_3', c_3), and (c_4', c_4) can be used in the historic spectrum to estimate the target spectrum within $t_{\text{ph}} = 60$ seconds (i.e., two bins).

This chapter is based on [47].

Conclusion in Chapter 10. Finally, we summarize the main contribution of this thesis for both academia and industry, overview the software tools we implemented, consider the limitations of our approaches, and briefly discuss how some of them can be overcome in future work.

Next, we discuss the contributions of this thesis.

1.5 Contributions

In this section, we summarize both the scientific and practical contributions of this thesis. We organize it along the thesis structure, i.e., we start with the performance spectrum and PQR-system, because they are used for the remaining methods and techniques of the thesis. Then, we discuss the contributions of our methods for conformance checking, descriptive performance analysis, and predictive performance monitoring of systems with shared resources and queues.

1.5.1 Performance Spectrum and Performance Patterns

We proposed the performance spectrum, a novel technique for fine-grained performance description of systems with shared resources and queues from event logs. It is a generic technique that takes an event log to describe transitions between directly following process steps for each case in the log over time. The power of this technique comes from the following key properties.

- The provided performance description is *unbiased* because no model, that would most probably introduce a bias, is required for computing.
- The performance spectrum is capable of describing *non-stationary processes* because the time is represented explicitly and no aggregation is used.
- The performance spectrum reveals case interactions on shared resources because it describes the performance of *all cases together*, thereby allowing for analysis of systems with shared resources and queues.
- Finally, it can be tailored for the analyst's needs by defining a performance classifier over any information available from the event log (e.g., from event attributes).

Additionally, we proposed the *aggregate* performance spectrum, useful if quantification of the performance description is required. We also proposed how to capture more performance aspects by combining performance spectra that use different performance classifiers and aggregation types into a three-dimensional structure.

The performance spectrum can be used as:

1. a data structure capturing process dynamics, and
2. a visual analytics technique allowing for its analysis by stakeholders.

To facilitate performance spectrum-based analysis, we suggested a taxonomy of *performance patterns* observed in the spectra of processes of various domains. It helps detect, isolate, and describe various performance phenomena in terms of distinct pattern instances.

Our evaluation had two main goals. First, we wanted to validate whether the information in the performance spectrum visualization containing performance pattern instances, can be

- perceived by various analysts, and
- recognized unambiguously.

Second, we wanted to prove that

- the performance spectrum-based analysis can be used for solving complex performance analysis problems of large processes, and
- the multi-channel performance spectra allow us deeper insights into process behaviors.

For the former, we conducted an empirical study [56] aimed to prove that the findings reported in [45] can be reproduced by untrained analysts, using [45] as a guideline, and the Performance Spectrum Miner (PSM) [46] as a software tool for working with performance spectra. We asked six participants the following questions over the same 12 event logs that were considered in [45] (except the BHS log that could not be shared).

1. **ExQ-1.** Whether the performance patterns, provided in the taxonomy [45] (Figure 3.9), can be identified in the performance spectra of various processes?
2. **ExQ-2.** Whether the analyst can identify the same performance patterns as the authors of [45]?

As a result, the participants could do both, i.e., they could identify patterns, and these patterns were mostly the same as the authors' ones. The main problem was the absence of concrete time information for the examples in [45]. It made the analysis more time-consuming.

For the latter, we conducted a study at Vanderlande. For that, Vanderlande's domain experts shared a BHS event dataset and formulated a performance analysis problem as follows.

- In a major European airport, a BHS experienced severe performance problems. How detect these problems and find their root causes, using the system material flow diagram, and a dataset of recorded event data?

We were informed that this was a difficult problem, whose analysis was already done but took a long time — around three man-months. However, no analysis results were shared beforehand.

We computed "regular" and multi-channel performance spectra, using the given dataset. We used various event attributes to define different performance classifiers for the multi-channel performance spectrum. As a result, we successfully detected and explained the incident scenario, and explained its root causes. This scenario showed how the bag interactions on the system resources, together with equipment problems, eventually caused the complete system halt. Validation with Vanderlande's engineers showed that our results were similar to theirs but obtained significantly faster — in three man-weeks instead of three man-months.

Our evaluation proved that the performance spectrum is a powerful technique for the performance description and analysis of processes and systems, including systems with shared resources and queues. Additionally, the performance patterns allow an expressive high-level performance description, easily understandable even by untrained analysts.

1.5.2 Modeling Systems with Shared Resources and Queues

For modeling systems with shared resources and queues, we proposed the PQR-system, a dedicated synchronous proclet system modeling the process, queue, and resource dimensions. Its purpose is to enable model-based process mining techniques for systems with shared resources and queues. Our contribution here is twofold:

1. the model (i.e., PQR-system) itself, and
2. the way we approached its design.

The latter showed how the problem of modeling systems with shared resources and queues (and perhaps many other types of systems/processes) can be approached. We showed how

- the actual analysis questions were formulated first,
- the state-of-the-art techniques were considered next, using process mining techniques (performance spectrum) to determine whether their assumptions held for the studied systems, and
- concepts to be reused in our own design were identified in the existing approaches.

Following this approach, we used a queueing network-based state-of-the-art approach [18, 20] for modeling MHS performance and validated its assumptions using the performance spectrum and domain knowledge. Although we concluded they did not hold for the systems we studied, we identified the key elements capable of explaining MHS behaviors conceptually:

- finite-capacity FIFO queues with pre-defined time characteristics,
- resources (servers) with pre-defined time characteristics,
- and a routing function that defines the handover of jobs (TSUs, cases) among them.

Further, we represented these concepts with a synchronous proclet system [48]. That is, we mapped the queues, resources, and routing function to dedicated queue, resource, and process proclets respectively, using channels to synchronize their interactions. We used a small subset of the CPN syntax to model these proclets. We also defined the PQR-system replay semantics that checks whether a given event table can be generated by the modeled system. The resulting model enables process model-based techniques for the analysis of systems with shared resources and queues.

Last but not least, we designed and implemented a BHS simulation model and the corresponding PQR-system to demonstrate how a concrete system can be modeled with the PQR-system, and what kind of event data it generates[3]. It allows for materializing of a sample BHS for any researcher that needs it for doing experiments in the fully controllable environment of this simulation model.

Next, consider how the PQR-system replay semantics allows addressing conformance checking of systems with shared resources and queues.

1.5.3 Generalized Conformance Checking

We addressed the problem of *generalized* conformance checking that combines conformance checking, model repair, and log repair under the common roof [44]. We

[3]The source code and documentation are available on `https://github.com/processmining-in-logistics/psm/tree/pqr`.

chose it instead of "classical" conformance checking because in real-world settings we cannot consider either an MHS model (PQR-system) or event data as fully trusted.

For conformance checking, we

- discussed conformance checking use cases for the MHS domain, and
- showed how existing techniques (sometimes after minor extensions) can be used for PQR-system-based conformance checking and model repair.

Thus, we showed that conformance checking of the P-proclet allows for the detection of sensor malfunctioning and manual interventions in material handling processes, as well as the detection of incomplete logging. Conformance checking of Q- and R-proclets allows for the detection of some control-flow outliers that cannot be detected through P-proclet conformance checking, to be used for the same purposes. Additionally, it allows for concept drift detection for Q- and R-proclets parameters, and estimation of log repair accuracy (if log repair was applied). Finally, synchronization conformance checking allows for revealing issues in data that violate correlation constraints defined by the PQR-system.

For log repair, which is a part of generalized conformance checking, we proposed a novel method for inferring unobserved events with timestamp information using the PQR-system. It allows for obtaining complete event tables from usually incomplete ones recorded by nowadays MHSs. Such complete event tables are correct, i.e., they can be replayed over a PQR-system according to its replay semantics. The evaluation of our implementation using synthetic and real-life data showed that

- the error of the estimated timestamps and derived performance characteristics (load) was less than $< 5\%$ under regular performance, and
- real-life dynamics (i.e., load peaks) were correctly restored after irregular behavior (e.g., after completion of blockage or high load instances).

1.5.4 Multi-Dimensional Performance Analysis

The contributions of our method for multi-dimensional performance analysis of systems with shared resources and queues are threefold:

1. the way to relate performance spectra to the PQR-system,
2. model enhancement via combining the PQR-system with performance spectra, and
3. the method itself.

The stand-alone performance spectrum, as we introduced it first in Chapter 3, is a powerful technique. However, alone it (1) has a lack of structure, and (2) its interpretation is possible only using domain knowledge in the analyst's mind. Both factors limit its application in practice. Relating the performance spectrum to the PQR-system

adds structure to the performance spectrum: its segments now can be sorted according to valid sequences of process steps, known from the P-proclet. Additionally,

- the segments of the *process* (control-flow) performance spectrum get interpreted as transitions between the process step in the P-proclet,
- queue performance spectra show actual queue waiting times for cases, and
- resource performance spectra show actual resource service and idle time during and after case handling respectively.

Additionally, temporal parameters of the Q- and R-proclets can be used for defining performance classifiers with accurate thresholds for performance classes.

Interestingly, there is another view on it. In process mining, a process model can be *enhanced* by projecting, for example, performance information onto its elements [1]. However, it must not be understood only literally, i.e., as drawing some information on top of the model visualization. By relating performance spectrum segments to the places and transitions of the PQR-system, we map the performance information to the model elements, as our tool [54] demonstrates. In essence, we effectively enhance the model with a performance spectrum. Mentally, it can be seen as the stripes of performance spectrum segments attached to the model places in a three-dimensional space.

Last but not least, our method for performance analysis goes beyond outlier or bottleneck detection. It allows for identifying the origins of performance problems, pretty much without any domain knowledge outside one captured by the PQR-system. Further, it switches the view on the performance description from one in the process dimension to others in the queue and resource dimension to identify which root causes have the phenomena in the performance spectra of these dimensions.

We consider our method as a foundation for methods for the analysis of processes in other domains. For example, assuming an "extended" PQR-system supporting pools of resources (assignment groups) and non-FIFO ordering, the performance spectra of additional dimensions can be computed and analyzed for identifying root causes of performance issues, using more pattern types from our performance patterns taxonomy in Chapter 3.

For evaluation, we developed a proof-of-concept implementation as an open-source tool[4] [54], while a software solution based on this approach for the MHS domain was implemented internally by Vanderlande and evaluated by their domain experts for several systems and use cases. The evaluation outcome is twofold. First, the approach revealed findings that could not be obtained using other tools. Second, it allowed faster results than classical tools for MHS performance analysis. Additionally, the integrated model and performance spectrum allowed for a shallow learning curve for experts previously unfamiliar with the MHS and performance spectrum.

[4]The source code and further documentation are available on `https://github.com/processmining-in-logistics/psm/tree/pqr`

1.5.5 Predictive Performance Monitoring

Finally, in Chapter 9 we studied the problem of forecasting the performance of systems with shared resources and queues.

We showed that the multi-channel performance spectrum, derived from the event log of a process, allows for modeling a variety of process performance features over time, capturing also inter-case dependencies in systems with shared resources and queues. Specifically, we showed how its bins, defined over the three basic dimensions of process step, performance measure, and time interval, capture features and allow for the formulation of a large class of performance prediction problems as a regression problem. This vision can be used by other approaches for formulating PPM problems that must consider case interactions on shared queues and resources.

We proposed a method for obtaining a feature set, required for solving this regression problem as an ML task. It includes an approach for feature selection, that can be to a large extent "formalized", using domain knowledge available from the PQR-system. That is, the features are not obtained by trial and error, or by using some domain knowledge in the data scientist's mind, but are identified by computing the "locations" of origins of future performance in the performance spectrum (in the space of time and segments) using the PQR-system.

We provided examples of real-life problem instances for a BHS and evaluated our method by training sound models for solving these problem instances on the real event log of a major European airport BHS. We demonstrated the feasibility of our approach and compared it to the current state-of-the-art approaches, e.g., [35]. The experiments showed that our performance spectrum-based linear model outperforms the more complex non-linear model of [35]-based approach, and the performance spectrum-based non-linear model outperforms the naive baseline for the problem instance where [35] was not applicable due to the optionality of the target process step.

The remainder of the thesis is structured as follows. We introduce basic definitions and notations in chapter 2. We propose the performance spectrum in Chapter 3. We explore the modeling of systems with shared resources and queues with queueing theory in Chapter 4, review related work in Chapter 5, and model the PQR-system in Chapter 6. We address the problems of generalized conformance checking in Chapter 7. We suggest our methods for descriptive performance analysis and PPM in Chapter 8 and Chapter 9 respectively. We conclude the thesis in Chapter 10.

Chapter 2

Preliminaries

This chapter introduces basic concepts used throughout this thesis. In Section 2.1, we recall sets, multisets, sequences, relations, functions, and partial orders. In Section 2.2, we review the key concepts we use for process modeling, such as Petri nets. Then in Section 2.3, we recall colored Petri nets, including their version with support of time. We consider both syntax and semantics for these types of nets, described through process runs represented through occurrence nets. Finally, as in process mining, the behavior of processes and systems is typically captured through events they generate, we define concepts needed for describing event-based data, such as events with attributes, event traces, event tables, and event logs, in Section 2.4.

2.1 Notations - Set, Multiset, Relation, Function, Sequence, Graph, and Partial Order

We use the following notations on multisets, sequences, functions, graphs, and relations.

- We denote *sets* by capital letters X, Y, etc., and write $\{x_1, x_2, ...\} = X$ for the elements of a set. A set can be infinite.
- We write $R \subseteq X \times Y$ for a *binary relation* R on sets X, Y, and $(x, y) \in R$ when x is R-related to y. For some relations, we write xRy for $(x, y) \in R$. We write *relation on* X for a subset of $X \times X$.
- We write R^+ for the *transitive closure* of binary relation R on set X, that is the smallest relation on set X that contains R and is *transitive*, i.e., whenever R^+ relates a to b and b to c, then R^+ also relates a to c. We write $R^\otimes$ for a *transitive reflexive closure* of binary relation R on set X. that is the union of the transitive closure R^+ and the relation $\{(x, x) \mid x \in X\}$. We write R^- for a *transitive reduction*

of relation R, that is a minimal relation on X which has the same transitive closure as R.

- A *total function* f from X to Y, denoted as $f : X \to Y$, is a relation that maps *every* element from X to an element of Y.
- A *partial function* f from X to Y, denoted as $f : X \not\to Y$, is a relation that maps every element from a *subset* $X' \subseteq X$ to an element of Y. We write $f(x) = \perp$ if f is undefined for x, i.e., when $x \in X \setminus X'$.
- We write $f\big|_{X'}$ for the *restriction* of function $f : X \to Y$ to a smaller domain $X' \subseteq X$, i.e., for a new function $f\big|_{X'} : X' \to Y$ such that $f\big|_{X'}(x) = f(x)$ for $x \in X'$.
- A *multiset* (or *bag*) over finite set X is a total function $m : X \to \mathbb{N}$ that maps every element from X to a number of its occurrences in the multiset. We write, for example, $m = [a, b^3]$ for a multiset m over the set $\{a, b, c\}$ where $m(a) = 1$, $m(b) = 3$ and $m(c) = 0$. We write $[]$ for the empty multiset. We denote the set of all multisets over X as $\mathbb{B}(X)$.
- We denote *sequences* by small Greek letters σ, θ etc., and write $\langle a_1, a_2, \ldots, a_n \rangle = \sigma$ for the ordered elements of a sequence, and $|\sigma|$ for the sequence length,
- We write $G = (V, E)$ for *graph* G consisting of a non-empty set of vertices V, and a set of edges E. A graph is called *directed* if E consists of ordered vertex pairs, and *undirected* otherwise. A graph is *connected* if there is a path from any vertex $v \in V$ to any other $v' \in V$.

A *partial order* is a binary relation $<$ on set Y if for all y, y', and y'' in Y it is:

1. *irreflexive*, i.e., $\neg(y < y)$,
2. *antisymmetric*, i.e., $y < y' \implies \neg(y' < y)$,
3. *transitive*, i.e., $y < y' \wedge y' < y'' \implies y < y''$.

We write $(Y, <)$ for partial order $<$ over the set Y. For a *labeled partial order* on a set Y for the set Σ of labels, we write $(Y, <, \ell)$ where $(Y, <)$ is a partial order, and function $\ell : Y \to \Sigma$ maps the elements of Y to the labels of Σ. We write $e_1 < e_2$ if event e_1 *precedes* event e_2, and we write $e_1 \lessdot e_2$ iff e_1 *directly* precedes e_2, i.e., when $e_1 < e_2$ and there is no other event e_3 with $e_1 < e_3 < e_2$.

2.2 Process Model and Process Runs

Process models are typically used for describing either *intended* or *observed* behavior of processes. A process model usually documents and represents process activities and their causal dependencies using a graph-based notation. In this thesis, we extensively use the Petri net notation as it is widely used in process modeling and serves our needs well. In this section, we recall Petri nets, explain their semantics in terms of process runs and introduce workflow nets, a sub-class of Petri nets widely used for

modeling business processes. Since we only explain the basic notions, we refer to [42] for more details.

2.2.1 Labeled Petri Nets

Here we formally define the syntax of labeled Petri nets, i.e., Petri nets where nodes have labels, and their semantics, i.e., the state of Petri nets, called a *marking*, and how a state of a labeled Petri net changes from one marking to another. We assume that these labels, i.e., the names of process steps to be projected on the model, are given.

A labeled Petri net is a *directed graph* with nodes of two types, called *transitions* and *places*. Each transition node can only have adjacent nodes of the type place, and each place node can only have adjacent nodes of the type transition. Each node of a labeled Petri net has a *label* from the set Σ of labels. A *transition label* can be considered as the *observable action*. A *place label* can be considered as a name of a state between actions, represented by adjacent transitions of the place.

In this thesis, we use labels that clearly show the regular structure of the models of material handling processes For that, we introduce a special naming convention for transition and place labels in Section 6.3.1. Later, it is used for simplifying definitions related to the process models of this thesis.

Sometimes, it is necessary to express that a particular transition is not observable. For this, we reserve the label $\tau \in \Sigma$. A transition whose label is τ is *unobservable*. Such transitions are often referred to as *silent* or *invisible* in literature [1].

The formal definition of a labeled Petri net reads as follows.

Definition 2.1 (Labeled Petri net)**.** *Let Σ be a finite set of labels. A* labeled Petri net *$N = (P, T, F, \ell)$ over Σ is a tuple in which:*

- *P is a finite set of places,*
- *T is a finite set of transitions such that $P \cap T = \emptyset$,*
- *$F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs, called the* flow relation*,*
- *$\ell : T \cup P \to \Sigma$ is a labeling function.*

For every transition $t \in T$, ${}^\bullet t = \{p \in P | (p, t) \in F\}$ is the set of *pre-places* of t and $t^\bullet = \{p \in P | (t, p) \in F\}$ is the set of *post-places* of t, *pre-* and *post-transitions* of a place are defined correspondingly. We write $N_1 \cap N_2$, $N_1 \cup N_2$, and $N_1 \subseteq N_2$ for an intersection, union, and subset of nets respectively, which is defined element-wise on the sets of nodes and arcs of N_1 and N_2, and we write $\emptyset$ for the empty net.

A *state* of a labeled Petri net is defined by its *marking* $m \in \mathbb{B}(P)$ that assigns a number $m(p)$ of tokens to each place $p \in P$. Given the significance of a model's initial state, the initial marking is often included in the net structure as follows.

Definition 2.2 (Labeled, Marked Petri net)**.** *A* labeled, marked Petri net *is a tuple* $(N, m_0) = (P, T, F, \ell, m_0)$ *in which:*

- N *is a labeled Petri net,*
- m_0 *is a marking of* N*.*

A state of a labeled, marked Petri net changes when a transition *occurs*. A transition $t \in T$ is *enabled*, i.e., can occur, at marking m of N, when each pre-place of transition t contains at least one token, i.e., iff $\forall p \in {}^\bullet t \; m(p) > 0$.

When transition t occurs, it consumes one token from each of its pre-places and produces one token on each of its post-places, thereby generating a new marking

$$m'(p) = \begin{cases} m(p) - 1 & \text{if } p \in {}^\bullet t \setminus t^\bullet \\ m(p) + 1 & \text{if } p \in t^\bullet \setminus {}^\bullet t \\ m(p) & \text{otherwise.} \end{cases}$$

We write $m \xrightarrow{t} m'$ for the occurrence of transition t that changes the marking from m to m', and call it a *step*. Let us consider two markings $m_1, m_n \in \mathbb{B}(P)$. If for those markings there exists a sequence of transitions $\sigma = \langle t_1, t_2, \ldots, t_{n-1} \rangle$ such that $m_1 \xrightarrow{t_1} m_2 \xrightarrow{t_2} m_3 \ldots \xrightarrow{t_{n-1}} m_n$, we write $m_1 \xrightarrow{\sigma} m_n$ and call this sequence an *occurrence sequence* of N. In this case, marking m_n is *reachable* from marking m_1. The marking m'' is *unreachable* from marking m_1 if there is no sequence of transitions starting at marking m_1 that reaches marking m''.

A toy example of an airport BHS is shown in Figure 2.1(a). It has two check-in counters $a1, a2$, a merge unit b where the flows from check-in counters $a1, a2$ merge into a single flow that goes to lateral (exit) c, where bags are off-loaded to aircraft.

The corresponding labeled, marked Petri net that models the BHS is shown in Figure 2.1(b). In Petri nets, we model process steps with transitions whose labels describe the corresponding process steps. For example, when a bag *front* side passes location $a1$, we consider that as the start of checking in at counter $a1$, and label the corresponding transitions as $a1_s$. Respectively, when a bag *back* side finally leaves the counter, we consider that as the completion of this process step, and label the corresponding transitions as $a1_c$. We draw transition identifiers *outside* the transition rectangles, and we draw labels *inside* the transition rectangles. For example, the top left transition in Figure 2.1(b) has identifier $t1$ and label $a1_s$. A place between *start* and *complete* transitions of the same process step defines that a bag is being processed, and the place label is the name of this process step. For example, the place with label $a1$ defines that a bag is being processed (at process step $a1$). A place between transitions of different process steps defines that a bag is in transit from one process step to the next one. For example, the place with label $a1{:}b'$ defines that a bag is in transit from process step $a1$ to process step b'. As for transitions, we draw place identifiers and labels outside and inside the nodes respectively.

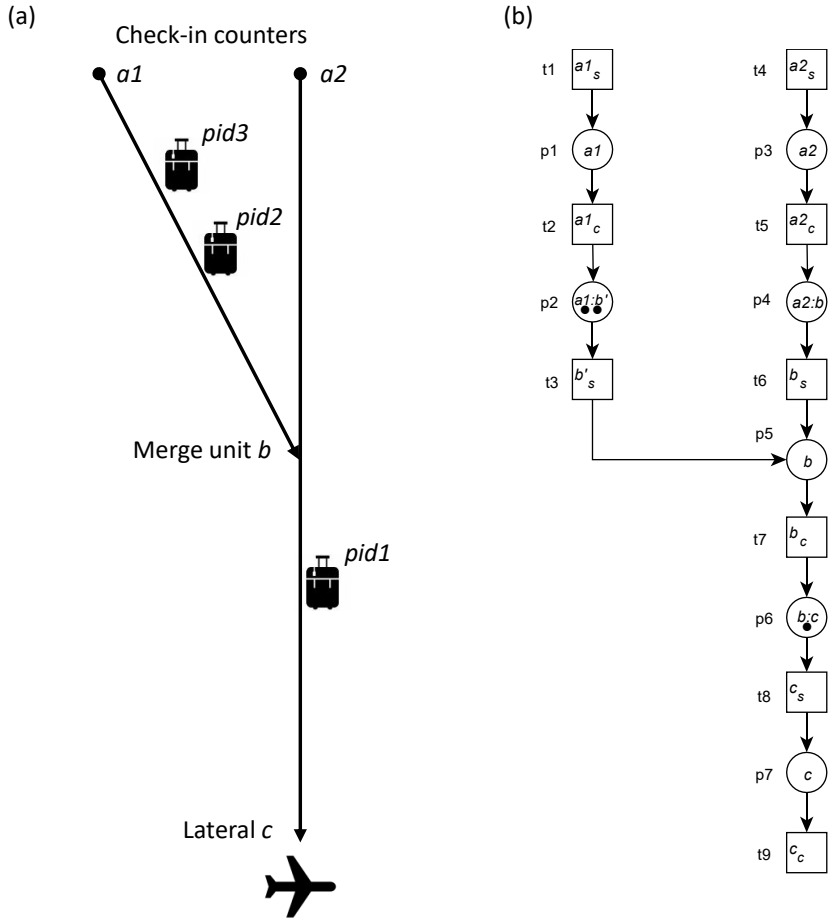


Figure 2.1: MFD of a BHS (a) and an example of the corresponding labeled, marked Petri net (b).

The diagram in Figure 2.1(a) also shows the current state of the system, where two bags are traveling from location $a1$ to b, and one bag is traveling from location b to c. The net shown in Figure 2.1(b) models bags, transported through the BHS, as tokens. The state of the system is described by a distribution of tokens on the places, i.e., by its marking. In Figure 2.1(b), the current state is described by the marking $[a1{:}b'^2, b{:}c^1]$, which is visualized as two tokens on place $p2$ with label $a1{:}b'$ and one token on place $p6$ with label $b{:}c$.

2.2.2 The Semantics of Petri Nets

Processes often have parallelism, for example, to speeding-up their steps or due to distributed nature of systems. In literature [42, 57, 58, 59], among others, two semantics for analysis of such processes have been considered: (1) *interleaving semantics* and (2) *true concurrency semantics*. Interleaving semantics equates the concurrent execution of actions with their execution in an arbitrary but *global* order. The assumption about the global order can be undesirable. For example, in Figure 2.1(b) bag $pid1$ is located on conveyor $b:c$, while bags $pid2$ and $pid3$ are located on conveyor $a1:b$, i.e., they are in different parts of the system. Any ordering of $pid2$ and $pid3$ to $pid1$, assumed by interleaving semantics, is inadequate unless they reach a common part where they synchronize. So such an interpretation has a significant drawback, as it provides an inaccurate description of what can happen. By contrast, true concurrency semantics allows multiple actions to be executed *simultaneously*, thereby allowing, for example, *unordered* execution of process steps b'_s for $pid2$ and c_s for $pid1$. As a result, systems with concurrency can be modeled more adequately.

Petri net theory has a long tradition of studying true concurrency by way of *partial order semantics* [57], which we recap in this section. Most often, a partial-order semantics of Petri nets is based on so-called *distributed runs* [42] (that we refer to further in this thesis as *runs*). Runs are based on a special relatively simple class of nets, called in literature either *causal nets* or *occurrence nets*. We use the latter term in this thesis.

Definition 2.3 (Occurrence net)**.** *An* occurrence net $\pi = (B, E, G)$ *is a net, as defined in Definition 2.1, but without labeling where:*

1. *each place* $b \in B$ *is called a* condition*,*
2. *each transition* $e \in E$ *is called an* event*,*
3. *the transitive closure* G^+ *is irreflexive,*
4. π *is finitely preceded, i.e., the set* $past(x) = \{y \mid (y,x) \in G^+\}$ *is finite for each* $x \in B \cup E$*,*
5. *each condition* $b \in B$ *has at most one pre-event and at most one post-event, i.e.,* $|{}^\bullet b| \leq 1$ *and* $|b^\bullet| \leq 1$*.*

We write $min(\pi) \subseteq B$ to refer to the conditions that have the empty preset.

Definition 2.4 (Run)**.** *A* run *of a labeled marked Petri net* $N = (P, T, F, \ell, m_0)$ *is a labeled occurrence net* $\pi = (B, E, G, \lambda)$ *where the labeling function* λ *satisfies the following properties:*

1. $\lambda(B) \subseteq P$ *and* $\lambda(E) \subseteq T$*, i.e., the labeling function* λ *preserves the nature of nodes,*
2. *let* $t = \lambda(e)$*; for each* $b \in {}^\bullet e$ *exists exactly one* $p \in {}^\bullet t$ *with* $p = \lambda(b)$*, and for each* $p \in {}^\bullet t$ *exists exactly one* $b \in {}^\bullet e$ *with* $\lambda(b) = p$*, and similarly for each* $b \in e^\bullet$ *exists exactly one* $p \in t^\bullet$ *with* $p = \lambda(b)$*, and for each* $p \in t^\bullet$ *exists exactly one* $b \in e^\bullet$ *with* $\lambda(b) = p$*, i.e.,* λ *preserves the environments of events,*

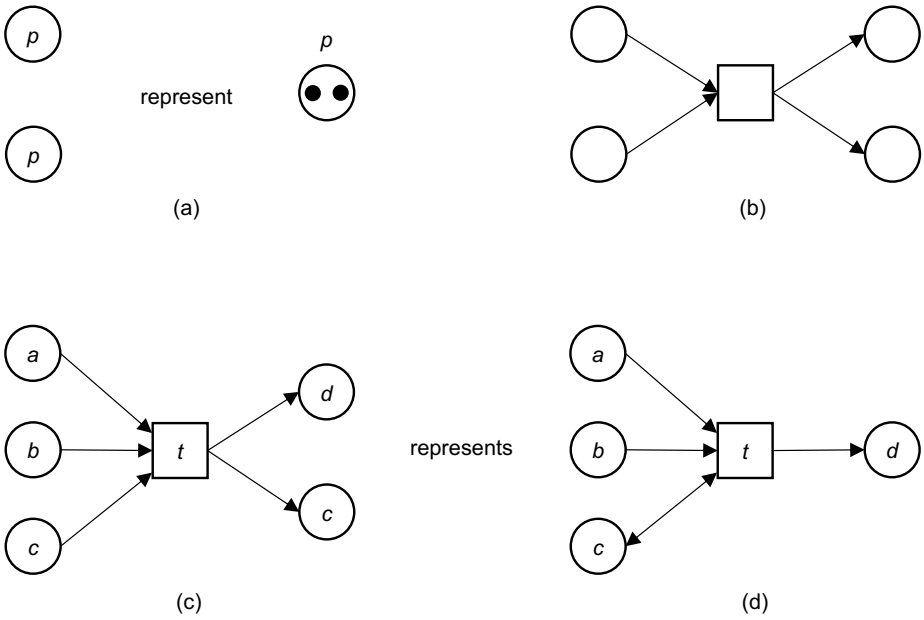


Figure 2.2: Occurrence nets.

3. *π "starts" at initial marking m_0 as follows:*

   - *for each $b \in min(\pi)$ there exists exactly one $p \in m_0$ such that $p = \lambda(b)$, and*
   - *for each $p \in m_0$ the number of corresponding conditions in $min(\pi)$ is equal to the number of tokens on p in initial marking m_0, i.e., $m_0(p) = |\{b \mid b \in min(\pi), \forall b_1, b_2 \in min(\pi), b_1 \neq b_2, \lambda(b_1) = \lambda(b_2) = p\}|$.*

To show what a run of a net is, we first provide examples of the building blocks of a run and then provide a run of a net with concurrency. In an occurrence net of a run, each occurrence of a token is represented as an individual, labeled place. For example, if during a run two tokens occurred on a place, such a place is represented in the corresponding occurrence net as two conditions (Figure 2.2(a)). However, a condition alone cannot be seen as a building block because a token occurrence in a run is always related to a transition firing (besides conditions corresponding to the initial marking). So, a fact of a transition firing and its effect on the tokens involved is described as a net of a particular structure, shown in Figure 2.2(b), called a *transition occurrence*. An occurrence of transition t of the model of Figure 2.2(d) is shown in Figure 2.2(c). Notice that place c, from which a token is consumed and also produced by transition t, is represented through two condition places in the occurrence net. So,

a run of a net is composed of transition occurrences, connected into an occurrence net.

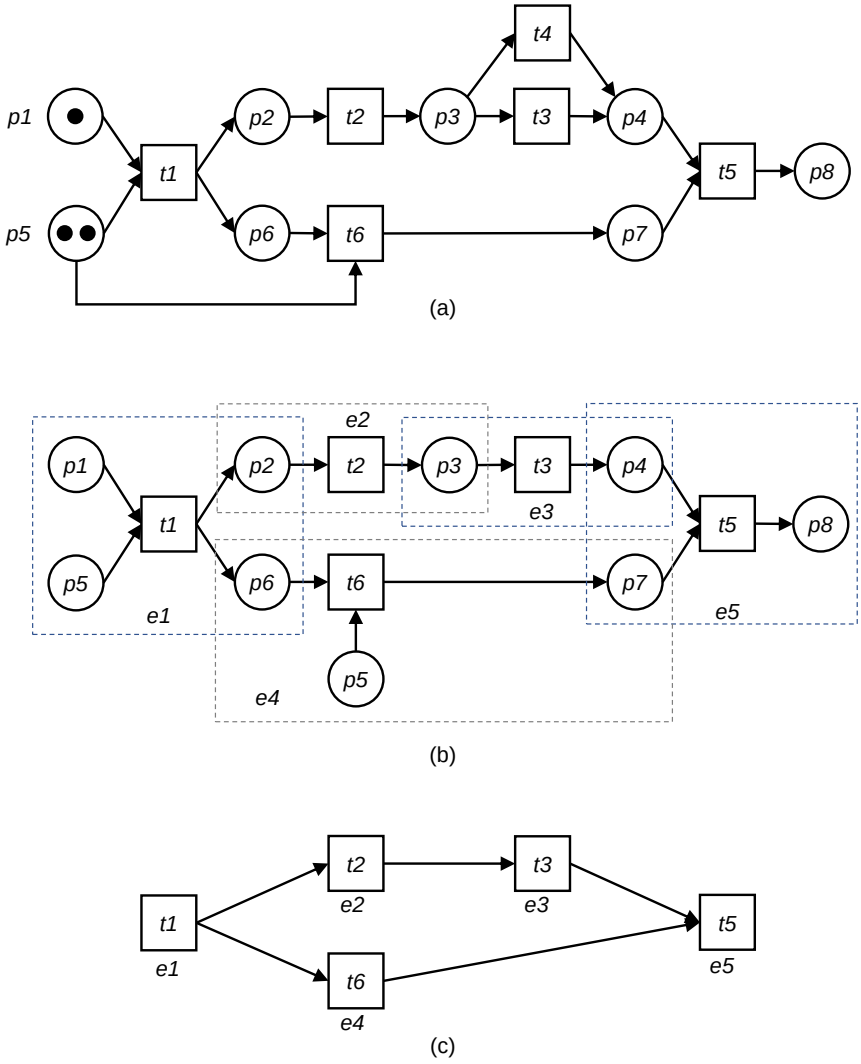


Figure 2.3: Marked Petri net (a), its run (b), and its labeled partial order (c).

Figure 2.3(b) shows a run $\pi = (B, E, G, \lambda)$ of a net with concurrency and a choice, shown in Figure 2.3(a), where the transition occurrences are shown within dashed rectangles. The run shows the ordering of events. For clarity, we extract the labeled

partial order π'_E of events of π: $\pi'_E = (E, <_E, \lambda\big|_E)$, with $<_E = (G^+ \cap (E \times E))^-$ shown in Figure 2.3(c). We can clearly see that event $e2$ is directly followed by event $e3$, while the order among $e2$ and $e4$, or $e3$ and $e4$ is not defined. In a run, concurrent events represent transitions that can occur concurrently, e.g., events $e2$ and $e4$ in Figure 2.3(b).

2.2.3 Workflow Nets

For modeling business processes, a sub-class of Petri nets known as *WorkFlow nets* (WF-nets) is often used [1].

Definition 2.5 (Workflow net)**.** *Let $N = (P, T, F, \ell)$ be a labeled Petri net. N is a* workflow net *if*

1. *P contains a unique input place $i \in P$, also called a* source place*, such that ${}^\bullet i = \emptyset$,*
2. *P contains a unique output place $o \in P$, also called a* sink place*, such that $o^\bullet = \emptyset$,*
3. *every node in $P \cup T$ is on a path from i to o (along the arcs of F).*

The transition bounded labeled Petri net $N = (P, T, F, \ell)$, shown in Figure 2.1, is not a WF-net because it contains neither place $p' \in P$ such that ${}^\bullet p' = \emptyset$ nor sink place p'' such that $p''^\bullet = \emptyset$. To illustrate how it can be transformed into a WF-net N_{WF}, we do the following:

1. we add places i and o to P, i.e., $P_{WF} = P \cup \{i, o\}$ to have the unique source and sink places, and
2. extend relation F as $F_{WF} = F \cup \{(i, a1_s), (i, a2_s), (c_c, o)\}$ to connect the current non-unique input and output transitions with these places.

As a result, each node of this net becomes on a path from unique source i to unique sink o, i.e., it becomes a workflow net. The corresponding example is shown in Figure 2.4. The formal definition of the transformation of a transition bounded labeled Petri net into a WF-net reads as follows.

Definition 2.6 (Transformation of a transition bounded labeled Petri net into a WF-net)**.** *Let $N = (P, T, F, \ell)$ be a transition bounded labeled Petri net, i.e., $\forall p \in P \ {}^\bullet p \neq \emptyset$ and $p^\bullet \neq \emptyset$. Let $i, o \notin P$ be the source and sink places, and let $P_{WF} = P \cup \{i, o\}$ be a set of places of N joined with the sink and source places. Let $F_i = \{(i, t) \mid t \in T, {}^\bullet t = \emptyset\}$ be arcs between the source place and transitions of N without preset. Let $F_o = \{(t, o) \mid t \in T, t^\bullet = \emptyset\}$ be arcs between the transitions of N without postset and the sink. The WF-net $N_{WF} = (P_{WF}, T, F \cup F_i \cup F_o, \ell)$ is the corresponding WF-net of N.*

Not every WF-net represents a correct process, so a *sound* WF-net [1] is desirable. A sound WF-net is characterized by the following properties.

- *Safeness.* In any reachable marking, there is no place holding multiple tokens at the same time.

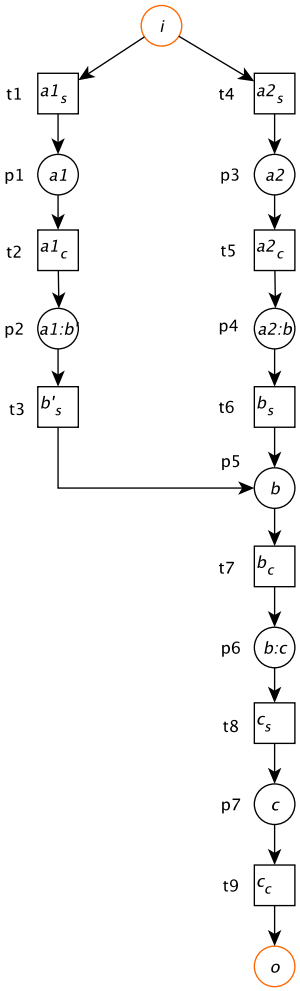


Figure 2.4: Marked, labeled Petri net of Figure 2.1 transformed into a workflow net.

- *Proper completion.* Any marking that assigns a token on sink place o is $[o]$, i.e., when a token reaches the sink, there are no other tokens "in progress" on other places.
- *Option to complete.* The marking $[o]$ (corresponding to the proper completion) is reachable from any marking of the net.
- *Absence of dead parts.* For each transition $t \in T$ there is an occurrence sequence that enables it.

Checking for soundness allows for avoiding coarse design errors that can drastically impede the quality of process analysis based on such problematic nets.

2.3 Colored Petri Nets

In this section, we recap the concepts of Colored Petri Nets (CPNs), a language for the modeling and validating of various complex systems, including concurrent and distributed ones. In this thesis, we use CPNs to model system aspects that are difficult or even impossible to model using Petri nets with black tokens, e.g., element orderings in queues and time-related system characteristics. In the following, we provide a running example and explain the key elements of CPN syntax and semantics required for describing FIFO queues and time. For additional information on CPNs, we refer to [49].

Running Example. Because we use CPNs for modeling FIFO queues with time parameters in Chapter 6, we use such a queue as a running example of this section (Figure 2.5). A FIFO queue can be implemented as a list data structure. An element is added to the queue (enqueued) by appending it to the tail and is taken out of the queue (dequeued) by removing the head element of the queue. The queue has a fixed non-zero *capacity* and the *minimum waiting time* $t_{\text{wQ}} \in \mathbb{T}$, i.e., each enqueued element waits for at least t_{wQ} before dequeuing. Additionally, queue elements must be identifiable. Given such a queue, we want to model it as a CPN.

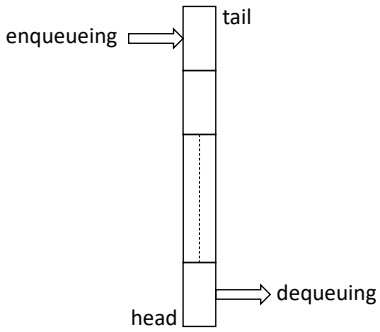


Figure 2.5: FIFO queue.

2.3.1 Colored Petri Nets

In the following, we show how CPNs extend labeled Petri nets by adding types, variables, inscriptions, etc., to the net structure, and explain their semantics.

Places, Transitions, and Arcs. We assume the set *Types* of types, set *Var* of variables, and set *Exp* of expressions explained in detail below. Similarly to labeled Petri nets (see Definition 2.1), the CPN structure $N = (P, T, F, \ell, Var, colSet, m_0, arcExp)$ consists of disjoint finite sets of places P and transitions T, directed arcs F connecting places with transitions and transitions with places, i.e., $F \subseteq (P \times T) \cup (T \times P)$, and an optional labeling function $\ell : T \cup P \to \Sigma$. Although such a labeling function is not typically part of a CPN, we include it in the CPN structure for reasons provided in Section 2.2.1, and we use it extensively later, especially in Chapter 6. On top of this skeleton, the CPN syntax adds variable declarations *Var* and *inscriptions*:

- for *color sets*: function $colSet : P \cup Var \to Types$ assigns for color sets (i.e., types) to values a place or variable can hold,
- for *initial marking*: function $m_0 : P \to \mathbb{B}(Values)$ assigns values that places hold initially,
- for *arc expressions*: function $arcExp : F \to Exp$ is evaluated when a transition occurs and defines which values to consume/produce.

Further, we explain these concepts in more detail.

Color Sets and Markings. As in labeled Petri nets, in CPNs each place can be marked with one or more tokens, but tokens in CPNs are not "black", they have data values attached to them. Such a data value is called a *token color*. CPNs allow defining types *Types* for places that are called *color sets*, each place can have only tokens of the same (as the place type) color. Each $Type \in Types$ describes a set of values $Type \subseteq Values$. The color set function $colSet : P \cup Var \to Types$ assigns to each place $p \in P$ and variable $var \in Var$ a color set $colSet(p)$. A marking in a CPN assigns to each place a multiset of values of the type of this place, i.e., it is a function $m : P \to \mathbb{B}(Values)$ so that for each place p and each value $v \in m(p)$ holds $v \in colSet(p)$.

In the environments for CPN modeling and simulation [60, 61], the *CPN ML* language is used for defining color sets, declaring variables, defining expressions, and so on. The CPN ML language is based on the functional programming language *Standard ML* (SML) [62]. Similarly to other programming languages, a library with the implementation of the common data structures is available, from which we use only structure `list` in this thesis. Later in the thesis, we assume the use of the CPN ML language in CPN models.

In our CPN model of the queue (Figure 2.6), places $p0$ and $p1$ have a color set `ID` to distinguish different elements, e.g., identifiers for bags, which for technical reasons we model as a string (`colset ID = STRING`). Marking m_0 assigns initially a multiset of values to each place p, it is called an *initial marking*. To specify initial markings, `n‘X++m‘Y` in CPN ML denotes the multiset $[X^n, Y^m]$. For example, the expression `5‘a++10‘b` denotes the multiset $[a^5, b^{10}]$. Place $p0$ is initialized with a multiset of token identifiers for elements (e.g., bags) to enter the queue, e.g., `1‘pid1++1‘pid2++1‘pid3` is the multiset of elements $[pid1, pid2, pid3] = m_0(p0)$ as-

signed to place $p0$ initially. In Figure 2.6, place $p1$ is initialized with token identifiers of the queue, modeling how many items can still enter the queue. We distinguish queues through their unique identifiers, exactly as TSUs (bags), e.g., `k`*QID* describes that there are k tokens (queues) of value *QID*.

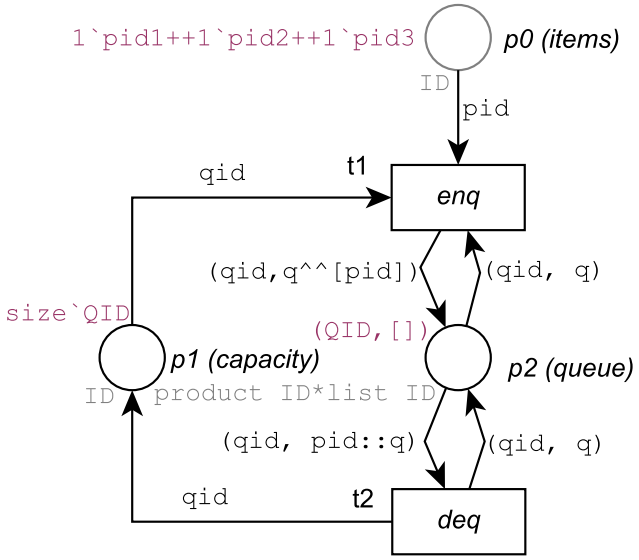


Figure 2.6: CPN model of the queue shown in Figure 2.5.

Place $p2$ has color set `colset QUEUE = product ID * list ID`, i.e., a pair (tuple) of a queue identifier (`ID`) and a list of element identifiers (of color set `ID`), the list implements the queue data structure. For example, a token on place $p2$ can have value $(QID, [pid1, pid2])$, describing a queue with identifier *QID* that contains two elements with identifiers $pid1$ (in the head) and $pid2$ (in the tail). The element to be dequeued first is $pid1$ because it is in the head. The initial marking of $p2$ is $m_0(p2) = [qid1, [\,]]$, where $[\,]$ denotes the single empty list $\langle\rangle$. As a result, the initial marking of the net is

- $m_0(p0) = [pid1, pid2, pid3]$,
- $m_0(p1) = [QID^{size}]$,
- $m_0(p2) = [(QID, [])]$,
- $m_0(p3) = []$.

Expressions and Variables. Each arc $(x, y) \in F$ is annotated with a structured expression $arcExp(x, y)$ that can be evaluated to a value of a particular type. This value is then consumed or produced when a transition occurs. The expressions have variables and constants as atoms and use various ML functions. In this thesis, we only use a

limited set of functions. In our running example of Figure 2.6, we define three variables: pid, qid of type ID and q of type list ID. The incoming arcs of transition $t1$ (labeled *enq*) in Figure 2.6 have the following expressions:

- arc $(p0, t1)$ has expression pid (variable pid of type ID),
- arc $(p1, t1)$ has expression qid (variable qid of type ID),
- arc $(p2, t1)$ has expressions (qid, q). This tuple consists of variable qid of type ID and variable q of type list ID.

The outgoing arcs of transition $t1$ have the following expressions:

- arc $(t1, p3)$ has expression pid (variable pid of type ID),
- arc $(t1, p2)$ has expression (qid, q^^[pid]). This tuple consists of variable qid of type ID and the expression q^^[pid] which describes that element qid is appended to list q.

Transition $t2$ has one expression different from the expressions of $t1$:

- outgoing arc $(p2, t2)$ has expression (qid, pid :: q), where expression pid :: q describes a token (qid, q') where list q' is decomposed into its head in variable pid and the rest elements in list q.

Variables pid, qid, and q are called *free variables*. If we bind values to the variables, we can evaluate the expressions on the arcs to concrete values. In this way, both individual identifiers and structures, such as a pair of a (1) concrete identifier and (2) list of identifiers, can be evaluated to a value $v \in Values$. A binding *bind* is a mapping of free variables to concrete values. For example, we can bind the following values to pid, qid, and q: $bind(\texttt{pid}) = pid3$, $bind(\texttt{qid}) = QID1$, $bind(\texttt{q}) = [pid2, pid1]$. We write $bind = \{(\texttt{pid}, pid3), (\texttt{qid}, qid1), (\texttt{q}, [pid2, pid1])\}$ in the following. We write $exp\langle bind\rangle$ for the result of evaluating expression *exp* under binding *bind*. If we evaluate the expression (qid, q^^[pid]) of outgoing arc $(t1, p2)$ using binding *bind*, we obtain the expression $(qid1, [pid2, pid1]\,\texttt{\^{}\^{}}\,[pid3])$ which we evaluate to $(qid1, [pid2, pid1, pid3])$, in other words we enqueued $pid3$ in the queue q . The evaluation of the expression pid of the outgoing arc $(t1, p3)$ using the same binding is basically just mapping of variable pid to value $pid3$, defined by binding *bind*.

Enabling and Occurrences. For transition $t1$, a binding *bind* yields a value x for the expression *exp* of each input arc $(p, t1)$, i.e., $x = exp(bind)$. Thus, evaluating the expression on the input arcs of $t1$ for *bind* tells:

1. which values must be on the respective input places to enable and fire $t1$ for *bind*,
2. which values are to be produced when firing $t1$.

Transition $t1$ is enabled for binding *bind* at marking m if the resulting value x of the expression of each arc $(p, t1)$ is on place p, i.e., $x \in m(p)$. An enabled transition can *occur*. When an enabled transition $t1$ occurs under binding *bind*,

- it *consumes* from place p a token carrying value x yielded by the binding for expression on the arc $(p, t1)$,
- it *produces* on each output place p' connected with $t1$ by an outgoing arc $(t1, p')$ that carries the result of the evaluated expression on arc $(t1, p')$.

We use $m \xrightarrow{(t,bind)} m'$ to write the occurrence of binding $(t, bind)$ that changes the marking from m to m' and call it a *transition occurrence*. Let us consider two markings m_1 and m_n. If for these markings exists a sequence of bindings $\sigma = \langle (t_1, bind_1), (t_2, bind_2), \ldots, (t_{n-1}, bind_{n-1}) \rangle$ such that $m_1 \xrightarrow{(t_1,bind_1)} m_2 \xrightarrow{(t_2,bind_2)} m_3 \ldots \xrightarrow{(t_{n-1},bind_{n-1})} m_n$, we write $m_1 \xrightarrow{\sigma} m_n$ and call this sequence a *transition occurrence sequence* of N.

In the CPN in Figure 2.6, we can bind `pid` to $pid1$; `qid` to QID; and `q` to the empty list. Transition $t1$ is enabled for this binding, firing $t1$ for this binding results in enqueuing of $pid1$ and consuming one token of place $p1(QID^{size})$, decreasing the free capacity of the queue. That leads to the following marking:

- $m_1(p0) = [pid2, pid3]$,
- $m_1(p1) = [QID^{size-1}]$,
- $m_1(p2) = [(QID, [pid1])]$,
- $m_1(p3) = [pid1]$.

At this new marking, the following bindings are possible:

- $bind_1 = \langle (pid, pid2), (qid, QID), (q, [pid1]) \rangle$ and $bind_2 = \langle (pid, pid3), (qid, QID), (q, [pid1]) \rangle$ for $t1$, and
- $bind_3 = \langle (pid, pid1), (qid, QID), (q, []) \rangle$ for $t2$.

Transition $t1$ with binding $bind_1$ occurs. As a result, element (bag) $pid2$ is enqueued, and another token of place $capacity^{QID}$ is consumed, decreasing the free capacity again. This occurrence leads to the marking representing two elements in the queue:

- $m_2(p0) = [pid3]$,
- $m_2(p1) = [QID^{size-2}]$,
- $m_2(p2) = [(QID, [pid1, pid2])]$,
- $m_2(p3) = [pid1, pid2]$.

To fire $t2$, we need a binding that bind *pid* to the head element of the queue in place $p2$, i.e., binding $\langle (pid, pid1), (qid, QID), (q, [pid2]) \rangle$. This occurrence leads to dequeuing of $pid1$ and producing ("returning") of one token on place *capacity*:

- $m_3(p0) = [pid3]$,

- $m_3(p1) = [QID^{size-1}]$,
- $m_3(p2) = [(QID, [pid2])]$,
- $m_3(p3) = [pid2]$.

We write this occurrence sequence, which changes marking m_0 to m_3, as $\sigma_{m0-m3} =$

- $\langle(t1, \langle(pid, pid1), (qid, QID), (q, [])\rangle)$,
- $(t1, \langle(pid, pid2), (qid, QID), (q, [pid1])\rangle)$,
- $(t2, \langle(pid, pid1), (qid, QID), (q, [pid2])\rangle)\rangle$.

So far, we recapped the syntax and semantics of CPNs. The next section recalls how the concept of time, required for modeling many complex systems, is supported in CPNs.

2.3.2 Timed Colored Petri Nets

In timed CPNs, timing information can be added to CPN models when the correctness of the system relies on the proper timing of the events. For example, in Figure 2.6, an element can only leave the queue on place $p2$ when the minimum waiting time t_{WQ} passes for the element. For that, a token in timed CPNs besides the color can carry the second value, called a *timestamp*. A color set of a place with such tokens must be *timed*, in CPN ML it is declared by appending the keyword `timed` to the color set name, e.g., `ID timed`. Additionally, the model has a *global clock*, representing *model time*. There are two widely used options for a model global clock: clock supporting integer timestamps, and clock supporting *real* timestamps. For models of this thesis, we select the more general global clock version with real timestamps.

A distribution of tokens on the places, together with their timestamps and the value of the global clock, is called a *timed marking*. The timestamp (a non-negative value) specifies the time at which the token is *ready* to be used, i.e., the earliest time when it can be consumed by an occurring transition. For example, the initial marking m_0 on place $p0$ at time 0 becomes $m_0(p0) = [pid1@0, pid2@0, pid3@0]$, i.e., each token is immediately ready at time 0. Without any time-related restrictions, a produced token is available immediately after it is produced. To model that a token is only available in some time after the transition occurred, the expression of an outgoing arc of the transition can be appended with a *delay*. In the CPN Tools, the following syntax is used: $exp\,@delay$. As a result, a token produced by an occurrence of the corresponding transition at time $time_1$ becomes ready at time $time_1 + delay$. Note that formally, these timed extensions just enrich the "structure" of the possible *Values*, and *Types* assigned to places P and variables *Var* by function *colSet*, and of the expressions assigned to arcs F by function *arcExp*; see [49] for details.

Let $\mathbb{T}$ be the set of time durations and timestamps, e.g., the rational or real numbers. A *state* of the CPN at time $ts \in \mathbb{T}$ is a tuple (m, ts) of a marking m (over timed

tokens) and a timestamp *ts*. We write $m \xrightarrow{(t,bind)@time} m'$ for an occurrence of transition t at time *time* that changes marking m to m', i.e., for an occurrence of transition t that changes state $(m, time)$ to state $(m', time)$. Note, we write markings (not states) before and after the arrow because this notation allows for chaining multiple labeled occurrences into a sequence, while the time information for the state before and after occurrences can be easily derived from the occurrence time *time* and marking m and m' respectively.

Now, we convert the CPN model in Figure 2.6 to a timed one because we want to ensure the minimum waiting time $\mathrm{t1_{wQ}}$ of elements in queues on place $p2$. For that, we

1. add a new place $p3$ of color `ID` (see Figure 2.7),
2. add an arc from transition $t1$ to place $p3$ with arc inscription $pid@\mathrm{t1_{wQ}}$, and
3. add an arc from place $p3$ to transition $t2$ with arc inscription *pid*.

As a result, each occurrence of transaction $t1$ produces a token with element identifier *pid* on place $p3$, whose presence is required to remove this element from the queue later. Arc inscription *pid@* ensures that such a token becomes ready on place $p3$ only after waiting time $\mathrm{t1_{wQ}}$.

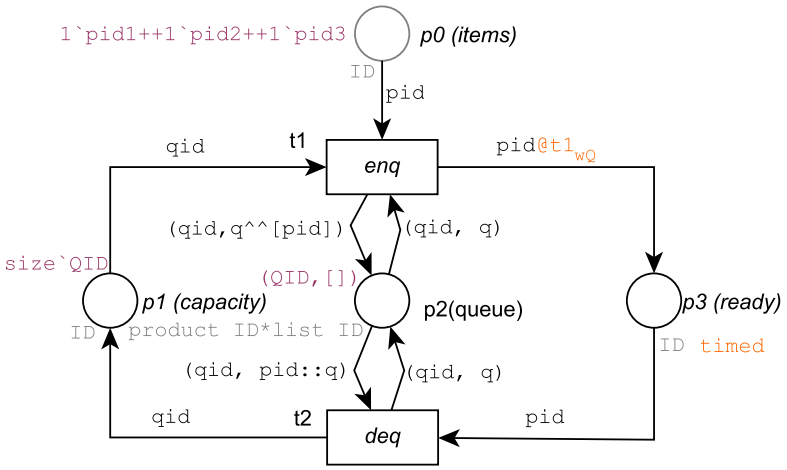


Figure 2.7: *Timed* CPN model of the queue in Figure 2.5.

Let us reconsider sequence σ_{m0-m3} (see the previous section) for this timed CPN, assuming $\mathrm{t1_{wQ}} = 10$, $size = 3$ and initial marking m_0 at model time 0, i.e., initial state $(m_0, 0)$. In the CPN semantics [63] time has to advance at most until the next transition becomes enabled. However, input places $p0$ and $p1$ are not timed, so let us assume that two bindings for $t1$ (i.e., bindings $\langle (pid, pid1), (qid, QID), (q, []) \rangle$ and

$\langle(pid, pid2), (qid, QID), (q, [pid1])\rangle$) occurred at time 0 and 1 respectively, i.e., at time 1 we have the following marking m_1:

- $m_1(p0) = [pid3@0]$,
- $m_1(p1) = [QID@0]$,
- $m_1(p2) = [(QID, [pid1, pid2])@1]$,
- $m_1(p3) = [pid1@10, pid2@11]$.

Note, in state $(m_1, 1)$ transition $t2$ is not enabled because token $pid1$ on $p3$ becomes available at time 10. When time advances, at time 10 at marking $m2$ transition $t2$ is enabled with the third binding $(t2, \langle(pid, pid1), (qid, QID), (q, [pid2])\rangle)$ and fires, leading to the following marking m_3:

- $m_3(p0) = [pid3@0]$,
- $m_3(p1) = [QID@0, QID@10]$,
- $m_3(p2) = [(QID, [pid2])@10]$,
- $m_3(p3) = [pid2@11]$.

In Chapter 6, we use CPN models, including an extension of the model considered in this example, for defining a dedicated synchronous proclet system [48] for modeling MHSs, i.e., for defining a model consisting of multiple CPN models connected into a system through synchronous channels, which are a piece of syntax and semantics defined for such systems.

2.4 Events, Attributes, Event Logs, and Event Tables

The behavior of a process can be captured through the events it generates. In process mining, such events are typically used for various kinds of process analysis. Usually, events are grouped in traces (i.e., sequences) such that each trace contains events related to the same case. For example, if we consider a web shop, one case can be a shopping session of a customer, and a trace of such a case contains events related to the customer's actions, e.g., logging in, search requests, payments, etc. For this example, a customer session identifier, assigned to each event as its attribute, can be used as a unique case identifier for grouping events into a trace. Such an attribute is called a *case notion*. Most process mining techniques require that an attribute carrying case identifiers is defined for each event.

While event logs where events have a single case notion are widely used in process mining, this view on event data may be too simplistic for complex processes. For example, in a web shop, besides customer-related processes, other processes such as a delivery process, may exist. So, a payment can initiate the start of a delivery case. Thus, event *payment* belongs to two cases: (1) a shopping session case, and (2) a delivery case. Each delivery case has its own case identifier, e.g., a delivery identifier.

We say that event *payment* has two case notions: a shopping session and delivery. In general, event logs where events have multiple case notions allow for describing multiple processes in one event log.

In this thesis, we consider MHSs that have entities of different types (case notions), for example, bags, resources, conveyors, etc. The events they generate are typically recorded into database tables and exported into CSV files for analysis. Recently, such data started to be represented in object-centric formats, such as *Object-Centric Event Logs* (OCEL) [64], instead of single-case notion event logs in the *XES format* [65]. However, in this thesis, we use "raw" *event tables* for representing multi-case notions event data because it is a much simpler structure that still suffices our needs.

In an event table, each row is an event, and each column contains values of event attributes, where one or multiple attributes are case notions. Each event refers to at most one case for one case notion. An event table allows for deriving a "classical" single-case notion event log for one of its case notions. This event log can be seen as a *view* on the event table data.

In the following, we define event-related universes, a generic event table, whose events have some minimal attribute set, and then a *complete* event table, whose attribute set is sufficient for techniques considered in this thesis. Finally, we define "classical" event logs and traces.

Note, we use the term *process step* instead of *activity* for describing MHS processes, while we keep the classical terms *activity* and *activity name* for the corresponding event attributes, as an event is a general concept that is not necessarily related to MHS processes.

First, we define the universes required for our event model.

Definition 2.7 (Events-related universes)**.** *We write the following notations for the universes:*

- $\mathbb{T}$ *denotes the set of time durations and timestamps, e.g., the rational or real numbers.*
- *AN denotes the set of all possible event attribute names.*
- $\mathbb{CN} \subseteq$ *AN denotes the set of all possible case notions.*
- *Val denotes the set of all possible event attribute values.*
- $\mathscr{I} \subseteq$ *Val denotes the set of* identifiers*.*
- *Act* $\subseteq$ *Val denotes the set of all possible activity names.*

The definition of a generic event table reads as follows.

Definition 2.8 (Event table)**.** *Let $\mathscr{E}$ be the event universe, i.e., the set of all possible event identifiers. An* event table $ET = (CN, E, \#)$ *has:*

1. *a non-empty set of case notions* $CN \subseteq \mathbb{CN}$*,*

event ID	pid	qid	time	act
e1	*pid*1	*qid*1	1	a
e2	*pid*1	*qid*1	2	b
e3	*pid*2	$\perp$	3	d
e4	*pid*2	*qid*2	4	a

Table 2.1: Complete time-monotone event table.

2. *a set of events* $E \subseteq \mathscr{E}$,
3. *an attribute function* $\# \in AN \times E \nrightarrow Val$ *that maps an attribute name* $n \in AN$ *and event* $e \in E$ *to the value of* n. *If the value is undefined, it returns* $\perp$. *We write* $\#_n(e)$ *for calling* $\#$ *for attribute name* n *and event* e.

Function $\#$ *must define the following attribute values for each event* $e \in E$:

1. *an activity name* $\#_{\text{act}}(e) \in Act$,
2. *at least one case identifier* $\#_{cn}(e) \in \mathscr{I}$ *for a case notion* $cn \in CN$, *i.e.,* $\exists cn \in CN \mid \#_{cn}(e) \neq \perp$.

A case notion $cn \in CN$ *is a* global case identifier *iff for each* $e \in E$ $\#_{cn}(e)$ *is defined.*

In an event table, some timestamp information can be missing, for example, if some events were restored through a log repair technique that does not restore event timestamps. Because the timestamp information is required for many analysis techniques we consider in this thesis, we define a *complete event table*, where each event has a defined timestamp, as well.

Definition 2.9 (Complete and incomplete event table)**.** *An event table* $ET = (CN, E, \#)$ *(Definition 2.8) is* complete *iff function* $\#$ *defines a timestamp value for each event* $e \in E$, *i.e.,* $\#_{\text{time}}(e) \neq \perp$. *Otherwise, ET is called an* incomplete *event table.*

Table 2.1 shows a complete event table $(\{\text{pid}, \text{qid}\}, \{e1, e2, e3, e4\}, \#)$ in a tabular form, where each row is an event in E, and the columns contain the following information, i.e., the cell values define the attribute function $\#$:

- column *event ID* contains event identifiers,
- column pid contains values of attribute pid carrying process identifiers,
- column qid contains values of attribute qid carrying queue identifiers,
- column time contains values of attribute time carrying event timestamps,
- column act contains contains values of attribute act carrying activity names.

If an attribute value is undefined, the corresponding cell contains $\perp$, e.g., $\#_{\text{qid}}(e3) = \perp$. pid is a *global case identifier* because it is defined for each event in ET. The value of another case notion $\text{qid} \in CN$ is undefined for $e3$, so identifier qid is not global.

Let us consider event $e1 \in ET$, which has the following attributes:

- $\#_{\text{act}}(e) = a \in Act$,
- $\#_{\text{pid}}(e) = pid1 \in \mathscr{I}, \text{pid} \in CN$,
- $\#_{\text{qid}}(e) = qid1 \in \mathscr{I}, \text{qid} \in CN$,
- $\#_{\text{time}}(e) = 1 \in \mathbb{T}$.

When we refer to an event as a set of attribute name/value pairs, we use a simplified notation. For example, we write $\langle(\text{act}, a), (\text{pid}, pid1), (\text{qid}, qid1), (\text{time}, 1)\rangle$ for $\{(\text{act}, \#_{\text{act}}(e1)), (\text{pid}, \#_{\text{pid}}(e1)), (\text{qid}, \#_{\text{qid}}(e1)), (\text{time}, \#_{\text{time}}(e1))\}$.

Additionally, we define a *time-monotone* event table for event tables where no two events of the same case have identical timestamps.

Definition 2.10 (Time-monotone event table)**.** *An event table $ET = (CN, E, \#)$ (Definition 2.8) is* time-monotone *iff for distinct $e, e' \in E, cn \in CN'$ holds if $\#_{cn}(e) = \#_{cn}(e')$ then $\#_{\text{time}}(e) \neq \#_{\text{time}}(e')$.*

The complete event table in Table 2.1 is time-monotone.

In general, an event table contains events related to multiple case notions. However, in many cases, a simpler "classical" single-notion view on data is required, for example, to apply existing process mining techniques. For that, we define a single case notion, or "classical", event log, its cases, and traces.

The event log definition reads as follows.

Definition 2.11 (Event log, complete event log, time-monotone event log)**.** *An* event log *$L = (cn, E, \#)$ is a set E of events, and a designated* case notion attribute *$cn \in \mathbb{CN}$. Similarly to Definition 2.8, an attribute function $\# \in AN \times E \not\to Val$ maps an attribute name $n \in AN$ and event $e \in E$ to the value of n. If the value is undefined, it returns $\perp$. We write $\#_n(e)$ for calling $\#$ for attribute name n and event e. Function $\#$ must define for each event $e \in E$:*

1. *an activity name $\#_{\text{act}}(e) \in Act$,*
2. *a case identifier $\#_{cn}(e) \in \mathscr{I}$.*

We call an event log complete *if each event has a timestamp defined, i.e., $\forall e \in E, \#_{\text{time}}(e) \neq \perp$, otherwise the event log is* incomplete*. We call an event log* time-monotone *if for any two events with the same defined identifier id of case notion cn and defined timestamps, the values of the timestamps are different, i.e., $\forall e, e' \in E, \#_{cn}(e) = \#_{cn}(e'), \perp \neq \#_{\text{time}}(e) \neq \#_{\text{time}}(e') \neq \perp$. Let $\mathscr{L}$ be the set of all event logs.*

We can obtain an event log from an event table as follows.

Definition 2.12 (Event log from event table)**.** *Let $ET = (CN, E, \#)$ be an event table (Definition 2.8). Let $cn \in CN$ be a case notion. Let the set of events E' containing events that have case notion cn defined, i.e., $E' = \{e \mid e \in E, \#_{cn}(e) \neq \perp\}$. $L = (cn, E', \#\big|_{E' \times AN})$ is an event log obtained for case notion cn from event table ET.*

We write L_{cn}^{ET} for an event log obtained from ET for case notion cn.

Now we define traces and cases of an event log.

Definition 2.13 (Cases and traces)**.** *$L = (cn, E, \#)$ be an event log with case notion attribute cn (see Definition 2.11).*

The set of cases *in L with respect to cn is $cn(L) = \{\#_{cn}(e) \mid e \in E\}$, i.e., all case identifier values in L.*

All events that have the same case identifier $id \in cn(L)$ are correlated to *id, i.e., $corr(L, cn = id) = \{e \in E \mid \#_{cn}(e) = id\}$.*

A trace *of case id is a non-empty sequence $\langle e_1, \ldots, e_n \rangle$ of all events in E correlated to id that preserves the time ordering, i.e., $corr(L, cn = id) = \{e_1, \ldots, e_n\}$ and $\forall 1 \leq i < j \leq n$ holds if $\#_{\text{time}}(e_i) \neq \perp$ and $\#_{\text{time}}(e_j) \neq \perp$ then $\#_{\text{time}}(e_i) \neq \#_{\text{time}}(e_j)$.*

In Table 2.1, a trace for case notion pid and identifier $pid1$ is $\langle e1, e2 \rangle$. Its events are ordered by time, i.e., $\#_{\text{time}}(e1) < \#_{\text{time}}(e2)$.

2.5 Chapter Summary

In this chapter, we recalled the key concepts of the set theory, and Petri nets, and provide definitions of various event data structures used throughout the thesis. We introduced notation on sets, multisets, sequences, relations, functions, and partial orders. We recalled Petri-net with black tokens, and discussed their syntax and semantics, with a focus on their partial-order semantics interpretation. Then, we recalled CPNs, their semantics, and how they model the time aspects of processes. Finally, we defined the event-related data structure, such as events, event logs, and event tables. All these notations, concepts, and definitions are extensively used in the remainder of this thesis.

Chapter 3

Fine-Grained Description of Processes Performance from Event Data

Most process mining techniques for performance analysis are based on a process model. Usually, a process model is discovered automatically (or created manually) first and then used, for example, to annotate the model elements with some performance information [1]. Intuitively, it implies that a performance description exists only in the scope of the process description that the model materializes. As a result, any bias that a process model introduces "propagates" to the description and analysis of the process performance, impacting the analysis outcome. Moreover, such techniques cannot be used if a process model is unavailable. In this chapter, we propose a technique for performance description, called *performance spectrum*. It does not inherit any bias of a process model because it is a model-less technique, i.e., the performance spectrum is derived directly from event data.

3.1 Motivation

Performance analysis is an important element in process management relying on precise knowledge about actual process behavior and performance to enable improvements [66]. Descriptive performance analysis has been intensively studied within process mining, typically by annotating discovered or hand-made models with time-related information from event logs [1, 67, 68]. These descriptive models provide aggregate measures for performance over the entire data, e.g., an average, median, or maximum waiting time between two process steps. For example, a model in Fig-

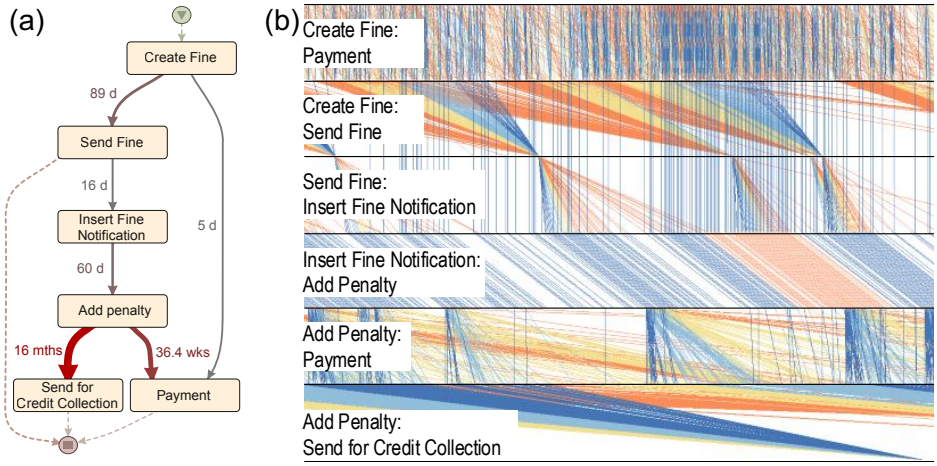


Figure 3.1: Performance analysis using a graph-based model (a), and the performance spectrum (b).

ure 3.1(a) shows a process model of the RTFM process that we briefly considered in Section 1.1.3. In this model, each arc is annotated with an average time between directly following process steps that the arc connects, e.g., the average duration between steps *Create Fine* and *Payment* is five days. Models for predicting the waiting time until the next step or remaining case duration learned from event data distinguish different performance classes or distribution functions based on case properties [36, 69, 70, 71].

However, these techniques assume the time-related observations to be taken from stationary processes that are executed in isolation, i.e., that distribution functions describing the performance of a case do not change over time and do not depend on other cases. These assumptions are often made by a lack of a more precise understanding of the (changes in) process performance across cases and over time.

In this chapter, we address **RQ-1**. *Given an event log of a process, how to describe the performance information the log contains in a way that reveals both the performance of individual cases and how these cases interact over time?*. It considers the problem of descriptive analytics of the process behavior and performance *over time*. In particular, we aim to provide the *comprehensive* description of the raw process behavior without enforcing prior aggregation of data, the representational bias of an algorithm, or a particular formal model.

We approach the problem through *visual analytics*, which employs structuring of data in a particular form that, when visualized, allows offloading the actual data processing to the human visual system [72] to identify patterns of interest for the subsequent analysis. We propose a new simple model for event data, called the *performance*

spectrum, and its visualization. Figure 3.1(b) shows the performance spectrum of the data used to discover the model in Figure 3.1(a) over a 20-month period. The performance spectrum describes the event data in terms of *segments*, i.e., pairs of related process steps. The performance of each segment is measured and plotted for any occurrences of this segment over time and can be classified, e.g., regarding the overall population.

The visualization in Figure 3.1(b) shows that the represented cases performed very differently due to systematic and unsystematic *variability of performance in the different steps over time* and synchronization of multiple cases. We implemented this visualization in an *interactive* tool called the *Performance Spectrum Miner* (ProM package "Performance Spectrum") [46].

Exploring the performance spectrum of real-life logs often reveals numerous patterns in the process behavior as shown in Figure 3.1(b) that cannot be seen in process models as in Figure 3.1(a). To enable documenting and conceptualizing these patterns for further analysis, we propose a *taxonomy* for describing elementary patterns in the performance spectrum, which is obtained through analysis of event logs of various processes. We evaluated the performance spectrum and the taxonomy on 12 real-life logs of business and logistics processes. Many elementary patterns, as well as larger patterns composed of elementary ones, were detected throughout different event logs. We showed how these patterns reveal insights into the interplay of the control flow, resource, and time perspective of processes. For example, the performance spectra showed that

1. the case performance may depend on the performance of the other cases,
2. the performance generally varies over time (non-stationarity), and
3. many processes exhibit temporary or permanent concept drift.

We performed also an empirical study that proved that the same performance patterns can be (re)-identified in the processes by independent analysts without skills in working with the performance spectrum. We reported also on a case study performed at Vanderlande for identifying and explaining performance problems in very large logistics processes. Further, we found that each process has a characteristic *signature* of the patterns in its performance spectrum and that similar signatures indicate processes similar not only in the control flow but also in the performance perspective.

The remainder of this chapter is structured as follows. We formally define the performance spectrum in Section 3.2 and introduce the taxonomy for performance pattern parameters in Section 3.3, including the reports on our evaluation using real-life event logs, and re-identification of performance patterns by independent analysts. We introduce the concept of the multi-channel performance spectrum in Section 3.2.3 and provide the evaluation in Section 3.5. We conclude with the discussion of findings and future work in Section 3.6.

3.2 Performance Spectra

In this section, we introduce the concept of the performance spectrum. The performance spectrum is a data structure, which can be computed from event data. In this thesis, we consider computing performance spectra from event data in the form of an event log (see Definition 2.12). We first introduce the performance spectrum "building blocks", then define the performance spectrum itself, and finally define its aggregate, which allows quantifying various characteristics over the performance spectrum.

3.2.1 Segments, Segment Occurrences, and Performance Classifiers

Let as consider an event log $L = (cn, E, \#)$ (see Definition 2.11), and its trace $\sigma = \langle e_1, \ldots, e_n \rangle$, $n > 1$ (see Definition 2.13). Each pair of events $(e_i, e_{i+1}), 0 < i < n$ of σ represents a *step* from activity $\#_{\text{act}}(e_i)$ to activity $\#_{\text{act}}(e_{i+1})$, occurred in an instance $id = \#_{cn}(e_i)$ of case notion cn. In general, such a step is a progression from a process step (or activity) a to a process step (or activity) b, hand-over of work from a resource a to a resource b, or transportation of materials from a location a to a location b that can occur in instances of case notion cn. We call this step a *segment*. Its formal definition reads as follows.

Definition 3.1 (Segment)**.** *Let $a, b \in Act$ be activity names. A tuple $seg = (a, b) \in Act \times Act$ is a* segment.

When a segment $seg = (a, b)$ is observed in a trace of L, we say that this segment *occurred*. The definition of a segment occurrence reads as follows.

Definition 3.2 (Segment occurrence)**.** *Let a time-monotone event log $L = (cn, E, \#)$ (see Definition 2.11), and segment $seg = (a, b)$ (see Definition 3.1). Events $e, e' \in E, e \neq e'$ comprise a comprise a* segment occurrence *(e, e') if e' directly follows e with respect to their common case identifier for case notion cn, i.e.,*

- $\#_{cn}(e) = \#_{cn}(e')$,
- $\#_{\text{time}}(e) < \#_{\text{time}}(e')$,

and $\forall e'' \in E, e'' \neq e, e'' \neq e', \#_{cn}(e'') = \#_{cn}(e)$ holds that either

- $\#_{\text{time}}(e) > \#_{\text{time}}(e'')$ *or*
- $\#_{\text{time}}(e') < \#_{\text{time}}(e'')$.

We write $occ(a, b, L)$ for the set of all segment occurrences of (a, b) in L, and $occ(L)$ for the set of all segment occurrences in L.

Each segment occurrence allows for measuring the time between occurrences of the segment activities, e.g., the time between occurrences of activities a and b for

segment (a, b). For example, an occurrence (e, e') of segment (a, b), where $\#_{\text{time}}(e) = t_a, \#_{\text{time}}(e') = t_b$ is shown in Figure 3.2. There are two axes a and b in this figure, with

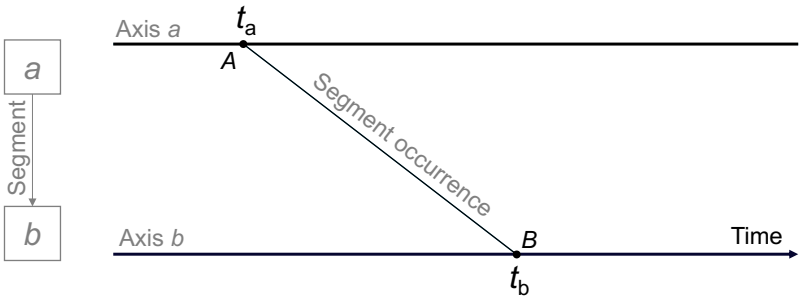


Figure 3.2: Occurrence of segment (a, b).

some space in between. Each axis represents time. To visualize segment occurrence (e, e'), we draw a point A on axis a at time t_a, then draw a point B on axis b at time t_b, and connect them by a line AB, which represents (e, e'). This example shows that segment occurrences, when visualized, show the time between events of each segment occurrence. Further, *multiple* occurrences of segment (a, b) are visualized in Figure 3.3. In this figure, the angle or length of each line, visualizing a segment

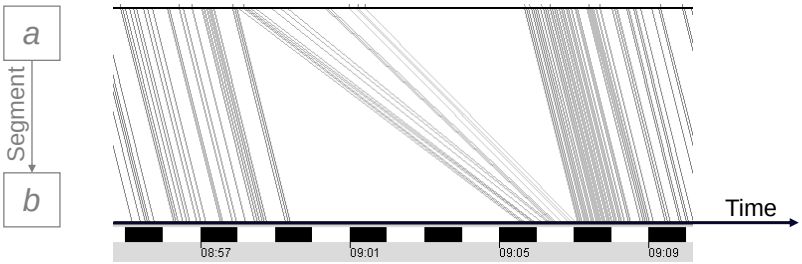


Figure 3.3: Multiple occurrences of segment (a, b).

occurrence, can be used for estimating its duration, i.e., a time interval $t_b - t_a$. We allow the analyst to classify each segment occurrence duration with respect to the other observations for the same segment or the whole event log.

For example, if a histogram $H = H(a, b, L) \in \mathbb{B}(\mathbb{T})$ describes how often all the *time intervals* $(t_b - t_a)$ between a and b have been observed in event log L, a specific classification function can show to which quartile of the histogram H a segment occurrence belongs. It allows for faster recognition of various patterns indicating, for example,

bottlenecks in the process. Another example is a classification function that depends on the remaining time until the case completion. We define such classification functions as follows.

Definition 3.3 (Performance classifier)**.** *Let a time-monotone event log $L = (cn, E, \#)$ (see Definition 2.11), and set $O = occ(L)$ of all segment occurrences in L. Let set C be a finite set of performance classes. Let $CN' = \{cn\}$ be a set of the log case notions. We call the function $\mathbb{C} : O \times \mathscr{L} \rightarrow C$ that maps each observed segment occurrence of L and log L itself to a performance class $c \in C$, a* performance classifier.

The segment occurrences of Figure 3.3, classified according to their duration into four performance classes, are shown in Figure 3.4, where each class is encoded through a unique color, according to the legend at the bottom. Comparing Figure 3.3

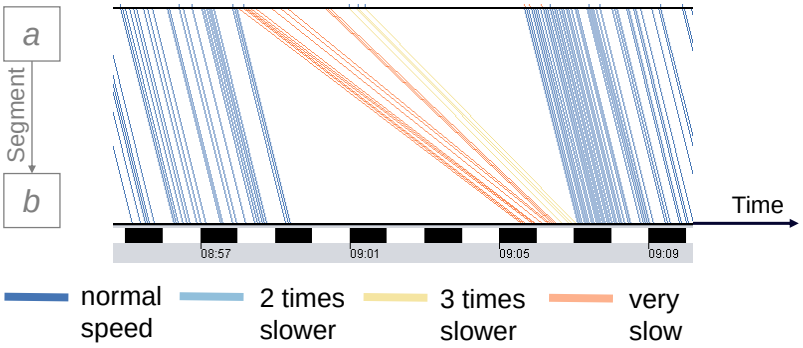


Figure 3.4: Classified segment occurrences.

and Figure 3.4, one can conclude that the use of a performance classifier and the color coding of performance classes within segment occurrence visualization allows, for example, for quicker recognition of possible bottlenecks, e.g., one shown by the yellow (performance class "3 times slower") and orange (performance class "very slow") lines in Figure 3.4. In general, we assume that a performance classifier can use any attributes of events e, e' forming a segment occurrence, any occurrences in O, and any attributes of any event in L if needed. Potentially, that provides richer possibilities for segment occurrence classification and analysis.

In the next section, we define data structures for multiple segment occurrences of single and multiple segments.

3.2.2 Performance Spectra

In this section, we define a data structure for segment occurrences, similar to the one visualized in Figure 3.4 and discussed above, first for a single segment, and then for multiple ones.

The occurrences of a segment *seg* classified by a performance classifier $\mathbb{C}$ describe the performance observed for *seg* in event log L. We call this description a performance spectrum of a segment, whose formal definition reads as follows.

Definition 3.4 (Segment performance spectrum)**.** *Let a time-monotone event log* $L = (cn, E, \#)$ *(see Definition 2.11), segment* (a, b) *(see Definition 3.1), and performance classifier* $\mathbb{C}$ *(see Definition 3.3). The* segment performance spectrum *of* (a, b) *for* L *is a multiset* $\mathbb{PS}_L^{\text{seg}}((a, b), \mathbb{C}) =$

- $[(t_a, t_b, c) \mid (e, e') \in occ(a, b, L), c = \mathbb{C}((e, e'), L)] \in \mathbb{B}(\mathbb{T} \times \mathbb{T} \times C)$.

The visualization of a segment performance spectrum is exactly the same as the visualization of multiple classified occurrences of a single segment, already shown in Figure 3.4. Next, we define a performance spectrum of multiple segments, in order to describe a larger "piece" of the process performance. For that, we first define a data structure for describing multiple segments as follows.

Definition 3.5 (Segment series)**.** *Let SEG be a non-empty set of segments (see Definition 3.1), i.e.,* $SEG \subseteq \{(a, b) \mid a, b \in Act, a \neq b\}$. *A* segment series *is a sequence of these segments* $\langle seg_1, \ldots, seg_n \rangle, n > 0, seg_i \in SEG$.

Further, we define a data structure for referring to the performance spectra of a segment series and performance classifier.

Definition 3.6 (Layer)**.** *Let a time-monotone event log* $L = (cn, E, \#)$ *(see Definition 2.11), segment series SEG (see Definition 3.5), and performance classifier* $\mathbb{C}$ *(see Definition 3.3). A* layer *is a tuple* $lr = (SEG, \mathbb{C})$.

Consequently, we lift Definition 3.4 to a layer by organizing the performance spectra of the individual segments of a layer into a sequence.

Definition 3.7 (Layer performance spectrum)**.** *Let a time-monotone event log* $L = (cn, E, \#)$ *(see Definition 2.11), and layer* $lr = (SEG, \mathbb{C}) = (\langle seg_1, \ldots, seg_n \rangle, \mathbb{C})$ *(see Definition 3.6). The* layer performance spectrum *of* lr *for event log* L *is a sequence* $\mathbb{PS}_L^{\text{lr}}(lr) = \mathbb{PS}_L^{\text{lr}}((\langle seg_1, \ldots, seg_n \rangle, \mathbb{C})) = \langle \mathbb{PS}_L^{\text{seg}}(seg_1, \mathbb{C}), \ldots, \mathbb{PS}_L^{\text{seg}}(seg_n, \mathbb{C}) \rangle \in \mathbb{B}(\mathbb{T} \times \mathbb{T} \times C)^*$.

In the following, we usually drop "layer" when referring to the layer performance spectrum.

The performance spectrum provides a fine-grained performance description on the level of individual cases and segment occurrences. Nevertheless, this information can be difficult to precept when there is a large number of segment occurrences, and/or when multiple occurrences overlay, i.e., multiple occurrences are seen as one when visualized. In the next section, we introduce the concept of the aggregate performance spectrum, which represents quantified information about segment occurrences and their performance classes.

3.2.3 Aggregate Performance Spectra

Let us consider the segment performance spectrum of segment *seg* whose occurrences are classified by performance classifier $\mathbb{C}$. As the number of performance classes in the domain C of $\mathbb{C}$ is *finite* by definition, this performance spectrum can be *aggregated* over the "bins" of some chosen, fixed, non-zero duration. For each bin b_j, and each class $c \in C$, we count how many occurrences of segment *seg* of performance class c occurred during the time interval defined by b_j. The definition of the resulting aggregate performance spectrum reads as follows.

Definition 3.8 (Aggregate performance spectrum)**.** *Let a time-monotone event log $L = (cn, E, \#)$ (see Definition 2.11), a segment (a, b) (see Definition 3.1), and a performance classifier $\mathbb{C}$ (see Definition 3.3). Let $PS = \mathbb{PS}_L^{seg}((a, b), \mathbb{C})$ be a segment performance spectrum (see Definition 3.4). Let period $p > 0 \in \mathbb{T}$ be a* bin size*, and let $g \in \{start, pending, end\}$ be a parameter called* grouping*. The occurrences of segment (a, b) in bin $j \in \mathbb{N}$ (of length p) regarding grouping g are a multiset b_j such that:*

- $b_j = [(t, t', c) \in PS \mid j \cdot p \leq t < (j+1) \cdot p]$ *if $g = start$,*
- $b_j = [(t, t', c) \in PS \mid j \cdot p \leq t' < (j+1) \cdot p]$ *if $g = end$, and*
- $b_j = [(t, t', c) \in PS \mid j \cdot p > t \wedge t' \geq (j+1) \cdot p]$ *if $g = pending$ (i.e., the segment starts before the start of the bin, and ends after (or at) the end of the bin).*

The aggregate performance spectrum *over performance spectrum PS, bin j, and grouping g is the vector $v_j = \langle v_j^1, \ldots, v_j^k \rangle \in \mathbb{N}^k$ counting how often performance class c^i occurred in bin v_j: $v_j^i = |[(t, t', c^i) \mid (t, t', c^i) \in b_j]|$. Let $\mathbb{APS}_L^j((a, b), \mathbb{C}, g, p, j) = v_j$.*

For example, in Figure 3.5(d) the aggregate performance spectrum for segment (a, b) over bin 7 and grouping $g = end$ is a vector $\langle 0, 1, 2 \rangle$, which counts ends of all segment occurrences in this bin: zero for class c^1, one for c^2 (b_7) and two for c^3 (points b_{5-6}).

An aggregate performance spectrum has three dimensions:

1. the segment (a, b),
2. the parameters describing the bins, i.e., the performance classifier $\mathbb{C}$, the grouping g, and the period p,
3. the bin number j.

To simplify notation, we call the bin parameters $ch = (\mathbb{C}, g, p)$ a *channel*, and write $\mathbb{APS}_L^j(seg, ch, j)$ for the aggregate vector v_j of Definition 3.8.

Multi-Channel Performance Spectra. Now, we show that a bin $\mathbb{APS}_L^j(seg, ch, j)$ of an aggregate performance spectrum is a basic building block for describing various aspects of the process performance over time. In Figure 3.6, each bin of the aggregate performance spectrum is placed into a three-dimensional space defined by a segment series *SEG* (see Definition 3.5), *channel series* $CH = \langle ch_1, \ldots, ch_x \rangle$, and time interval of

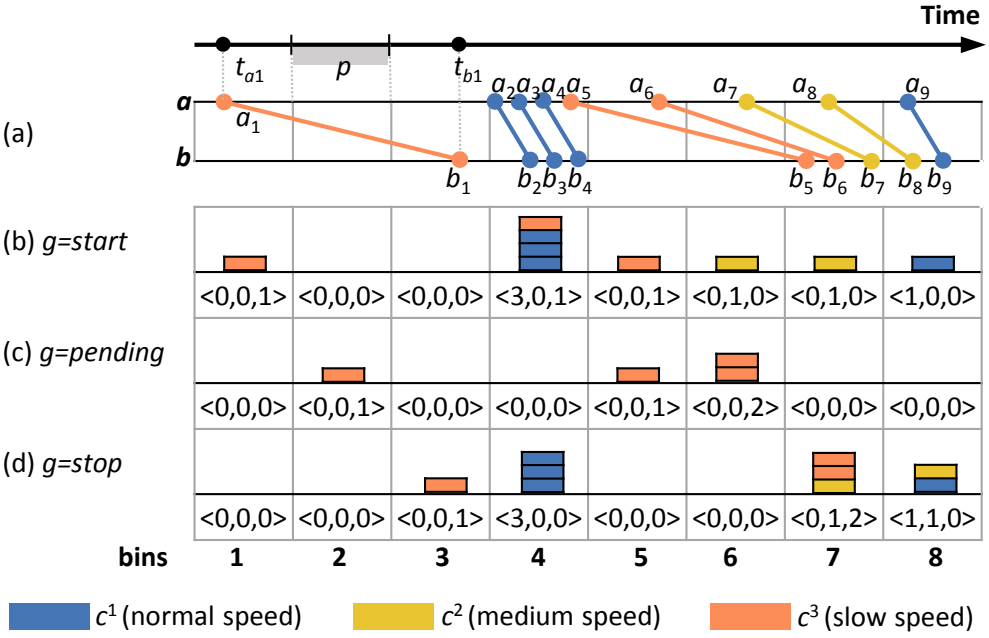


Figure 3.5: In the performance spectrum (a) the color-coded lines show cases with different speed classes, while the aggregate performance spectra with various grouping (b-d) capture various performance aspects of case handling for each time window (bin).

bins $[s,e] = \langle s, s+1, \ldots, e \rangle$ of interest. Adopting notation from algebra software, we let the arguments of $\mathbb{APS}_L(\cdot,\cdot,\cdot)$ range over the sequences of segments, channels, and bin numbers to denote the rows, columns, matrices, and cubes of bins along those dimensions. Let an event log L, and a segment series $SEG = \langle (a_1, b_1), \ldots, (a_n, b_n) \rangle$ (according to Definition 3.5). Let a channel series $CH = \langle ch_1, \ldots, ch_x \rangle$ be a sequence of channels (of identical period p), $ch \in CH$, and let $[s,e] = \langle s, s+1, \ldots, e \rangle$ be a sequence of bin numbers, $j \in [s,e]$. We write $\mathbb{APS}^{\text{seg}}_L(SEG, ch, j)$ for the column vector $\langle \mathbb{APS}^{\text{j}}_L((a_1,b_1), ch, j), \ldots, \mathbb{APS}^{\text{j}}_L((a_n,b_n), ch, j) \rangle^\top$. Note that this vector consists of vectors v^i_j for each segment (a_i, b_i) and bin j. In Figure 3.6, such a column vector corresponds to a column of blocks of size $n \times 1 \times 1$, e.g., area (1).

We write $\mathbb{APS}^{\text{seg}\times\text{b}}_L(SEG, ch, [s,e])$ for the row vector

- $\langle \mathbb{APS}^{\text{seg}}_L(SEG, ch, s)$,
- $\ldots$,
- $\mathbb{APS}^{\text{seg}}_L(SEG, ch, e) \rangle$.

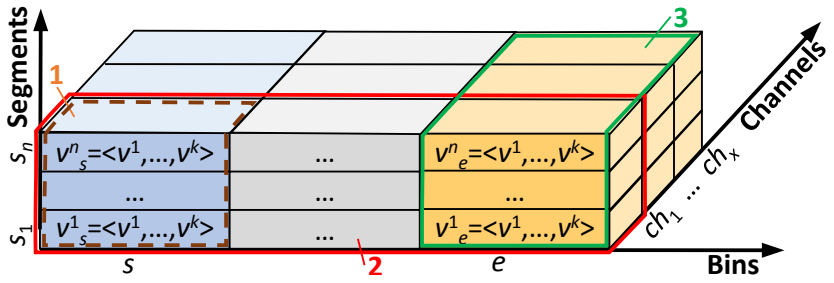


Figure 3.6: Multi-channel performance spectrum.

Note that each j^{th} entry of this vector corresponds to a column vector $\mathbb{APS}_L^{\text{seg}}(SEG, ch, j)$. In Figure 3.6, $\mathbb{APS}_L^{\text{seg}\times\text{b}}(SEG, ch, [s, e])$ corresponds to a frontal 'slice' of size $n \times 1 \times (e - s + 1)$ for channel ch , e.g., area (2). We use this matrix to *visualize* the aggregate performance spectrum of segment series SEG over the time period $[s, e]$ in a single channel ch. For example, Figure 3.5(b) visualizes one row of such a matrix and Figure 3.6 visualizes the entire matrix.

Performance analysis typically requires considering the information from *multiple* channels during the same time period. We write $\mathbb{APS}_L^{\text{seg}\times\text{ch}}(SEG, CH, j)$ for the column vector

- $\langle \mathbb{APS}_L^{\text{seg}}(SEG, ch_1, j)$,
- $\ldots$,
- $\mathbb{APS}_L^{\text{seg}}(SEG, ch_x, j)\rangle^\top$.

Note that each r^{th} entry of this vector corresponds to a column vector $\mathbb{APS}_L^{\text{seg}}(SEG, ch_r, j)$. In Figure 3.6, $\mathbb{APS}_L^{\text{seg}\times\text{ch}}(SEG, CH, j)$ corresponds to a vertical 'slice' of a single bin column of size $n \times x \times 1$, e.g., area (3), where it is shown as a matrix over segments and channels. Note that the order of segments and channels is arbitrary but fixed, whereas the order of bins is determined by time.

We write $\mathbb{APS}_L(SEG, CH, [s, e])$ for the row vector

- $\langle \mathbb{APS}_L^{\text{seg}\times\text{ch}}(SEG, CH, s)$,
- $\ldots$,
- $\mathbb{APS}_L^{\text{seg}\times\text{ch}}(SEG, CH, e)$,

which corresponds to the whole cube in Figure 3.6. Note that this vector is a matrix with columns corresponding to bins, i.e., time, and rows corresponding to segments and channels. This structure allows for slicing and dicing in the following ways.

- Row vectors can be used for visual analytics, as we show in Section 3.5,

- Columns vectors of various bin intervals can serve for extracting independent and dependent variables for learning a predictive ML model, as we show in Chapter 9,
- The aggregation along the bin and segment axes allows for feature space reduction.

We call $\mathbb{APS}_L(SEG, CH, [s, e])$ a *multi-channel* performance spectrum of event log L over segments SEG, channels CH, and period $[s, e]$.

Next, we discuss how both the performance spectrum and its aggregate together allow for obtaining deeper insight into the process performance than each of them separately.

3.2.4 Combined Performance Spectra

In general, the performance spectrum and its aggregate do not replace each other but complement. Let us consider a performance spectrum $\mathbb{PS}_L^{\text{lr}}(lr)$ of a layer $lr = (SEG, \mathbb{C})$, and its aggregate $\mathbb{APS}_L(SEG, \langle ch \rangle, [s, e])$ over some time interval $[s, e]$, where $ch = (\mathbb{C}, g_1, p_1)$, i.e., channel ch has the same performance classifier $\mathbb{C}$ as layer lr. For visualization purposes, each row vector $\mathbb{APS}_L^{\text{seg}\times\text{b}}(SEG, ch, [s, e])$ visualized over $\mathbb{PS}_L^{\text{lr}}(lr)$ such that

1. the segments of $\mathbb{APS}_L(SEG, \langle ch \rangle, [s, e])$ are "drawn" over the segments of $\mathbb{PS}_L^{\text{lr}}(lr)$,
2. the time axis origins are aligned, and
3. the scales of the time axes are identical.

We call this view a *combined performance spectrum*. It often allows for better information perception by the user than the "regular" or aggregate spectrum separately.

An example of a combined performance spectrum is shown in Figure 3.7(a), which shows how an aggregate performance spectrum (Figure 3.7(b)) can be visualized "over" a "regular" one. In this combined spectrum, the bars in each bin show how many segment occurrences start during the bin time interval for each performance class. For example, the three segment occurrences with performance class c^1 and one segment occurrence with performance class c^3 start within the fourth bin. However, this information does not tell, for example, when each of these occurrences ended, and whether they preserved the initial order. For that, the "regular" performance spectrum can be used. Its lines show exactly when occurrences $(a_2, b_2), (a_3, b_3), (a_4, b_4)$ and (a_5, b_5) started and ended, and if they preserved the initial order.

In the next section, we consider which performance patterns can reveal the regular, aggregate, and combined performance spectra.

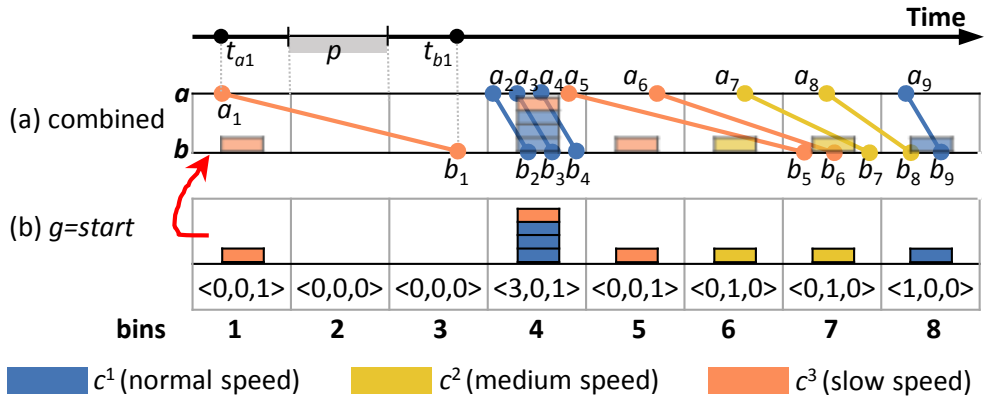


Figure 3.7: Combined performance spectrum (a), obtained by visualizing the aggregate performance spectrum for grouping *start* (b) on top of the "regular" performance spectrum.

3.3 Performance Patterns

Performance spectra of processes often contain an overwhelming amount of information and may read difficult for the untrained eye. However, processes with similar performance characteristics show similar *patterns* in their performance spectra, and vice versa, similar patterns indicate similar performance characteristics. These patterns introduce a higher abstraction level over 'plain' performance spectra, thereby aiding in the performance description and analysis. Next, we illustrate the idea of patterns in the performance spectrum, distinguishing *elementary* and *composite* patterns. We also provide the *taxonomy* of parameters of elementary patterns in Section 3.3.2. Additionally, we discuss the composite patterns in Section 3.3.3.

3.3.1 Elementary Patterns

Intuitively, a performance pattern is a specific configuration of the lines and/or bars in a performance spectrum that

- is visually distinct within a larger part of the spectrum,
- describes a particular *performance scenario* of *multiple* cases over time, and
- repeats when this scenario repeats.

An *elementary pattern* relates to a single segment and cannot be broken down further without loss of its meaning.

For example, in Figure 3.8, segment (*Insert Fine Notification*, *Add penalty*) of the *Road Traffic Fines Management* (RTFM) log[1] exhibits a pattern consisting of many parallel inclined lines of the same color, corresponding to multiple observations distributed over time. Non-crossing lines show a strict FIFO order, and identical inclinations show the constant waiting time for all the cases. Variation in the density of the lines (and in the height of the bars of the aggregate spectrum) shows continuous, varying workloads throughout the entire log. Elementary patterns with these characteristics are typical for highly standardized automated activities with strict time constraints. Note that existing models describe the performance of this segment as *constant* delay of 60 days (Figure 3.1(a)). We consider this pattern to be "elementary" in the sense that we cannot decompose it further without losing its key qualities: single segment, strict FIFO with a constant time, and workload is continuous and varying.

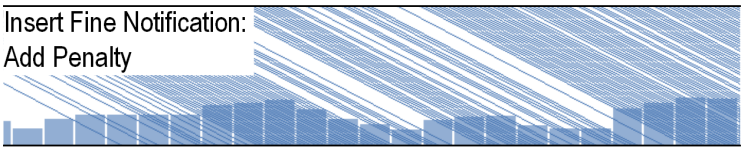


Figure 3.8: The entire performance spectrum of segment (*Insert Fine Notification*, *Add penalty*) of the road traffic fines management log exhibits an elementary pattern instance showing the FIFO behavior with a constant waiting time.

3.3.2 Taxonomy of the Parameters of Elementary Patterns

Real-life processes exhibit a great variety of elementary patterns and their combinations in the performance spectra. As a result, it is impossible to provide their comprehensive catalog. However, the comprehensive taxonomy of the *parameters* of elementary patterns can be created. The taxonomy proposed in this section allows for a complete and unambiguous high-level performance description of a process over time in a way that patterns that correspond to similar performance scenarios have identical parameters, and the identical parameters of patterns mean similar performance scenarios. At the same time, changing the value of any parameter in a pattern would mean a different performance scenario.

These pattern parameters characterize the Shape of lines and bars in a spectrum in a particular Scope over time. The line density and bar height describe Workload while their color describes Performance. Figure 3.9 shows the parameter values organized in a hierarchy, together with typical patterns having these parameter values. We

[1] https://doi.org/10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5

also provide a unique short-hand value [in brackets] for each parameter, to allow the succinct notation of patterns.

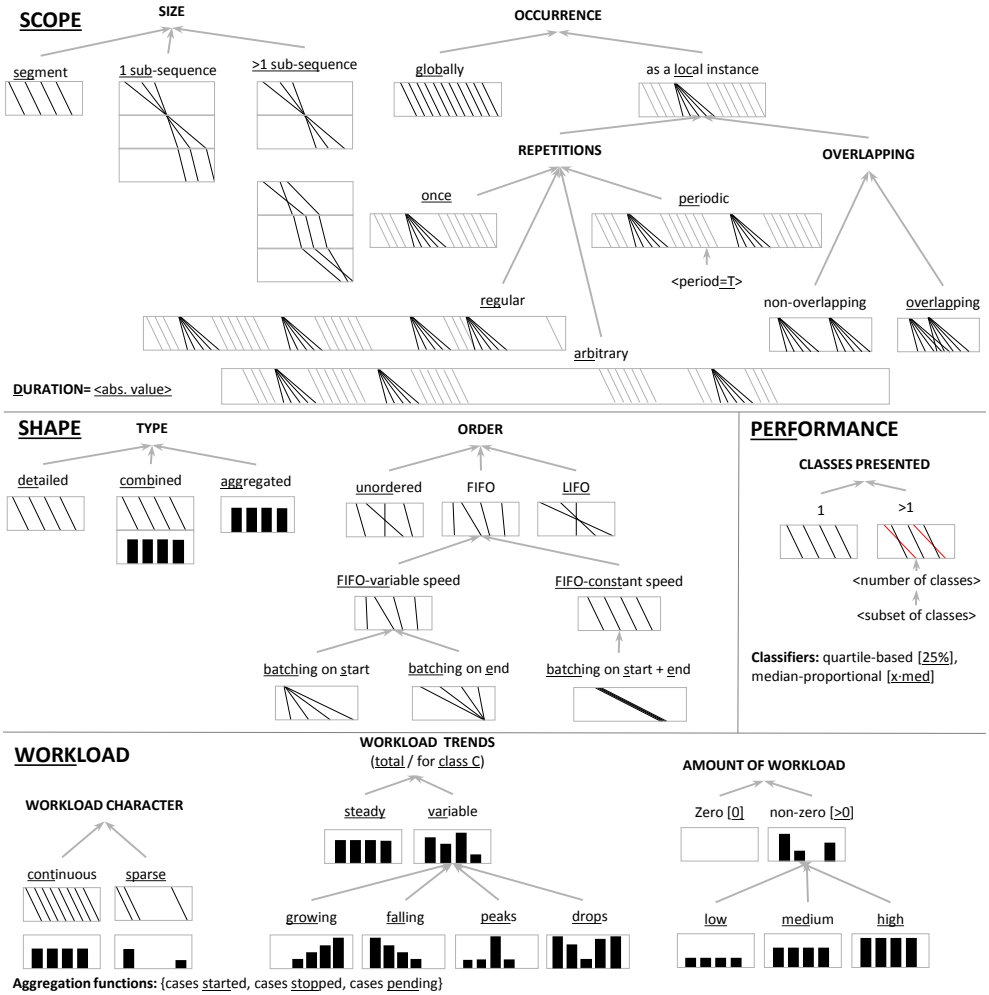


Figure 3.9: Taxonomy of the parameters of elementary patterns.

3.3.2.1 Scope Parameters

Scope parameters capture the place of pattern in the performance spectrum.

- size: one segment [1 seg], one sub-sequence [1 sub-seq], several sub-sequences [>1 sub-seq]
- occurrence: globally [glob], as a local instance [loc]

- repetitions (for patterns occurring in a local instance): once [once], regular [reg], periodic [per=T], arbitrary [arb],
- overlap (for repeating patterns): overlapping [overlap], non-overlapping
- duration: absolute value [D=T]

Parameter size describes the pattern length from the control-flow perspective:

- a single segment,
- a single sub-sequence, or
- several sub-sequences of segments.

Although all elementary patterns have size 1 seg, we include this parameter in the taxonomy for compatibility with composite patterns that we discuss in Section 3.3.3. A pattern occurrence can be either global, when it occurs continuously throughout a segment without clear boundaries, otherwise it distinctly occurs as a local instance. Pattern instances may occur once or repeat

- periodically in particular intervals T,
- regularly, i.e., seemingly systematic but not periodic, or
- arbitrarily.

Repeated pattern instances can be overlapping or non-overlapping in time. Parameter duration describes the absolute duration over time (e.g., as an interval in seconds).

3.3.2.2 Shape Parameters

Shape parameters describe the appearance of lines and bars in the performance spectrum visualization. Thus, parameter type has values

- detailed [det],
- aggregated [agg], and
- combined [comb].

Parameter order has values

- unordered [unord],
- LIFO [LIFO],
- FIFO with variable time [FIFO-var],
- FIFO with constant time [FIFO-const],
- batching on start [batch(s)],
- batching on end [batch(e)], and
- batching on start and end [batch(s+e)].

A pattern described just in terms of lines (bars) of a performance spectrum is detailed, while the pattern of an aggregate performance spectrum is aggregate. Other-

wise, it is combined. Parameter order describes the configuration of lines in a detailed pattern:

- unordered when lines irregularly cross each other,
- LIFO when lines end in reversed order of starting,
- FIFO when lines never cross.

For FIFO:

- non-crossing lines of variable inclination mean variable time [FIFO-var], where multiple lines starting (or ending) in a very short period show multiple cases batching on start (or on end), and
- lines of identical inclination show constant time [FIFO-const], where multiple lines starting and ending in a very short period (with no lines before/after) show batching on start and end.

3.3.2.3 Workload

Workload describes the height of bars in aggregate or combined patterns, and the density of lines in detailed patterns over time. Parameter aggregate function describes grouping:

- started [start],
- stopped [stop], and
- pending [pend].

Parameter workload character describes either continuous [cont] or sparse [sparse] workload. Parameter amount of workload has values:

- zero [0],
- non-zero [>0],
- low [low],
- medium [med], and
- high [high].

Finally, parameter workload trends (for a performance class or in total) describes

- steady [steady],
- variable [var],
- growing [grows], and
- falling [falls]

workload, that can additionally show peaks [peak] or drops [drop].

Workload is characterized by the aggregate function, i.e., grouping (see Definition 3.8). Workload character can be continuous or sparse (when there are longer gaps between lines or bars), and it is visible in both detailed and aggregate patterns. Amount

of workload is categorized as zero or non-zero, the latter can be categorized further as low, medium or high in relation to the maximum number of observations made on a segment (within the bin size p, see Definition 3.8). The trend over time can be steady (bars have almost the same height) or variable, the latter splits further into steadily growing or falling workload, or showing peaks (a few high bars surrounded by lower bars) or drops.

3.3.2.4 Performance

Performance is described in terms of the performance classes present in the pattern with respect to the performance classifier $\mathbb{C}$ (see Definition 3.3), chosen by the analyst. Thus, we distinguish the following values of parameter classes presented:

- 1,
- > 1,
- number of classes, and
- subset of classes.

Among classifiers, we distinguish quartile-based [25%] (e.g., all observations belonging to the 26%-50% quartile), and median-proportional [x·med] e.g., all observations 2-3 times longer than the median duration).

In the performance spectrum visualization, the classes are encoded by colors. A monochrome pattern has 1 class presented while a multi-color one has > 1 classes presented.

Now, we show how this taxonomy describes the elementary patterns E1-E3 found in the RTFM log and highlighted in Figure 3.10(left). Pattern E1 occurs in a single segment in local instances with a duration of six months, instances repeat regularly and overlap; the detailed pattern shows batching on end in a continuous workload for four performance classes in a quartile-based classifier. Using the short-hand notation, we write E1 = [Scope(seg,loc,reg,overlap,D=6mo), Shape(det,batch(e)), Work(cont), Perf(25%,4 classes)]. Similarly, we can characterize

- E2 = [Scope(seg, glob), Shape(det, FIFO-const), Work(sparse), Perf(25%, 1 class)], and
- E3 = [Scope(seg, loc, reg, overlap, D=1mo), Shape(det, batch(s)), Wo(cont), Perf(25%, 4 classes)].

Potentially, to create a catalog of elementary patterns, some additional information can be added to the pattern description. For example, a pattern's unique identifier, name, and meaning. The latter depends on the domain and/or chosen event classifier.

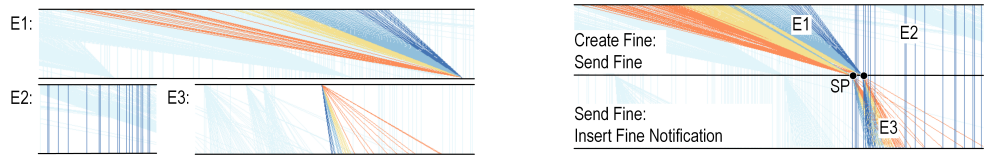


Figure 3.10: Three elementary patterns E1, E2, and E3 (left), and two occurrences of a composite pattern consisting of E1-E3 (right).

3.3.3 Composite Patterns

In the previous sections, we described the performance of a single segment through elementary patterns. However, the performance spectrum of real-life processes gives rise to *composite* patterns comprising several elementary patterns. While a full taxonomy is beyond the scope of this thesis, we outline some basic principles for describing composite patterns by relating elementary patterns to each other in their context.

The context of some pattern P1, as shown in Figure 3.11(a), consists of

1. observations earlier and later than P1 in the same process segment,
2. observations before and after P1 in the control flow perspective, and
3. a distinct pattern P2 occurring simultaneously to P1 in the same segment.

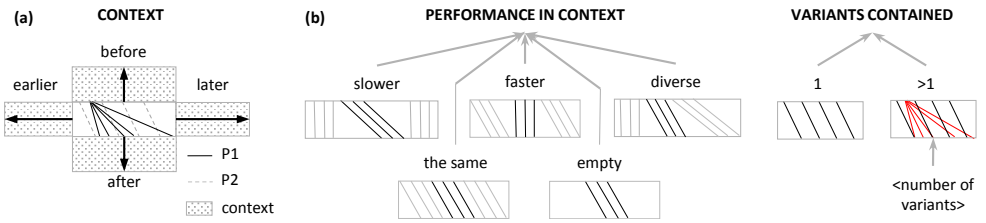


Figure 3.11: Pattern context (a), and context parameters (b).

Using this context, the taxonomy in Figure 3.9 can be extended. For instance, the observations before and after can be used to characterize the performance of a pattern in context, and the performance variants contained in the same time period, as shown in Figure 3.11(b).

For example, Figure 3.10(right) shows two instances of a composite pattern consisting of the elementary patterns E1, E2, and E3, described in Section 3.3.2. E1 and E3 align at a synchronization point SP, which shows synchronization of multiple cases in a “hourglass” pattern, while the cases in E2 do not synchronize with the cases in E1 or E3: we can clearly see 2 variants of behaviors contained (E1+E3 and E2). The performance context of the composite pattern is diverse.

The parameters of the taxonomy in Figure 3.9 and the new parameters in Figure 3.11 can only partially describe composite patterns. In particular, a comprehen-

sive taxonomy for a precise description of the alignment of patterns to each other in their context is the subject of future work.

3.3.4 Demonstration

For work with performance spectra, we implemented the *Performance Spectrum Miner* (PSM) — the tool for the transformation of logs into performance spectra and their visualization in an interactive ProM[2] plug-in in package "Performance Spectrum" [46][3]. Then, we conducted an exploratory analysis of several event logs of processes of different domains. In the following, we introduce the PSM, and summarize findings obtained from the event log of a BHS, provided by Vanderlande, and from the publicly available log of the Road Traffic Fine Management process. We also compared the spectra of 11 real-life event logs from business processes (BPI12, BPI14, BPI15(1-5), BPI17, Hospital Billing, RTFM)[4].

3.3.4.1 The Performance Spectrum Miner

The PSM is a tool implemented in Scala for work with performance spectra. It can be used as part of the ProM or as a stand-alone tool. The former is convenient if the other ProM plugins are in the analysis pipeline, while the stand-alone mode allows running the PSM on non-Windows computers.

Figure 3.12(a) shows how to engage the PSM in the ProM. In this figure, an event log is chosen as the PSM input in panel (1), the PSM is selected in the plugin list (2), and the output (i.e., the performance spectrum) is shown in panel (3).

After pushing "Start", the pre-processing configuration window appears (see Figure 3.12(b)). The user can specify

- the bin size for computing the aggregate performance spectrum,
- the custom implementation of the performance classifier (if needed),
- the quartile-based or median-proportional performance classifier,
- the custom activity classifier (if needed), and
- the folder for saving the performance spectrum files.

After the pre-processing is done and the performance spectrum is saved on disk, the user can optionally aggregate activities (see Figure 3.12(c)), and choose to load the whole spectrum in memory, or read it from disk on demand (for working with large datasets).

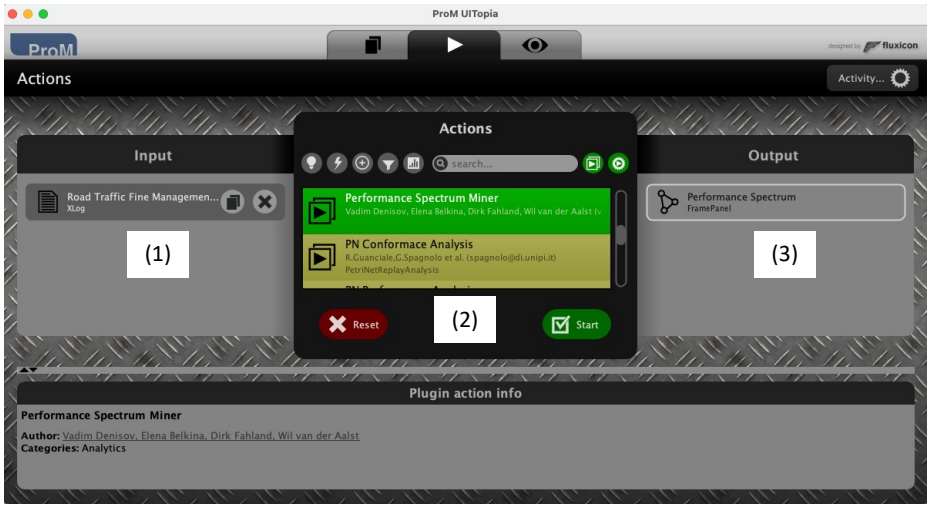
The main PSM panel with the performance spectrum is shown in Figure 3.13(a). In the ProM panel (1), the PSM panel is rendered. For each performance spectrum

[2]Process Mining Framework `https://pa.win.tue.nl/prom/`

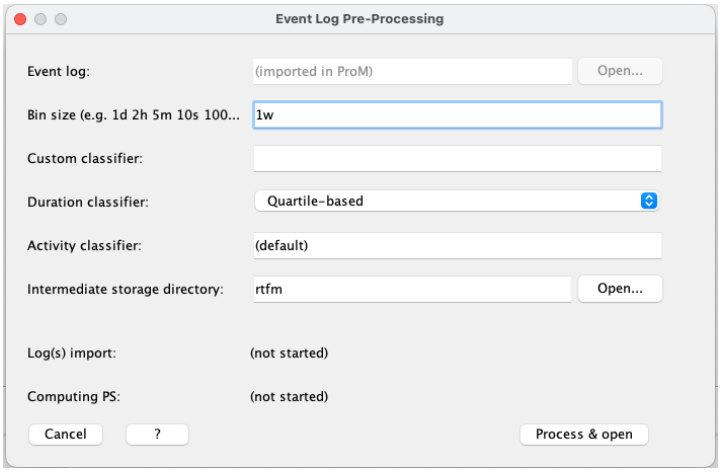
[3]source code and documentation are available at `https://github.com/processmining-in-logistics/psm`

[4]available at `https://data.4tu.nl/repository/collection:event_logs_real`

(a)



(b)



(c)

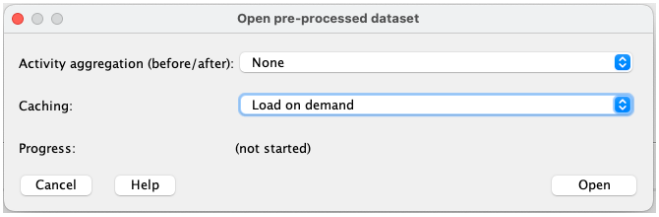


Figure 3.12: The initial panel of the ProM for starting the PSM (a), pre-processing configuration (b), and caching mode (c).

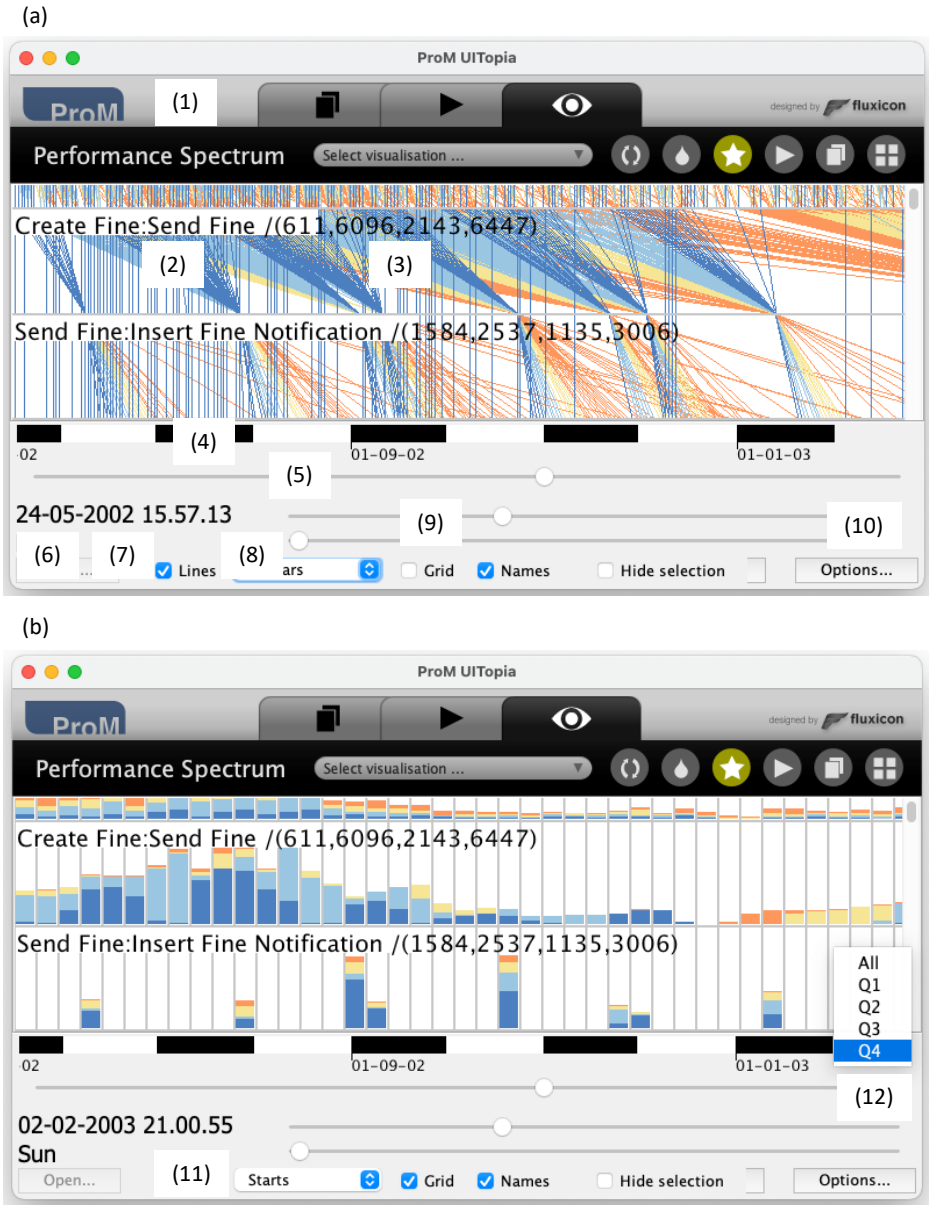


Figure 3.13: The PSM shows the "regular" performance spectrum (a), and the aggregate performance spectrum (b).

segment, the segment labels (2) and basic statistics (3) are shown. The grid (4) shows the absolute time, the scroller (5) allows navigation, and the label (6) shows the ex-

act time corresponding to the mouse pointer position in the spectrum. Checkbox (7) shows/hides the spectrum occurrences (lines), and list (8) controls the grouping of the aggregate spectrum (if shown). The scroller (9) allows zooming, and button Options (10) shows an additional window. Figure 3.13(b) shows the aggregate performance spectrum with grouping *start*. Note, the combined spectrum is also supported in the PSM. The menu (12) allows to show/hide occurrences and/or bars of particular performance classes.

The options shown in Figure 3.14 allow the user

1. to use regular expressions for filtering in and out segments,
2. to keep only particular cases in the spectrum by providing the set of their identifiers, and
3. to filter segment based on the average throughput.

For example, expression `(Insert.*:.*)|(.*:Appeal.*)` keeps only segments whose first activity label starts with *Insert* or whose second activity label starts with *Appeal*.

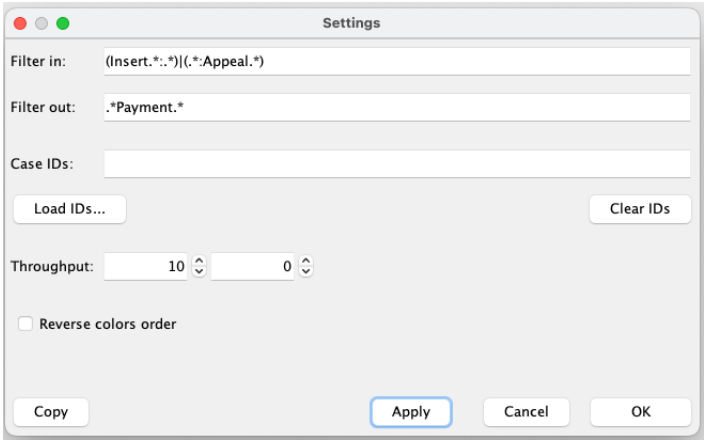


Figure 3.14: THe PSM additional options for segment and case level filtering.

Next, we demonstrate our findings obtained with the PSM.

3.3.4.2 Baggage Handling System of a Major European Airport

Event Logs. In this case study, we analyzed flows of bags throughout a Vanderlande-built BHS. In the given event log, each case corresponded to one bag, events were recorded when bags passed sensors on conveyors, and activity names described the locations of sensors in the system. For one day of operations, an event table contained on average 850 activities, 25.000-50.000 cases (for a case notion *bag identifier*), and 1-2 million events.

To provide examples of the BHS performance spectrum and patterns, we selected conveyor sub-sequence ⟨*a1*,*a2*,*a3*,*a4*,*a5*,*s*⟩ that forms a path from a check-in counter *a1* to a main sorter entry point *s*. At location *a1*, passengers put bags onto the belt of the check-in counter. We chose this particular part because (1) most BHSs have at least one path from check-in counters to a sorting area, and (2) it shows many performance patterns typical for BHSs. The diagram of the corresponding system part in Figure 3.15 shows that other bags can join from other check-in counters on the way at locations *a2-5*. We first discuss elementary detailed patterns in the performance spectrum, and then show how their compositions explained the complicated system behavior.

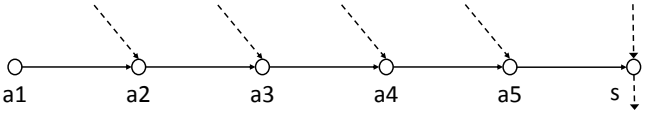


Figure 3.15: Path from check-in counter *a1* to sorter entry point *s*.

The performance spectrum in Figure 3.16 shows events over the period of one hour, using a median-proportional performance classifier. In the first segment S1 *a1*:*a2* we can observe pattern P1 (FIFO, constant waiting time, variable workload, normal performance) and P2 (batching on start and end with very slow performance). Empty zone Z1 shows zero workload. In BHSs, FIFO behavior is typical for conveyors where bags cannot overtake each other, and variable workload is typical for manual operations: the check-in counter arrival process depends on the passenger flow and the service time, which varies from passenger to passenger. Although conveyor speed is constant, segment S1 shows not only pattern P1 but patterns P2 and Z1 as well: some conveyors were temporarily stopped, so the bags whose occurrences comprised P2 experienced the same delay.

By looking at S1 alone, we cannot explain the causes of the delays in those pattern instances. But as segments in a BHS are synchronized through the movement of physical objects on conveyors, we can identify what caused them by following the control flow shown in Figure 3.15. After P4 in S2, we observe Z1 in S3 and S4, which contains non-zero workload earlier (P3) and later (P3, P6), followed by non-zero workload P5 in S5 (FIFO, constant waiting time, high workload, normal performance). This gives rise to pattern L, comprising P2 or Z1 on S1, P4 on S2, and Z1 on S3 and S4, and its context is highlighted in Figure 3.16.

Reading pattern L from S4 backward gives the following interpretation: the conveyors in S3 and/or S4 stopped operations, so bags from S2 could not move further to S3. When S2 was stopped, S1 also was stopped (point Y), because bags could not enter S2. The slow cases of P2 and P4 are the bags waiting on the stopped conveyors. This is called a *die-back* scenario, where delays or non-operation (in S3, S4)

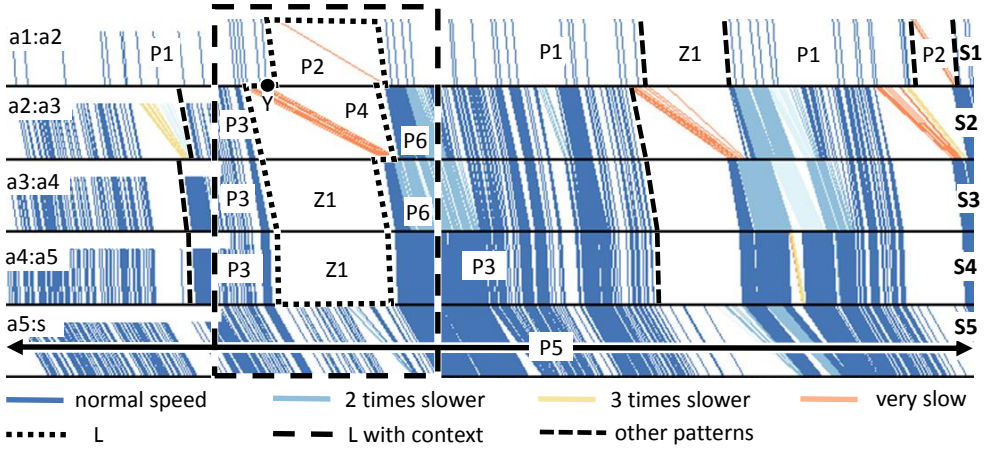


Figure 3.16: Performance spectrum of cases (bags) moving from the check-in counters toward the sorter *s*.

propagated backward in the control-flow direction. When S3 and S4 returned to operations, the waiting bags on S1 and S2 (and from other parts that are not included in Figure 3.16) resumed their movement. The two times slower performance in P6 shows that S2 and S3 were at their capacity limits in this restart phase until all workload decreased. Figure 3.16 also shows that pattern L repeats regularly during the day.

Next, we consider an event log from a non-MHS domain.

3.3.4.3 Road Traffic Fine Management Process

Event Logs. The RFTM event log consists of 11 activities, more than 150.000 cases, and 550.000 events over a period of 12 years. We analyzed the trace variants R1-R3 of Figure 3.17, which cover more than 80% of the events in the log, by defining a layer with a segment series

- ⟨*Create Fine*,*Payment*,*Create Fine*,*Send Fine*,
- *Insert Fine Notif.*,*Add penalty*,
- *Payment*,*Add penalty*,*Send for CC*⟩,

and quartile-based performance classes.

First, we discuss the detailed patterns P1-P5 that can be observed in the performance spectrum of a 2-year period in Figure 3.17, which represents the behavior typical for the entire 12-years period. All cases start from activity *Create Fine* and continue either with activity *Payment* (variant R1) or activity *Send Fine* (R2 and R3).

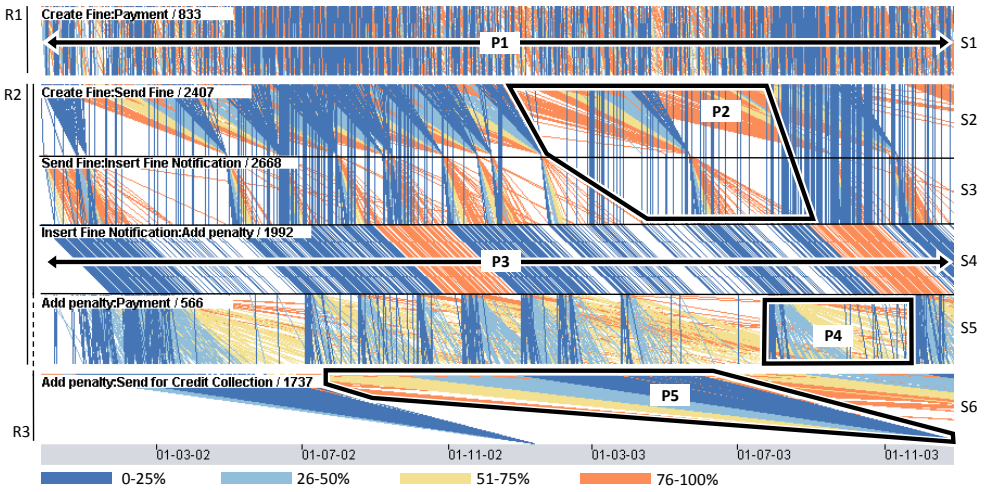


Figure 3.17: Performance spectrum of the RTFM process for years 2002 and 2003 for trace variant R1-R3.

Pattern P1. Segment S1 *Create Fine*:*Payment* globally contains many traces of variable duration, which are continuously distributed over time and can overtake each other, i.e., P1 = [Scope(seg,glob), Shape(det,unord), Work(cont), Perf(25%,4 classes)]. We can clearly observe that the traffic offenders pay at various speeds.

Pattern P2. The performance spectrum of Figure 3.17 shows that the sub-trace ⟨ *Create Fine*, *Send Fine Insert Fine notification*⟩ shared by R2 and R3 contains the composite pattern P2 which we already discussed in Section 3.3.3. P2 consists of two *different* performance variants. The "hourglass" pattern of E1+E3 of Section 3.3.1 (see Figure 3.10(right)) shows that cases are accumulated over a period of six months. The period until *Insert fine notification* varies from zero up to four months. Cases in pattern E2 of Section 3.3.1 are not synchronized but processed instantly.

Pattern P3. The two variants E1+E3 and E2 vanish in the next segment S3 *Insert Fine Notification*:*Add penalty* where all cases show a strong FIFO behavior: P3 = [Scope(seg,glob), Shape(det,FIFO-const), Work(cont), Perf(25%,2 classes)]. The switch from CEST to CET in October is shown as a slower performance class in Figure 3.17. After *Add penalty*, R2 continues with *Payment* (S5 in Figure 3.17) and R3 continues with *Send for Credit Collection* (S6 in Figure 3.17).

Pattern P4. In segment S5 *Add penalty*:*Payment* we unexpectedly observe batching on start despite the absence of batching on end in the preceding segment S4. It happened due to the "hourglass" batching in P2 resulting in dense groups of "fast" occurrences that were "forwarded" (pattern P3 in S4) together and created batching on start (pattern P4 in S5), which can take months to years to complete the *Payment*.

Pattern P5. The alternative segment S6 *Add penalty*:*Send for Credit Collection* shows batching on end every 12 months for cases that entered the batch 20 to six months prior: P5 = [Scope(seg, loc, per= 12mo, D=20mo), Shape(det,batch(e)), Work(cont), Perf(25%, 4 classes)]. The six-month delay revealed by P5 is mandated by Italian law.

A unique pattern for this process occurred in segment *Add Penalty*:*Send Appeal to Prefecture* shown in Figure 3.18(b) where a batch on end occurred only once with a duration of 10 years.

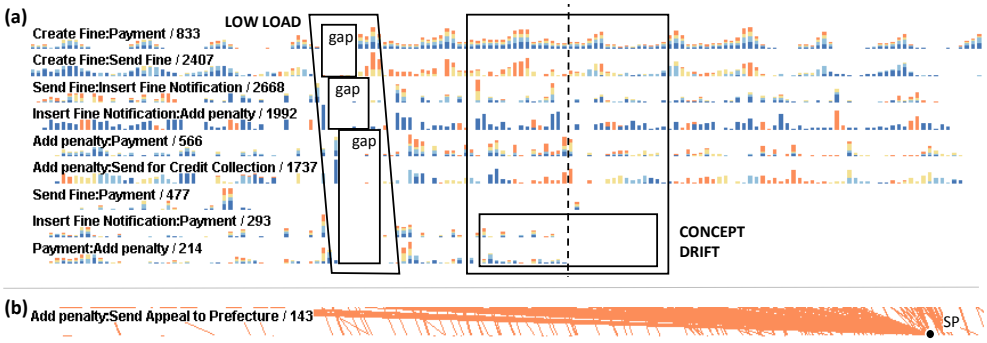


Figure 3.18: Aggregate performance spectrum of the road traffic fines management log (2000-2012)

Aggregate patterns. The aggregate patterns are shown in Figure 3.18(a), where every bar shows how many segments started every month. Here we can see patterns related to workload. For example, in the first quarter of 2004 we can see zero workload (named "gap") for three months, gap=[Scope(seg, loc, once, D=3mo),Shape(agg, batch(e)), Work(0)]. It propagated to subsequent segments, creating a composite pattern surrounded by context with much higher load. Figure 3.18(a) also reveals concept drift: the medium non-zero workload in segments *Insert Fine Notification*:*Payment* and *Payment*:*Add penalty* drops to 0 in 2007.

3.3.4.4 Comparison of Event Logs

We compared the 11 real-life business process event logs regarding the types of performance patterns they contain. We visualized the performance spectrum of each log and noted the properties of the immediately visible patterns in the terms of the taxonomy in Section 3.3.2. Table 3.1 shows the result.

Thus, we identified

- the combined patterns of unordered behavior with low and high workload,

Table 3.1: Presence of the selected pattern classes in the real-life event logs.

	BPI12	BPI14	BPI15-1	BPI15-2	BPI15-3	BPI15-4	BPI15-5	BPI17	Hospital	H-Billing	Road Fine
unord,low		glob	glob	glob	glob	glob	gob		glob	glob	glob
unord,high								glob		glob	glob
FIFO								glob		glob	glob
FIFO+unord									reg	glob	
FIFO (weekly)	glob	glob			arb			glob			
batching		arb						per		per	reg
workload spikes								arb			reg
concept drift		once	once	once	arb	arb	arb			once	reg
sparse work	reg	reg	glob*	glob*	glob*	glob*	glob*		glob	glob	

- the detailed patterns of the FIFO behavior, also overlaid with an unordered variant, FIFO+unord, and occurring only Mon-Sat, FIFO(weekly), and various forms of batching, and
- the aggregate patterns showing workload spikes, concept drift, and sparse work.

The cells in Table 3.1 indicate for each log the occurrence and repetition values of these patterns according to the taxonomy of Figure 3.9. The logs differ strongly in the presence and repetition of patterns, indicating that very different performance scenarios occurred in these processes. Interestingly, the BPI15 logs, which all relate to the same kind of process that is being executed in different organizations, show very similar patterns: glob* for sparse work means that sparse work *co-occurred* globally in a synchronized way, i.e., a large number of segments showed this behavior during exactly the same days.

3.3.5 Performance Spectrum Replication Study

The previous section showed how the performance spectrum reveals performance patterns in various types of processes. However, that demonstration cannot be considered as an evaluation of the technique, because it has been done by the authors, and therefore is biased. In contrast, a replication study of performance patterns in different processes, presented in [56], has been done by analysts previously unaware of the performance spectrum. In the following, we discuss the objectives, setup, and results of this empirical exploration and our conclusion about its results.

Objectives. In [45], we described the performance patterns, identified by the authors in various segments of different event logs. The replication study was aimed to prove that these results can be reproduced by untrained analysts, using [45] as a guideline, and the Performance Spectrum Miner [46] as a software tool for work with performance spectra. The following exploitative questions were aimed to be answered.

1. **ExQ-1.** Can the performance patterns, provided in the taxonomy [45] (Figure 3.9), be identified in the performance spectra of various processes?

2. **ExQ-2.** Can the analyst identify the same performance patterns as the authors in [45]?

Setup. Six participants were asked to reproduce all the results provided in Section 5 (Evaluation) of [45]. For that, the following scenario was designed:

1. Each participant reads the paper [45].
2. All the participants discuss it together.
3. Each participant gets familiar with the technique *independently* by using the PSM.
4. Finally, the paper results have to be reproduced:
   - for the *Road Traffic Fines Management* (RF) log[5] in details,
   - and for the other logs (Table 3.1) in a form of a summary.
5. Each participant describes if the results could be reproduced, and what difficulties were encountered.

Results. In this section, we summarize the study results [56] (pages 39-138) and answer **ExQ-1** and **ExQ-2**.

ExQ-1. All the participants could successfully obtain the performance spectra from the given event logs using the PSM, and easily identify a lot of various performance pattern instances, according to the taxonomy. However, the following difficulties were reported as well:

- *Ambiguity.* There were some "border cases" found in the obtained performance spectra, i.e., pattern instances that could be related to two patterns of the taxonomy. Moreover, some instances could be identified either as elementary ones or as part of a composite pattern instance.
- *Biased human interpretation.* Some participants could identify only one pattern instance in a particular area of a performance spectrum, while others could identify there the instances of alternative patterns as well.
- *Vague taxonomy.* Some reports claimed the taxonomy was not precise or detailed enough, so it was sometimes difficult to describe an identified pattern instance in the terms of the taxonomy parameters.

ExQ-2. The participants could successfully re-identify approximately half of the same pattern instances as the authors. Difficulties with the re-identification of the others were mostly caused by the following reasons:

- *Missing information about logs pre-processing.* The given event logs are large but the authors of [45] did not describe how the data were pre-processed for ob-

[5] `https://doi.org/10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5`

taining those concrete results (spectra), and no case identifiers or time intervals were provided.
- *Missing event logs.* The BHS dataset from a Vanderlande BHS was not available due to privacy reasons.

Additionally, the participants reported the good quality of the tool (the PSM) and its user documentation.

Conclusion. As a result of the participants' reports analysis [56], we conclude that the technique *generally works* because of the following.

- Pattern instances could be identified according to the taxonomy by participants without any skills in working with performance spectra.
- The patterns presented in [45] could be re-identified if the participant was able to find the corresponding area in the performance spectrum.
- Although the taxonomy of the performance patterns can be improved further, especially for composite patterns, its current version still allowed many findings and, therefore can be used for the performance patterns-based analysis of processes.
- The implementation in the PSM is good enough, i.e., it was never a bottleneck for the analysis.

To overcome difficulties that the participants encountered during the study, we recommend the use of the PSM documentation along with [46]. Additionally, automatic performance pattern detection algorithms, such as the one addressed in [41], can help avoid the time costs and ambiguity of manual detection.

In the next section, we show how "regular" and multi-channel performance spectra together allow for performance analysis of large material handling processes.

3.4 Practical Aspects of MHS Analysis Using Performance Spectra

So far, we proposed the performance spectrum as a tool for process performance description and analysis. However, we did not consider the following important aspects of its application:

- whether an event log, required for computing a performance spectrum, is a suitable event data representation for MHS data, and
- if the root cause analysis of MHSs using performance spectra is possible.

We discuss these aspects in the following.

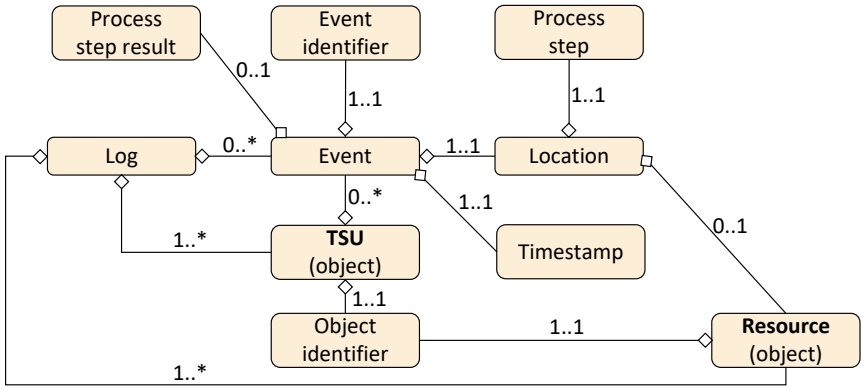


Figure 3.19: UML class diagram for MHS event log.

3.4.1 Object-Centric Event Logs of Material Handling Processes

Historically, most process mining techniques require a single-case notion event log, where each event is related to one "object" of the type corresponding to the case notion (see Definition 2.11). For example, in the RTFM event log, the case notion is a fine ticket. However, in complex business processes, many objects of different types can be related to an event. For example, for the delivery process of a web shop, one package is related to one or multiple items, while one order can be delivered via multiple packages. Such event data can be represented as *object-centric* event logs, e.g., using the OCEL standard [64]. A multi-case notion event log must be "flattened" to obtain a single-case notion event log for a chosen case notion.

The performance spectrum (see Definition 3.4) is computed from a single-notion event log, i.e., it requires the event data to be flattened if they represent multiple case notions. However, data flattening can cause well-known problems of

- *convergence* when one event is related to different cases [73], and
- *divergence* when there are multiple instances of the same activity within a case.

Is this problem actual for event data of material handling processes and, consequently, the use of performance spectra? To understand it, we customized the UML class diagram of the OCEL standard [64] for MHSs. The result is shown in Figure 3.19.

In the diagram, two *object* types, which an MHS *event log* usually has, are depicted: *TSU* and *resource*. For clarity, these objects are shown through separate boxes. In MHSs, the handling of TSUs generates events. Respectively, TSU has zero or more *events* in the diagram. Each event is generated at precisely one *location* at the time described by a *timestamp*. A TSU cannot be at more than one location at the same time, so the cardinality on the arc between the event and location boxes is 1..1. Further, each location corresponds to exactly one *process step* (1..1 on the corresponding

arc). A location has zero or one *resource* (0..1), where zero corresponds to the situation when the resource (e.g., a worker) is unavailable. Finally, both TSU and resource have an *identifier*. Additional attributes can be related to an event, e.g., a *process step result*.

To summarize,

- an event is always associated with one location, one process step, and one resource, and
- a resource is related to at most one event at a time.

Thus, no *one-to-many* or *many-to-many* relations exist for event logs corresponding to the diagram in Figure 3.19. As a result, no flattening is actually required, so no convergence or divergence problems exist.

3.4.2 Causality in Performance Spectra of MHSs

In Section 3.3.4, we showed how performance spectra could reveal various performance phenomena. Their interpretation depends on the process. However, in the MHS domain, performance spectra show the physical movement of materials in three-dimensional space. It allows for reasoning about *causality* among phenomena in performance spectra due to constraints related to MHS equipment and TSU movement.

Let us consider a BHS fragment, shown in Figure 3.20(a). In this system, conveyors (a,b) and (b,c) comprise the main conveyor, while (d,b) merges at location b. Note the bags of (a,b) have a higher priority at merge unit b than bags on (d,b).

Let us consider the following scenario. Initially, bags from locations a and d were evenly merged at b. Figure 3.20(b) shows the corresponding performance spectrum, where green and blue segment occurrences correspond to the bags that entered the system at a and d, respectively. However, at time t_1, the series of bags $pid1 - pid5$ entered the system via a. By estimating the duration between the bags at a, e.g., $t_2 - t_1$, we can conclude that these bags were so densely placed on the conveyor that no other bags could be merged in between. As a result, bag $pid6$, which arrived at b at t_3, had to wait to merge until it became free at t_4 because there was no alternative way toward c available.

The analyst, given this performance spectrum and knowing the minimum distance between two neighboring bags such that another bag can be merged in between, can conclude that bags $pid1 - pid5$ *caused* the delay of $pid6$. That is, it was not a coincidence or correlation between higher load on (a,b) and longer segment occurrence durations on (d,b).

Another scenario is shown in Figure 3.21(a), where $pid1$ got stuck on conveyor (b,c). As a result, the other bags could not continue toward c because the only way was blocked. In the corresponding performance spectrum, we can see that the delay on (b,c) indeed *caused* the delays in (a,b) and (d,b).

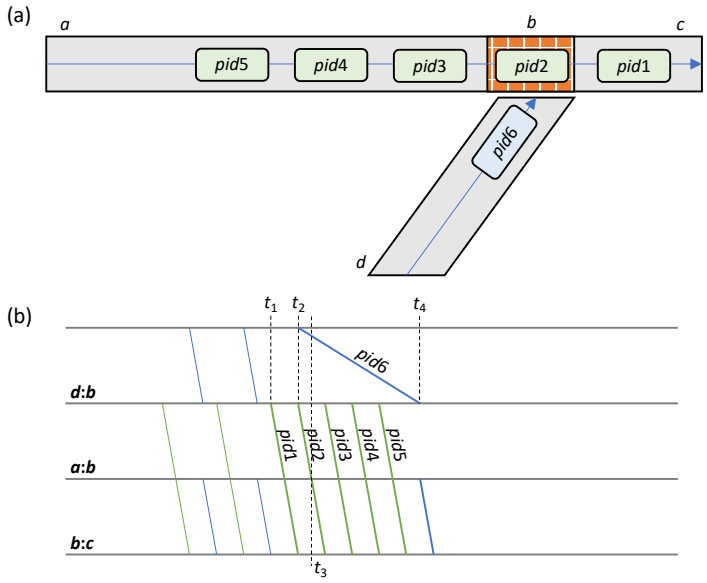


Figure 3.20: BHS fragment (a) and the corresponding performance spectrum (b) show how *pid*6 could not merge on (d, b) because of densely located *pid*1 – *pid*5 on the conveyor with a higher merge priority.

Note domain knowledge about the system or process is required to reason about causality in the performance spectra. For example, let a process model, shown in Figure 3.21(c), be the only information about some process where the activities follow each other in the same way as for the system in Figure 3.20(a). In this case, the spectra in Figure 3.20(b) and Figure 3.21(b) do *not* allow for reasoning about any causalities. For example, assuming that the model in Figure 3.21(c) represents some customer support process whose activities $a - d$ are executed by different pools of resources, no direct root causes of the delays in the spectra can be inferred.

In the following, we use the aforementioned aspects to analyze material handling processes.

3.5 Evaluation

The previous sections show that the performance spectrum can describe the process performance and reveal various performance pattern instances, which analysts can identify. However, can this technique be valuable for solving real-world problems of the process performance analysis, and which types of performance spectra, meaning a layer performance spectrum or multi-channel one, can be helpful for that? To an-

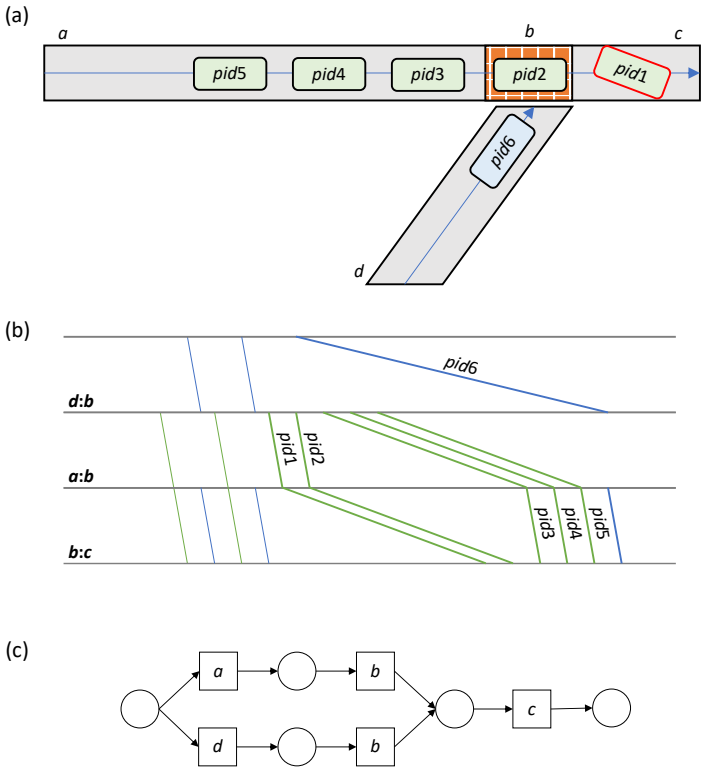


Figure 3.21: BHS fragment (a) and the corresponding performance spectrum (b) show how *pid*2 – *pid*6 were blocked by *pid*1; process model (c) describes a business process with activities $a - d$ that follow each other in the same way as in the system (a).

swer these questions, an evaluation in the form of a process mining challenge was conducted at Vanderlande.

Objectives. The evaluation aimed to answer the following questions:

- **EvQ-1.** Can performance spectra be used for the performance analysis of large material handling processes?
- **EvQ-2.** What types of performance spectra are needed for this analysis?
- **EvQ-3.** Can the performance spectrum-based analysis outperform the performance analysis based on other techniques and tools, typically used in the industry?
- **EvQ-4.** Are there any pitfalls needed to be addressed to facilitate the analysis?
- **EvQ-5.** Can the analysis results be delivered for an untrained audience in the form of performance spectrum snapshots?

Problem. The challenge was formulated by Vanderlande as follows. There was a BHS operating in a large European airport, whose high-level description was similar to the BHS shown in Figure 1.1 (see Section 1.1.2). Once, a severe performance incident happened unexpectedly, causing a long delay in handling many passenger bags. As a result, many bags could not make it to the flight. So, given

1. the recorded event data,
2. basic domain knowledge about the system, including its MFD, and
3. using any tools,

the incident had to be *detected*, and its *root cause(s)* had to be *determined*.

3.5.1 Analysis

Input Data and Data Pre-Processing. The input dataset was given in the form of a time-monotonic event log $L_1 = (\text{pid}, E_1, \#^1)$ (see Definition 2.11), recorded as a Comma-Separated Values (CSV) file. In this table, each event was generated by the BHS equipment, such as sensors, scanners, etc., and contained the following attributes that we considered as *mandatory* for this analysis:

- activity name (equipment location) act,
- bag unique identifier pid,
- timestamp time.

The traces in L showed the paths of the bags throughout the BHS. Additionally, the events had the following optional attributes:

- failed direction errors *fd*, indicating issues with bag diverting,
- bag tasks *bt*, describing the current bag task (sub-process),
- lost-in-tracking *lt*, showing that the current bag had to be re-identified,
- and dozens of others.

The first phase of the challenge was detecting the incident, i.e., *where* in the system and during *which period(s) of time* the incident happened. To determine that, the high-level overview of the baggage's progress throughout the system over time could be helpful, so we made the following preliminary steps.

1. *High-level MFD.* We designed a high-level MFD, shown in Figure 3.22, using the available low-level MFD, which was similar to one in Figure 1.1 but much larger and with more locations. In this high-level MFD, numerous baggage flows are aggregated into a single flow going from the check-in area to two pre-sorters P^1 and P^2, and afterward toward two final sorters Y^1 and Y^2. A small fraction of the baggage can return *back* to the pre-sorters because not all bags can make it to the flight, i.e., such bags must start their handling over on P^1 or P^2.

2. *Data pre-processing.* To align the event locations with the high-level MFD, we *replaced* the exact bag locations in the event activity labels with the names of the corresponding system areas. We kept the earliest timestamps for sequences of events being aggregated into a single one, dropping all optional attributes. The resulting event log $L_2 = (\text{pid}, E_2, \#^2)$ contained aggregate events E_2, and only mandatory event attributes.

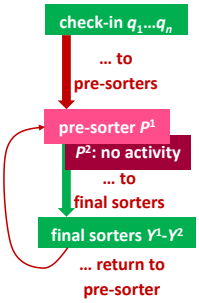


Figure 3.22: High-level MFD of the BHS, obtained by aggregating exact system locations into the names of their areas.

Combined Performance Spectrum of the Overall System Performance. In this step, we computed the combined performance spectrum using L_2. For that, we first defined:

1. a segment series $SEG^2 = \langle seg^2_1, \ldots, seg^2_{12} \rangle$, whose names corresponded to the high-level MFD segments (see these names on the left-hand side of the Figure 3.24),
2. a performance classifier $\mathbb{C}_{dur}$, which mapped the durations of segment occurrences into performance classes $\{normal, slow\}$, encoded in Figure 3.24 by the green and red colors respectively.
3. a bin size $p = 15$ minutes, corresponding to a single bar of the aggregate performance spectrum in Figure 3.24,
4. a channel $ch_{dur} = (\mathbb{C}_{dur}, pending, p)$,
5. a bin interval $[s, e]$, corresponding to the period of 28 hours, for which the events were provided in the input dataset.

Then, we computed

1. a performance spectrum $\mathbb{PS}^{\text{lr}}_{L_2}(SEG^2, \mathbb{C}_{dur})$ (not shown in the figures),
2. its aggregate $\mathbb{APS}^{\text{seg}\times\text{b}}_{L_2}(SEG^2, ch_{dur}, [s, e])$ shown in Figure 3.24, and
3. their combined performance spectrum, shown in Figure 3.23.

We also show the aggregate spectrum in Figure 3.24. It helps estimate the load on the segments because the bins do not read well on the combined spectrum when printed.

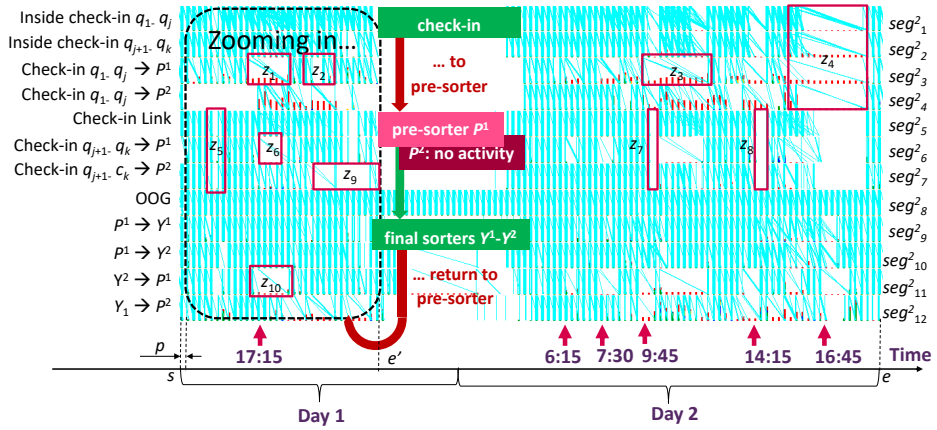


Figure 3.23: Combined performance spectrum, computed from the aggregate log L_2, shows many periods of zero or low load at different system areas, for example, elementary pattern instances $z_1 - z_3, z_6, z_9, z_{10}$ and combined ones z_4, z_5, z_7 and z_8.

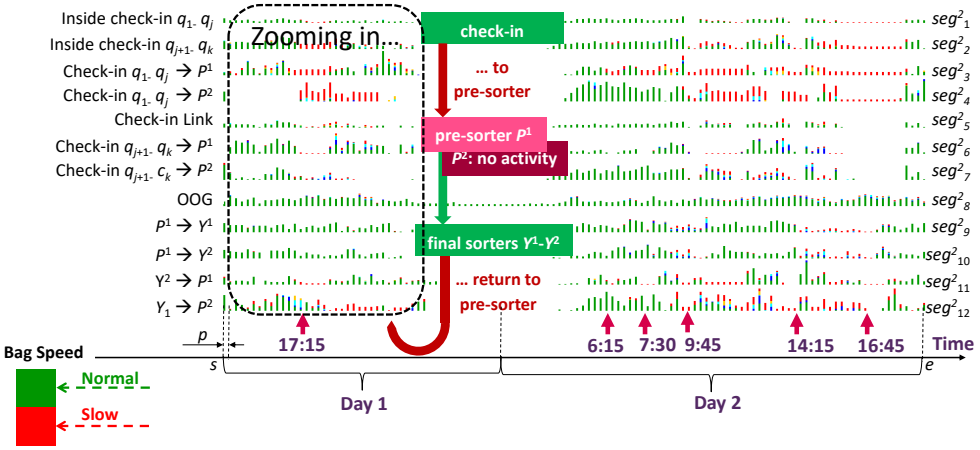


Figure 3.24: Aggregate performance spectrum of the combined one shown in Figure 3.23.

Using these spectra, we could easily detect periods $z_1 - z_{10}$ of zero or low workload in multiple areas, surrounded by context with an average load. Their durations varied from 30 to 90 minutes. In the combined performance spectrum (Figure 3.23), we saw

individual segment occurrences whose durations were up to one or two magnitudes longer than the durations of segment occurrences in the context around. These periods were observed during both Day 1 and Day 2 (as 28 hours of the event data were recorded over two working days). In this section, we provide the analysis of Day 1 only, as the analysis of Day 2 was done in a similar way and showed similar results. For that, we "zoomed in" the spectrum in Figure 3.23 to explore the performance for a shorter period $[s, e']$, where s and e' correspond to the beginning and end of Day 1 respectively (see the area on the left-hand side of Figure 3.23).

Unwinding the Incident Scenario. In the following, we discuss the analysis of the combined performance spectrum for time interval $[s, e']$, built from the initial event log L_1, i.e., for non-aggregate activities forming segments SEG^1. For obtaining aggregate spectra, we used the same value of p, and the same grouping *pending*, but investigated also additional layers and channels.

For presentation purposes, we gradually add the performance spectrum segments of different channels/layers on the same canvas. First, we put on top of the canvas the performance spectrum of segment seg_1^1 corresponding to pre-sorter P^1, for which we observed an interval INT_1 of longer segment occurrences indicating the time interval of a possible incident. This canvas is shown in Figure 3.25, where the numbers of the spectrum segment lines (from 1 through 10) are shown on the left-hand side, seg_1^1 is shown on top, and the time axis is shown at the bottom.

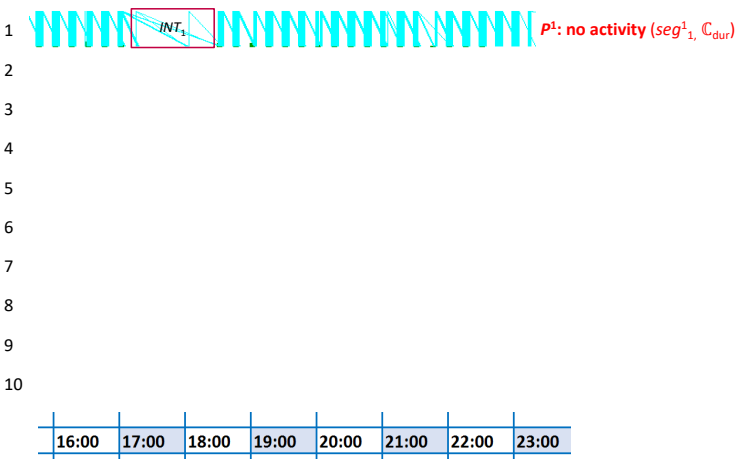


Figure 3.25: Longer segment occurrences on pre-sorter P^1 show its inactivity during interval INT_1.

We assumed that interval INT_1 most probably corresponded to a period when pre-sorter P^1 was presumably stopped. In BHSs, if one sorting loop is unavailable, the other parallel available loop(s) (if any are presented) handle the whole load. In our

case, it is sorting loop P^2, so we started by exploring its performance around the same time interval. For that, we defined:

- a *domain-specific* performance classifier $\mathbb{C}_{re}$ that showed how many "rounds" a bag made on pre-sorter P^2,
- a layer $lr_{re} = (SEG^1, \mathbb{C}_{re})$, and
- a channel $ch_{re} = (\mathbb{C}_{re}, pending, p)$.

Then, we computed the performance spectrum for this layer and channel. Segment seg_2^1 of this spectrum is shown in line 2 of the canvas in Figure 3.26.

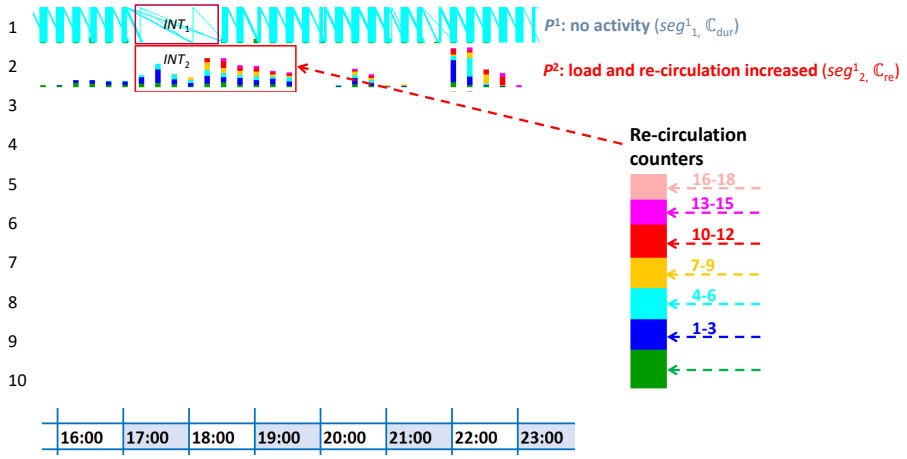


Figure 3.26: Recirculation of bags on pre-sorter P^2 grows during the first half of interval INT_2, and decreases during the last one.

During interval INT_2, the total load on P^2 increased at the end of INT_1, with a significant part of bags recirculating on the pre-sorter up to 12-15 times (while it was normally at most one round). It meant the pre-sorter could not handle the higher load, caused by extra bags that could not be handled on P^1. Note, the remaining figures of this section show only aggregate performance spectra because non-aggregate ones do not read well when printed.

Next, we analyzed which *tasks* had the bags recirculating on P^2 (on seg_2^1), i.e., which process steps they were waiting for. For that, we defined

- a *domain-specific* performance classifier $\mathbb{C}_{bt}$ that mapped each bag task into a unique performance class using the corresponding event attribute *task*,
- a layer $lr_{bt} = (SEG^1, \mathbb{C}_{bt})$, and
- a channel $ch_{bt} = (\mathbb{C}_{bt}, pending, p)$,

and computed the performance spectrum for this layer and channel. Its segment seg_2^1 (bags on P^2) is shown in line 3 of Figure 3.27.

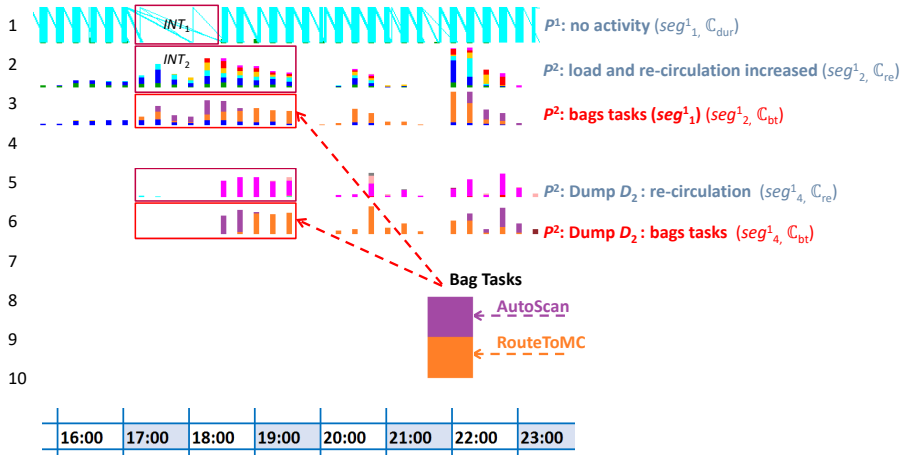


Figure 3.27: Many recirculating bags had to be automatically or manually identified (tasks *AutoScan* and *RouteToMC* respectively), but were dumped out of the system via exit D_2 instead.

Additionally, we considered segment seg_4^1 corresponding to the dump exit D_2 to see if any bags were dumped out of the system. For this segment, we show recirculation in line 5, and bag tasks in line 6 of Figure 3.27.

In the resulting spectrum (Figure 3.27), we saw the following:

- the majority of bags were waiting for either *automatic* or *manual identification* (see line 3),
- the load on pre-sorter P^2 started decreasing in the middle of INT_2 (see line 3) because the system started dumping bags that made many rounds on P^2 (line 5) through the dump exit (seg_4^1) out of the system in the middle of INT_2 (see line 6).

To summarize, during the first half of INT_2, many bags were accumulated on P^2 because they could not be identified. To prevent the overflow of P^2, the system started to dump them out of the system in the middle of INT_2.

Next, we wanted to understand why so many bags on P^2 were waiting for identification. Indeed, each bag should have been already identified upon entering the pre-sorter. We decided to check if it happened because of the so-called *lost-in-tracking* phenomenon. That is, if a system cannot detect (track) a bag at the location at the time when it is expected to be there, the bag is considered to be lost-in-tracking. Alternatively, when a system detects a bag at a location when there should not be any bags, the system considers such a bag to be lost-in-tracking as well. So, we defined a performance classifier $\mathbb{C}_{lt}$ to estimate the number of lost-in-tracking bags. We defined the corresponding layer, and channel, and computed the spectrum. Its segment seg_3^1, showing the number of lost-in-tracking bags, is shown in line 4 of Figure 3.28.

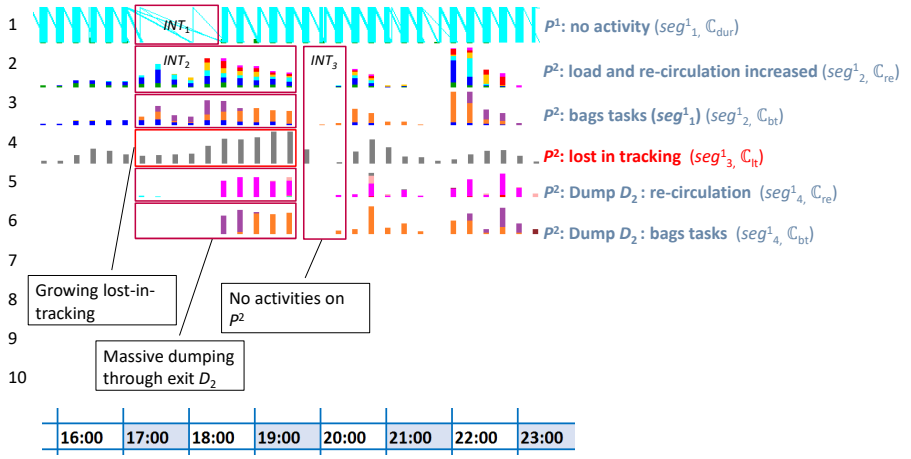


Figure 3.28: Growing lost-in-tracking on P^2 during INT_2 (line 4), and no activity on P^2 during INT_3.

In this spectrum, the amount of lost-in-tracking bags was growing within interval INT_2. Lost-in-tracking typically happens because of equipment malfunctioning, or bag misplacing. However, the exact reason usually cannot be directly determined from recorded event data attributes.

Eventually, at the end of the interval INT_2, any activities on pre-sorter P^2 stopped (see interval INT_3 in Figure 3.28), despite the extensive dumping of bags (line 5). Why did it happen? To investigate it, we defined a performance classifier $\mathbb{C}_{fd}$, which assigned classes according to *failed direction errors*, i.e., errors of diverting bags to other conveyors. The resulting performance spectrum is shown in Figure 3.29, lines 7 and 8.

In the resulting spectrum,

- segments seg^1_4 (line 7) and seg^1_5 (line 8) in Figure 3.29 show how many errors happened on diverting to the dump exit and manual encoding stations respectively, and
- seg^1_6 (line 9) shows the number of bags exiting these stations.

As described in Section 1.1.2, the manual encoding stations serve for identifying bags manually (task *manual identification*). In the spectrum, the number of errors *Not available or full* was gradually growing for diverting to the dump exit and the manual encoding stations, while activity on their exits (line 9) dropped to zero during interval INT_4.

To interpret the information in lines 7-9, we still needed to know if any operators were actually working at the stations during INT_4. For that, we defined a performance

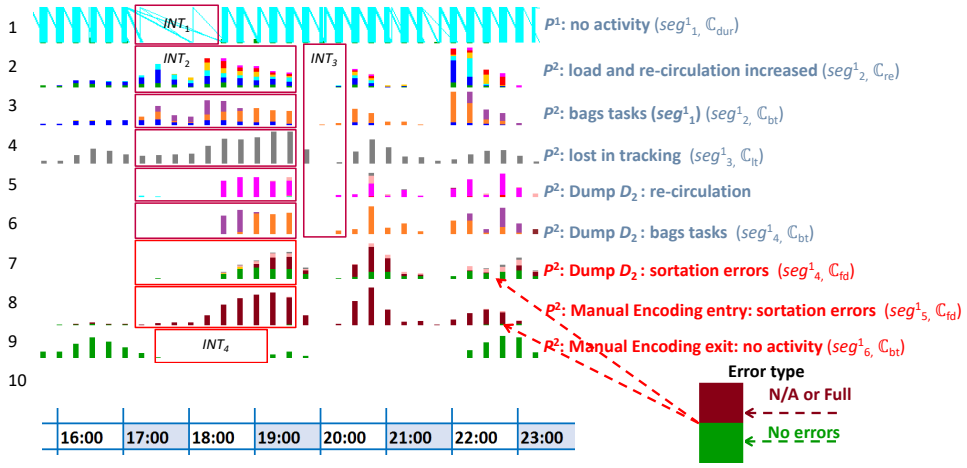


Figure 3.29: Errors on diverting toward the manual encoding stations (line 7) and dump exit D_2 (line 8); no activity at the manual encoding stations during INT_4 (line 9).

classifier $\mathbb{C}_{ms}$, which allowed us to see the stations' state. Thus, seg_7^1 (line 10) in Figure 3.30 shows that some manual encoding stations were *available* (with operators logged in) during INT_3. Nevertheless, they were not handling any baggage during this interval.

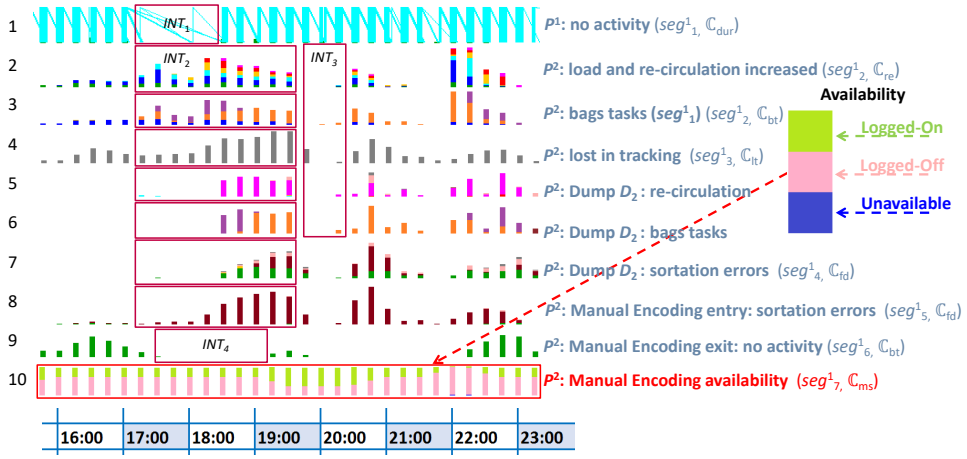


Figure 3.30: Availability of the manual encoding stations during the whole period (green bars in line 10).

Interpretation. By obtaining the information presented above, we reconstructed the whole chain of events, shown in Figure 3.31, and interpreted it as follows.

1. Pre-sorter P^1 had stopped for a reason not recorded in the given dataset (see the box in line 1 in Figure 3.31).
2. The load on another pre-sorter increased, as well as the amount of recirculating bags (arrow 1 in Figure 3.31).
3. Higher load caused the growth of lost-in-tracking bags (arrow 2 in Figure 3.31).
4. The lost-in-tracking bags had to be identified (arrow 3). As their significant part received task *RouteToMC* (orange bars in line 3), they had to be identified *manually* at the manual encoding stations.
5. The manual stations were available with operators logged in (green bars in line 10). However, they could not operate because:
   - already identified bags could not leave the stations because they could not merge onto P^2 due to high load (arrow 5), and
   - no new bags could enter the stations because they were fully packed with the bags waiting to be merged onto P^2 (line 4).
6. As a result, bags that were recirculating too many rounds (because of waiting for entering the manual stations) (line 6) were dumped out of the system.
7. However, the dump area D_2 got fully packed due to massive dumping, and the system could not get rid of the extra load anymore. As a result, no activity could be performed on P^2 (except meaningless re-circulations), so the pre-sorter was stopped (line 8) for removing extra bags *manually* by workers.

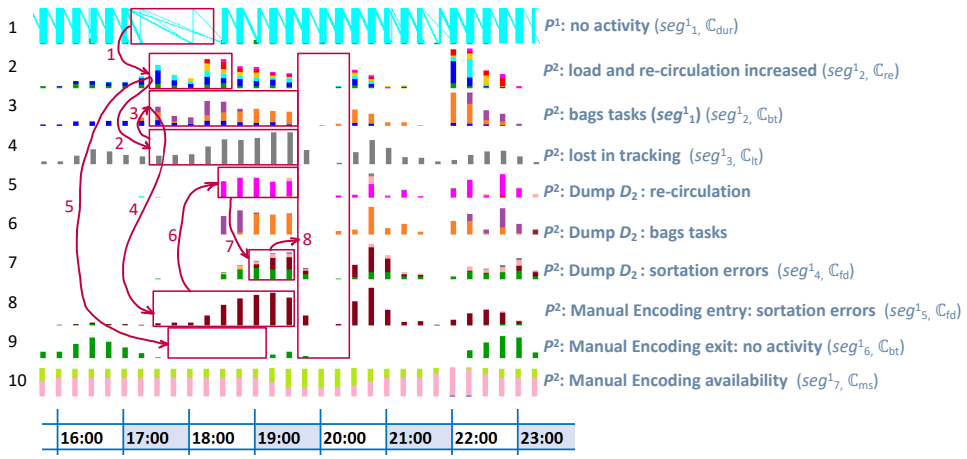


Figure 3.31: Reconstructed chain of events.

First, during the analysis, the inactivity of the manual encoding stations looked like a problem, but in reality, it was caused by the high load of pre-sorter P^2, i.e.,

identified bags could not be injected back onto the pre-sorter from the manual stations because the pre-sorter had no free space for that.

However, was the incident *unavoidable* after the failure of P^1? No, it was not. Apparently, some equipment malfunctioning caused a lot of lost-in-tracking bags, which still could be re-identified *automatically*. Instead, the system sent them for *manual* identification, and this behavior caused the final halt of P^2. This flaw in the baggage handling process, caused by factors we cannot disclose, was the root cause of this severe performance incident.

As the evaluation at Vanderlande proved, our analysis led to the right conclusions. Remarkably, the analysis was done much faster than the same analysis performed earlier by Vanderlande's experts without using the performance spectrum. Note, in the presented slides we did not show non-aggregate performance spectra due to their poor readability on paper for a large number of segment occurrences. Nevertheless, during analysis, we used it extensively along with the aggregate ones.

3.5.2 Evaluation Results

This evaluation showed that the performance spectrum can be extremely helpful for the performance analysis of logistic processes (**EvQ-1**). Moreover, both layer and multi-channel performance spectra were needed for different analysis steps (**EvQ-2**) because the use of different performance classifiers provided invaluable insight into the system performance. Another result of crucial importance was the amount of time spent on the analysis, which was significantly less than the time spent during the "classical" analysis of the same problem (**EvQ-3**). However, there were some difficulties during the analysis: a large amount of information, available in the system MFD, should be kept in the analyst's mind (**EvQ-4**). Finally, after presenting the analysis results for various types of audiences, including Vanderlande's top managers, we concluded that the information represented through the performance spectra can be easily digested by the audience without any data analysis skills (**EvQ-5**).

3.6 Chapter Summary

In this chapter, we proposed the unbiased fined-grained process performance description, called a performance spectrum. We defined all the building blocks of the performance spectrum, defined the performance spectrum for a single segment, and lift these definitions to many segments observed in an event log. We showed how this novel description of the process performance is capable of capturing various aspects of the process performance and revealing various performance patterns, whose pattern parameters' taxonomy we proposed as well. We suggested how the information in the performance spectrum can be quantified in the aggregate performance spec-

trum, which can, in turn, has multiple channels to represent various performance classifiers, segments, and ways of aggregation of individual cases. We provided the results of the empirical study proving that the same performance patterns can be identified in various event logs by different untrained analysts independently. Finally, we showed how both types of performance spectra, meaning the non-aggregate and aggregate ones, allow for the analysis of performance incidents in large complicated logistic processes for determining their root causes faster than the classical process mining and general purpose tools. For that, we reported on our evaluation using the dataset of a Vanderlande-built BHS of a large European airport.

Further in this thesis, the performance spectrum and performance patterns are extensively used for explaining various performance phenomena in MHSs in Chapters 4, for explaining our log repair method in Chapter 7, for analyzing undesirable performance scenarios in Chapter 8, and as the source of rich performance-related features for PPM in Chapter 9.

Chapter 4

The Nature of Material Handling Systems

The preceding chapter showed how the performance spectrum, i.e., a model-less process performance description, can be used for answering difficult performance analysis questions. However, other things being equal, analysis techniques can benefit from the information available from the model, potentially allowing for a richer and more accurate outcome. In this chapter, we consider what an MHS is in a nutshell and which system entities and behavioral phenomena should be captured by a model of an MHS for answering particular Analysis Questions (AQs). For that, we first identify MHS fundamental building blocks and discuss how they handle materials. Then, we distinguish MHSs with and without batching, and scope this thesis to the latter. The selection of the **AQs** is based on the results of the field study conducted with Vanderlande's process engineers supporting real-world MHSs all around the world. Next, by doing a literature study, we identify queueing theory as one of the most appropriate operations research fields for modeling and the structural, qualitative, and quantitative analysis of MHS without batching. We consider the state-of-the-art queueing theory-based techniques designed for the MHS modeling and analysis. Finally, we consider the limitations they imply and discuss the feasibility of their use for answering the **AQs**, as well as the key takeaways that, among other things, inspired our approach for modeling MHSs, which is presented in Chapter 6.

4.1 Classes of Material Handling Systems

In Section 1.1.2, we considered the BHS example in detail, focusing mostly on the system functions, processes, and key equipment elements. In this section, we take a broader look at different types of MHSs (not only BHSs) to identify what commonal-

ities allow all the equipment pieces to be seamlessly connected into a system. Based on these commonalities, we identify two types of *building blocks*, and also *units* they can comprise. Then, we show that Transport and Storage Units (TSUs), e.g., bags for BHSs and cartons for warehouse systems, can be handled *individually* or *in batches*, thereby defining two classes of MHSs.

4.1.1 Building Blocks

As we showed in Section 1.1.2, an MHS is a complex network of conveyors that move TSUs via locations to their final destinations. On the way, various process steps are executed either by machines or by workers at workstations. Typically, a TSU never leaves the surface of conveyors as it goes through the system because the machines and workstations are installed *over* or *near* the conveyors. For example, a screening X-ray machine is typically installed over a conveyor to screen TSUs passing through. Similarly, a workstation is typically installed over a conveyor as well, and its worker handles TSUs passing on the conveyor. Both machines and workers can temporarily stop the conveyor's movement for executing a longer process step, e.g., for making an X-ray shot, or for scanning an attached sticker with a hand scanner. That is, both machines and workers execute process steps in the same way. In the remainder of this thesis, we use the term *resource* for referring to machines and workers, emphasizing that (1) we do not model them differently, and (2) both are indeed the resources who *execute* process steps in MHSs. Thus, we have two types of fundamental building blocks:

1. *resources* that execute process steps,
2. and *conveyors* that move TSUs between resource locations.

In the following, we take a closer look at conveyor types, and basic unit types that can be assembled from a single resource, and one or multiple conveyors.

4.1.1.1 Conveyors

In this thesis, we call a *conveyor belt* any surface that moves TSUs, placed onto it, along a linear trajectory from the beginning to the end. At any moment in time, all the TSUs placed on the surface move at the same speed as if the surface is solid. When a conveyor belt starts or stops, the TSUs on it start or stop their movement (in space) respectively. In Figure 4.1, the conveyor belt surface moves from the beginning b toward the end e, carrying TSUs $pid1$ and $pid2$ ($pid2$ follows $pid1$).

We call a *linear conveyor* a chain of conveyor belts $\langle c_1, \ldots, c_n \rangle, n \geq 1$, such that any non-ending conveyor belt $c_i, 1 \leq i < n$, hands TSUs over to the next conveyor belt c_{i+1}. Note, if conveyor belt c_{i+1} is stopped, it is *unavailable* for taking TSUs from the previous conveyor belt c_i. In this case, c_i must stop when a TSU reaches its end. In

Figure 4.1: Conveyor belt carrying TSUs *pid*1 and *pid*2 from location b toward e.

Figure 4.2(a), the linear conveyor consists of four conveyor belts $\langle c_1, \ldots, c_4 \rangle$ and carries TSUs *pid*1 – *pid*3. Because conveyor belt c_4 is temporarily stopped for maintenance, *pid*1 cannot be handed over from c_3 to c_4, so c_3 is also stopped. At the same time, *pid*2 and *pid*3 can be moved until *pid*2 reaches the stopped conveyor belt c_3. Based on this behavior, an *accumulating linear conveyor* can be assembled. To minimize time when a linear conveyor is stopped while a TSU on its end is waiting for handing over, it has a queue at the end. It is made of multiple short conveyor belts, that keep together several TSUs, waiting for handing over, while the "main" conveyor belt still can move the other bags on it. Accumulating linear conveyors are typically installed right *before* resources and conveyors that can be often temporally unavailable because of a long-lasting manual operation, the lack of free space for merging incoming bags, and so on, to minimize the unavailability time of the conveyors leading to them.

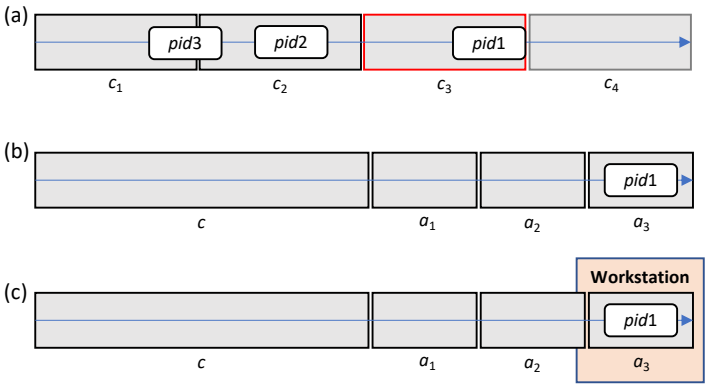


Figure 4.2: Linear conveyor (a), accumulating linear conveyor with "main" belt c and accumulating belts $a_1 - a_3$ for keeping waiting TSUs (b), and accumulating linear conveyor before a workstation (c).

For example, in Figure 4.2(b), the accumulating linear conveyor $\langle c, a_1, \ldots, a_3 \rangle$ consists of one long conveyor belt and three short ones, each of which can hold at most one TSU. Conveyor belt a_3 delivers TSUs to the workstation, i.e., the workstation uses it as a capacity element to hold a TSU being handled. When the workstation's worker is busy with the current TSU, e.g., *pid*1 on a_3 (Figure 4.2(b)), a_3 is stopped, while the

longer conveyor belt c still can operate unless (1) the accumulating conveyor belts ($a_1 - a_3$) are occupied, and (2) some TSU has reached the end of conveyor belt c.

In this thesis, we use the term *conveyor* when we do not distinguish conveyor belts, linear conveyors, and accumulating linear conveyors.

4.1.1.2 Resources and Units

In MHSs, a resource is seldom installed stand-alone. Instead, resources and conveyors are typically assembled into units of different types, so we *exclude* stand-alone resources from this thesis scope. In this section, we describe the most typical unit types. First, we describe a single resource installed over a conveyor, and then we describe more complex unit types, consisting of one resource and *multiple* conveyors.

Single-Resource Single-Conveyor Unit. While conveyors are moving TSUs throughout the system, precise information about the location of each TSU is required for computing their optimal routes and triggering the execution of process steps. Although on entering the system, the exact TSU location on the conveyor belt can be known by the system, later it can be lost due to various reasons. For example, TSUs can slide forward or backward (e.g., slippy bags in rainy weather), move unpredictably because of vibration, and so on. As tracking exact TSU positions at every location is costly, an MHS usually tracks their locations only before process steps that need this information. For example, a resource diverting (pushing) a TSU to another conveyor needs to know its exact location on the conveyor, otherwise, it may push just "the air" between two TSUs, or even push a wrong TSU. To avoid such problems, *sensors*, installed on conveyor belts throughout the system, track the precise TSU locations directly before resource locations. A typical sensor consists of a light source, installed on one side of the conveyor belt frame, and a photocell, installed across on another side (Figure 4.3). The photocell is capable of detecting the light of the source. When a TSU reaches the sensor, it obstructs the light, thus the TSU *front* side is detected. When the TSU has passed the sensor, the light is seen by the photocell again, thus the TSU *back* side is detected. As a result, the system obtains information about the precise TSU location. Note, such sensors detect TSUs but do not identify them. Dedicated resources serve for identification instead.

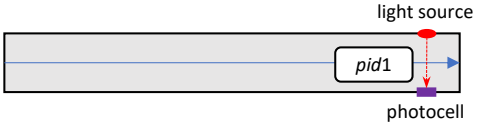


Figure 4.3: Passing TSU obstructs the sensor light beam.

Merging and Diverting Units. Combinations of conveyors and single-resource single-conveyor units can form a variety of linear paths. However, material handling processes, as well as many system design constraints, typically require a more complicated network topology (e.g., like one in Figure 1.1) for connecting resource locations. As a result, units for merging TSUs coming from several conveyors into one, as well as units diverting TSUs from one conveyor to others, are required. We call the former *merging units*, and the latter *diverting units*. We call all the TSUs being transported by the same conveyor within some time interval a *flow*. We limit ourselves to the units merging *two* flows, and the units diverting TSUs from one flow to *another*, assuming that most complex units can be built from combinations of simple ones.

Figure 4.4(a) shows a merging unit, consisting of a "main" conveyor c_{main} and incoming conveyor c_{inc}, where c_{inc} inserts TSUs at the merge location (shown in orange) in the *middle* of c_{main} from its "right-hand" side, i.e., not at the beginning. In the figure, TSU *pid*4 is about to be merged. However, it must wait until free space *fs* between *pid*3 and *pid*5 reaches the merge location.

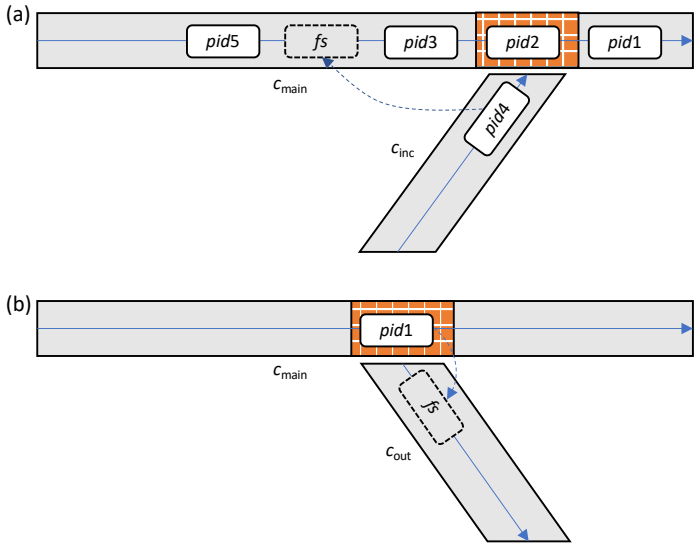


Figure 4.4: Merging unit (a), consisting of a main and incoming conveyor, and a diverting unit (b), consisting of a main and outgoing conveyor; TSUs can be merged/diverted if there is some free space *fs* available.

Figure 4.4(b) shows a diverting unit, consisting of a "main" conveyor c_{main} and outgoing conveyor c_{out}, where c_{main} diverts TSUs onto c_{out}. Similarly to merging units, some free space must be available at the beginning of c_{out} for diverting a TSU. Otherwise, it cannot be diverted and continues its journey on c_{main}. Note, on MHS conveyors typically some space, that we call *protective space*, is needed around (i.e.,

before and after) a TSU to avoid an accidental involvement of neighboring TSUs into a process step executed for a TSU. Otherwise, for example, on divert, the surrounding TSUs can be accidentally diverted as well.

Loops. We call a short-circuited conveyor a *loop*. In Figure 4.5, a loop is shown together with a conveyor c_{shortcut} that "shortcuts" the loop for allowing TSUs to skip a part of the loop (e.g., path $a-x-b$ in Figure 4.5) if needed. In reality, a loop is usually connected with multiple incoming and outgoing conveyors and can have zero, one, or multiple shortcuts. For example, sorting loop P_1 in Figure 1.1 has several incoming conveyors ($b_1,\ldots,b_4$, d_1,d_2 etc.), several outgoing conveyors, k shortcuts for routing baggage through the security scanners, and one shortcut for manual encoding station M_1.

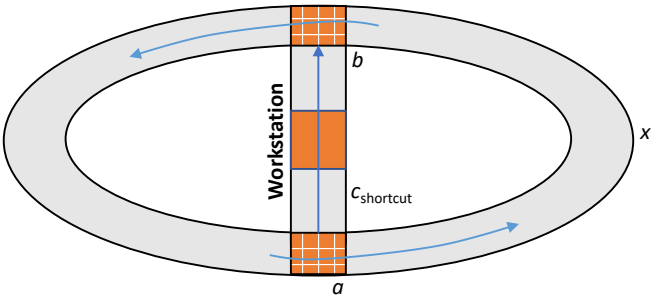


Figure 4.5: Loop with shortcut c_{shortcut} going through a workstation and skipping path $a-x-b$.

4.1.2 Material Handling Systems with and without Batching

So far, we considered how MHSs handle TSUs with conveyors and resources, assuming an individual TSU, e.g., a single bag or carton, as the only type of material being handled. However, it is true only for a part of the real-world MHSs. Some MHSs are capable of consolidating items of different types and handling them together in a *batch* [74]. For example, some warehouse systems allow consolidating heterogeneous Stock Keeping Units (SKUs) as prescribed in incoming customer *orders*, e.g., a dozen coca-cola bottles can be consolidated with a box of chips. After consolidation on a tray, the SKUs travel *together* on the system conveyors. Additionally, one set of consolidated SKUs (batches) can be re-consolidated into another set, and so on. As a result, the complete system behavior comprises the handling of individual SKUs, their batches, full and empty trays, etc. We relate the former and latter system types to the following two MHS classes:

1. *MHSs without batching*, and
2. *MHSs with batching*.

Nowadays, the majority of existing MHSs are systems without batching. However, many large systems have batching [74].

Taking into account a large number of open questions, related to systems without batching, we limit the thesis scope to this class of systems. In the remainder of this thesis, we use the term MHS for referring to systems without batching. In the next section, we provide the **AQs** actual for analysis and operational support of MHSs that we collected at Vanderlande.

4.2 Analysis Questions

The most important and difficult problem of MHS operators, and process engineers supporting the systems, is keeping the system *performance* on a desired level. It requires abilities for understanding the system behavior, especially when it differs from "normal". Previously, we showed how an MHS can be seen as a composition of some simple building blocks and units, whose behavior in isolation may seem relatively simple. However, real MHSs consist of hundreds or thousands of interconnected units, controlled by complicated business rules. It makes the system behavior overwhelmingly complicated and difficult to understand and analyze.

To identify the most actual problems related to MHS performance analysis, we conducted a field study with Vanderlande's process engineers, whose daily duty was to help the customers to keep the performance of their MHS on the designed level. We collected and documented the problems as 60 *user stories*. Afterward, we organized a workshop for ranking them according to their importance (from the engineers' point of view). Among the top-ranked user stories, we selected *eight* for our research. They were selected (1) as most actual, (2) as not solved by existing techniques, and/or (3) because they blocked our research related to the other selected stories. In the following, we present these stories, which we refer to as the **AQs** throughout this thesis. Note, we keep the language of the stories mostly intact, to avoid introducing any bias to the stories.

AQ1. **A high-level overview of the system behavior.** "As a Process Engineer or Customer, I want to see the main parameters of a system such as throughput (e.g., in bags per minute), travel durations between system locations, flows between process steps, re-circulations on loops, and so on, projected on an MFD. *I need that* for communicating with the customers and for learning the system. For example, I want to know the travel time between check-in counters $q_1^1 - q_1^{nz1}$ and scanner S_1 for the last 20 minutes (see Figure 1.1)."

AQ2. **An animated or dynamic representation of the system.** "As a Process Engineer, I want to see a dynamic view of the system as the most understandable one. *I need that* for engaging the Customer for further collaboration. For example, I want to see how the preliminary sorting area is working right now (see Figure 1.1)."

AQ3. **Detecting bottlenecks and understanding their causes.** "As a Process Engineer or Customer, I want to detect bottlenecks in general or for particular flows and understand their root causes. *I need that* for communicating and improving the system performance, and for root causes analysis. For example, I see that the link between the transfer-in area and the sorting loop P^2 is a bottleneck. I want to understand what caused it and how to get rid of it for improving the system performance."

AQ4. **The Detection, analysis, and prediction of whether a bag can make it to the flight.** "As a Process Engineer or Customer, I want to detect all bags that are late for the flight, understand why they are late, and I want to predict such cases. *I need that* for analysis of root causes to understand how to avoid such problems, and I want to predict such cases to prevent costs related to the delivery of such bags to passengers. For example, a bag of passenger X could not make it to the flight. Its delivery to X costs extra 200$. The bag was late because it failed to be identified automatically and had to wait too long for manual identification, making rounds on loop P^1, due to temporal unavailability of MS m_1. I want to predict such scenarios for, for example, making MSs available timely."

AQ5. **Analysis of the stalling of sorting loops.** "As a Process Engineer or Customer, I want to analyze and find the root cause for the sorting loop stalling. *I need that* for the fast understanding of the problem causes and their quick fixing. For example, the stalling of sorting loops P^1, P^2 of the system of Figure 1.1 can happen because of one or many of the following reasons:

- overload of MSs m_1, m_2,
- overload of scanners $S_1 - S_k$,
- delays in the manual part of the screening process,
- the load balancing between P^1 and P^2 is not proper.

The actual reason is needed to be found and fixed as soon as possible for avoiding the extra costs of late bag delivery."

AQ6. **Detecting and analysis of control flow outliers.** "As a Process Engineer or Customer, I want to detect all cases when cases (bags) did not follow paths allowed by the system design to analyze such cases further. *I need that* for (1) timely maintenance of sensors and photocells, (2) checking consistency of recorded event data, (3) conformance checking of process models. For example, while analysis of recorded event data, a direct path, that bag *pid*1 took, between the transfer-in area and scanner S_1 is observed. That means that sensors between these locations failed to detect this bag and need maintenance, or the event recording process has errors."

AQ7. **Detecting outliers in terms of time taken to reach the destination.** "As a Process Engineer or Customer, I want to see descriptive statistics about paths taken for the cases (bags) that took (1) the expected duration, (2) more than the expected duration, and (3) much lesser than the expected duration. *I need that* for root cause analysis, making feasibility reports, and system testing. For example, I want to see

descriptive statistics over bags that took paths from scanners $S_1 - S_k$, for each path variant I want to see which bags traveled faster and slower than on average."

AQ8. **Detecting and predicting aggregate KPIs.** "As a Process Engineer or Customer, I want to detect and *predict* when a load of a particular part of a system is above a certain threshold. *I need that* for preventing overload of system parts, and for preventing bottlenecks on sorting loops. For example, I want to know that the flow of bags to MS M_1 (see Figure 1.1) in bags per minute is below some threshold, e.g., 0.5 bags per minute, and I want to predict when it is going to be higher, to prevent overload of sorting loops P^1, P^2 because of bags waiting for the manual identification."

4.3 Modeling Building Blocks and Units with Queues

Previously, we identified queueing theory as the most appropriate way of modeling systems without batching. In this section, we show how the state-of-the-art approaches allow for modeling MHSs with queues of two different types. For that, we first consider a single-server queue for modeling a single conveyor/resource unit in Section 4.3.1. Then, we study the speed-density effect observed on MHS conveyors and discuss how multiple-server queues can model them, taking the speed-density effect into account, in Section 4.3.2.

4.3.1 Resource Model

To explain how the single-resource single-conveyor unit can be modeled with queueing theory, we consider the behavior of an MHS sensor, using its performance spectrum as a detailed performance description. Then, we provide its queueing theory model, which also works for single-resource single-conveyor units.

Let us consider a conveyor $a : b$ with some finite capacity, moving TSUs from a location a to location b, and a sensor b installed at the conveyor end. Note, we use the same "name" (b) for the sensor and location, as it is usually done in real MHSs. The conveyor and sensor are shown in Figure 4.6(a).

Conveyor $a : b$ moved TSUs toward sensor b at a constant speed. Initially (Figure 4.6(a)), *pid*1 and *pid*2 were going toward $a : b$. At time t_1, *pid*1 was ready to enter it, while *pid*2 was slightly behind. Later, at time t_3, *pid*1 reached the sensor (Figure 4.6(b)). At time t_3, sensor b became busy with *pid*1. In the performance spectrum in Figure 4.6(e), the corresponding part of its journey is shown as occurrence o_1 of segment *waiting* (on conveyor $a : b$). In the meanwhile, after *pid*1 entered the conveyor and before it reached the sensor, *pid*2 also entered the conveyor at time t_2 (occurrence o_4 in the performance spectrum).

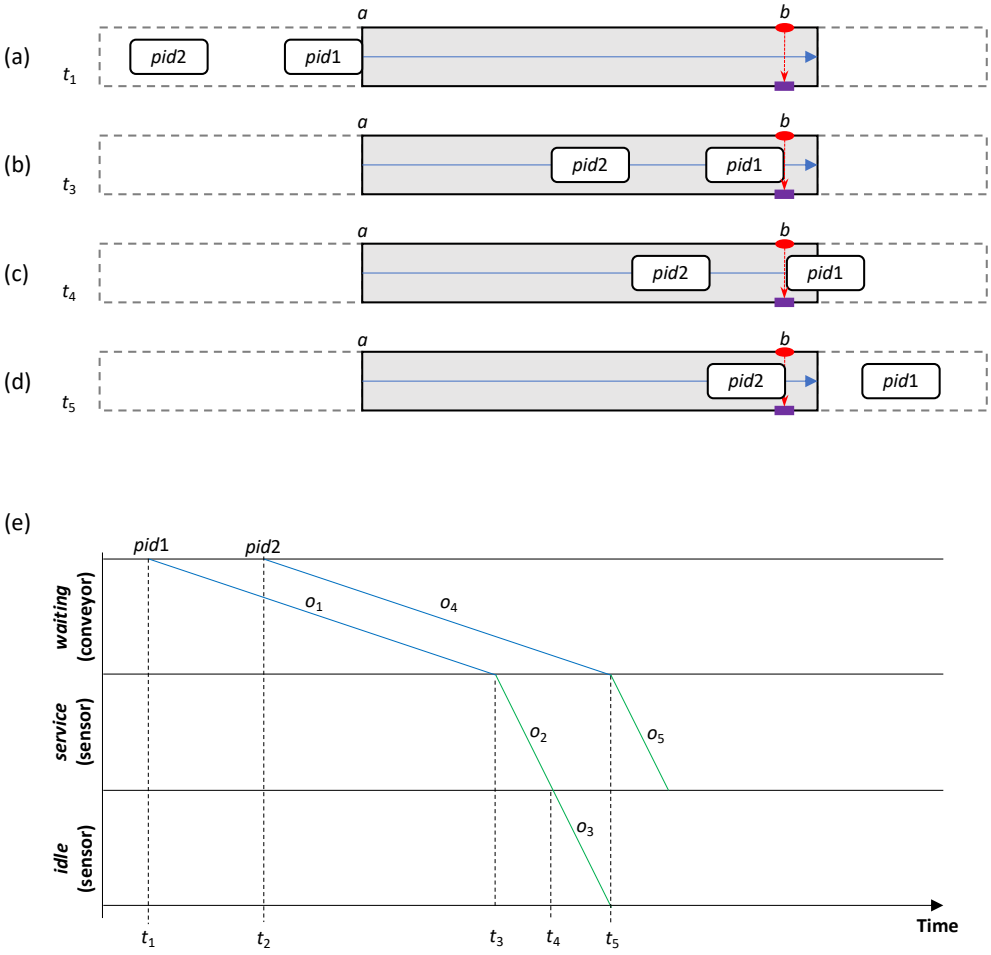


Figure 4.6: Detecting TSUs by sensor b: $pid1$ and $pid2$ are arriving to $a:b$ (a), the front of $pid1$ is detected (b), the back of $pid1$ is detected (c), and the front of $pid2$ is detected (d).

At time t_3, the sensor detected the front side of $pid1$ and became busy with it until time t_4, when it detected its back side. The sensor processed $pid1$ in the service time $t_4 - t_3$, shown as occurrence o_2 in the performance spectrum. Afterward, the sensor was idle till the other TSU arrived (occurrence o_3). The idle time included:

1. the time required for making protective space between neighboring TSUs ($pid1$ and $pid2$ in our example),
2. the time when the sensor had no TSUs to handle.

Eventually, *pid*2 arrived at time t_5, and the sensor served it in the same way (occurrence o_5). Note, because the conveyor moved at a constant speed, the service time was proportional to the length of the TSU being processed, and the idle time was proportional to the distance between the TSUs directly following each other, e.g., the distance between *pid*1 and *pid*2. In general, the service time may depend not only on the speed of the TSUs, passing the resource but also on many other things. For example, the time required for a TSU to be identified by a worker (a person) depends, among other things, on the presence and readability of the sticker with a bar code, attached to the TSU.

In queueing theory, MHS resources are typically modeled as servers and conveyors as queues. Indeed, a conveyor can be naturally seen as a queue where bags wait to be served, like bags on conveyor $a:b$ wait to be detected by the sensor. A conveyor moving bags toward a resource can be modeled as the classical general service model with a finite buffer [18]. It describes a queue with a single server, which serves jobs (bags), in the order of their arrival. This order is called the First-Come-First-Served (FCFS), or the First-In-First-Out (FIFO). In the extended Kendall's notation [75], it is written as *M/G/1/K* meaning the following.

- The *M* stands for a queueing system with stationary Poisson arrival process at a given rate λ, i.e., the times between job arrivals have an exponential distribution with parameter λ.
- The *G* stands for General service process. Job service times are generally distributed, and the service times of different jobs are *mutually independent*. When a job has been served, it departs from the queue.
- The 1 stands for the single server.
- The *K* stands for the finite capacity *K* of the waiting room, i.e., the conveyor capacity, including the capacity element of a job (bag) in service.

We write θ for the mean throughput rate, and μ for the service rate. The queue graphical representation is shown in Figure 4.7.

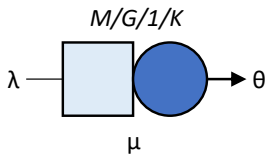


Figure 4.7: Representation of *M/G/1/K* queue.

4.3.2 Conveyor Model

In the previous section, we model the single-resource single-conveyor unit as a queue system where the service time is independent of the queue state. This approach may look good for MHSs, where conveyors typically move at constant speeds, and resources process each TSU independently. However, works on MHS modeling [18, 19, 20], as well as our research, show that conveyors have much more complicated dynamics. In the following, we explain how traffic speed on a road junction, i.e., the junction "performance", can be modeled using queueing theory. Then we show why the same approach can be used for modeling the (accumulating) conveyors of MHSs.

4.3.2.1 Speed-Density Effect

Vehicular Traffic Flow Example. Let us consider highway traffic. When traffic is light, vehicles can go through the link smoothly at the highest speed permitted. When traffic is moderate, drivers have to watch more vehicles around and take over slow-moving trucks, so the average speed drops. When traffic is heavy, it becomes difficult to overtake slow movers. As a result, these slow movers significantly impede the average speed and can cause congestion. In literature, this situation is typically illustrated by a curve showing how a traffic speed decreases as a car's density increases [21]. The similar dependency between the density of pedestrian traffic and its average speed is described in [27].

TSU Flows. Considering MHSs, we can use the analogy with vehicular traffic: TSUs are vehicles, and conveyors are roads. Indeed, TSUs have much less movement freedom, e.g., they cannot overtake each other. Nevertheless, there are a lot of similarities: TSUs must keep some minimal distance (protective space) between each other, they must follow roads (conveyors), they have speed limits (maximal conveyor speed), and so on. So, it comes as no surprise that the same speed-density effect is observed for MHS conveyors [18], i.e., the conveyor *average* speed drops as density (i.e., the conveyor load) increases.

We investigated if this effect can be observed in Vanderlande-built MHSs. For that, we analyzed several conveyors of different capacities (i.e., length), operating in the areas with different purposes. We computed the speed-density effect, using a dataset collected during six continuous months of operations. In Figure 4.8, the effect is presented as a curve showing the dependency between the conveyor load in TSUs (density) and median travel time (travel duration) through the conveyor (in seconds).

A similar trend was also observed for the other conveyors. Remarkably, the lower the conveyor capacity, the higher the effect. For example, for the conveyor with the designed capacity of 52 TSUs (see Figure 4.8), the difference between median duration under low and high load is just 5%, while for the conveyor with the smaller

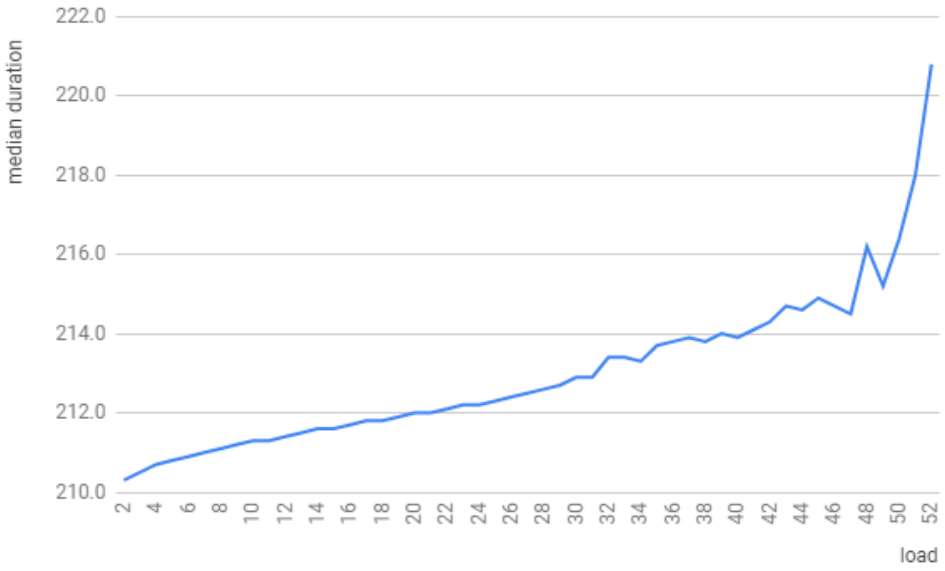


Figure 4.8: Speed-density effect observed for a long MHS conveyor with a capacity of 52 TSUs. Instead of TSU speed, median traveling duration is provided: longer duration shows a slower average speed.

designed capacity of 7 TSUs (see Figure 4.9), it is 32%. However, what phenomena cause this effect? To understand that, we explored the conveyor behavior using its performance spectrum (see Chapter 3).

Complex Conveyor Dynamics. To analyze the speed-density effect on MHS conveyors, we consider the following scenarios for an accumulating conveyor $a:m$ that merges bags onto conveyor $d:e$ through merge unit m (see Figure 4.10(a)):

1. a single-TSU scenario *without* the speed-density effect,
2. a two-TSU scenario *with* the speed-density effect,
3. a multiple-TSU scenario *with* the higher (than in the previous scenario) speed-density effect.

In the first scenario, TSU *pid*2:

1. started at the beginning of conveyor $a:m$,
2. traveled at a constant conveyor speed,
3. reached merge unit m,
4. merged onto conveyor $d:e$ via unit m,
5. continued travelling at the conveyor $d:e$ constant speed,
6. reached the end of conveyor $d:e$.

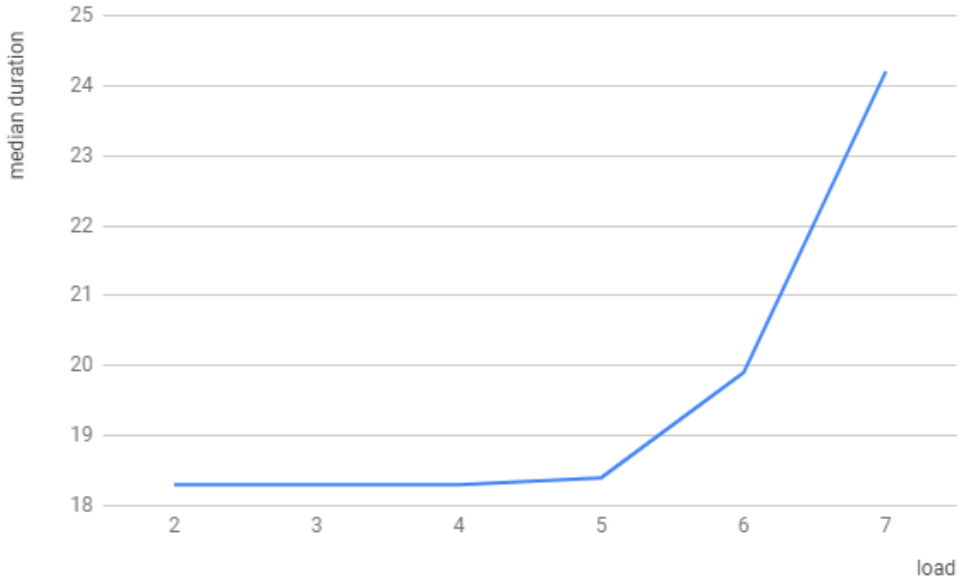


Figure 4.9: Speed-density effect observed for a short MHS conveyor with a capacity of 7 TSUs. Similarly to Figure 4.8, median traveling duration is provided instead of speed: longer duration shows a slower average speed.

Note, it had no obstacles (other TSUs) on the way, so it

- traveled at constant speed on both conveyors,
- smoothly (without any delays) merged onto conveyor $d:e$.

The performance spectrum of this journey is shown in Figure 4.10(b), where segments s_1 and s_3 describe traveling on $a:m$ and $m:e$, and segment s_2 describes the merge at m, i.e., passing through the merge unit and shifting to conveyor $d:e$. In this performance spectrum, occurrences $o_1 - o_3$ show the fastest possible performance scenario because there were no delays on the way.

Now, we consider the second scenario where another TSU *pid*1, traveling from location d to e, delayed the merging of *pid*2. For this scenario, we assumed that conveyor $d:e$ had a higher priority than $a:m$, so TSUs merging from $a:m$ onto $d:e$ had to yield TSUs on $d:e$ at merge location m. In this scenario, the TSUs were handled as follows.

Step 1. TSU *pid*1 started first at location d at time t_1.

Step 2. A bit later, *pid*2 started at a (t_2). Their locations are shown in Figure 4.11(a), where *pid*1 had already been moved from the beginning of $d:e$ toward

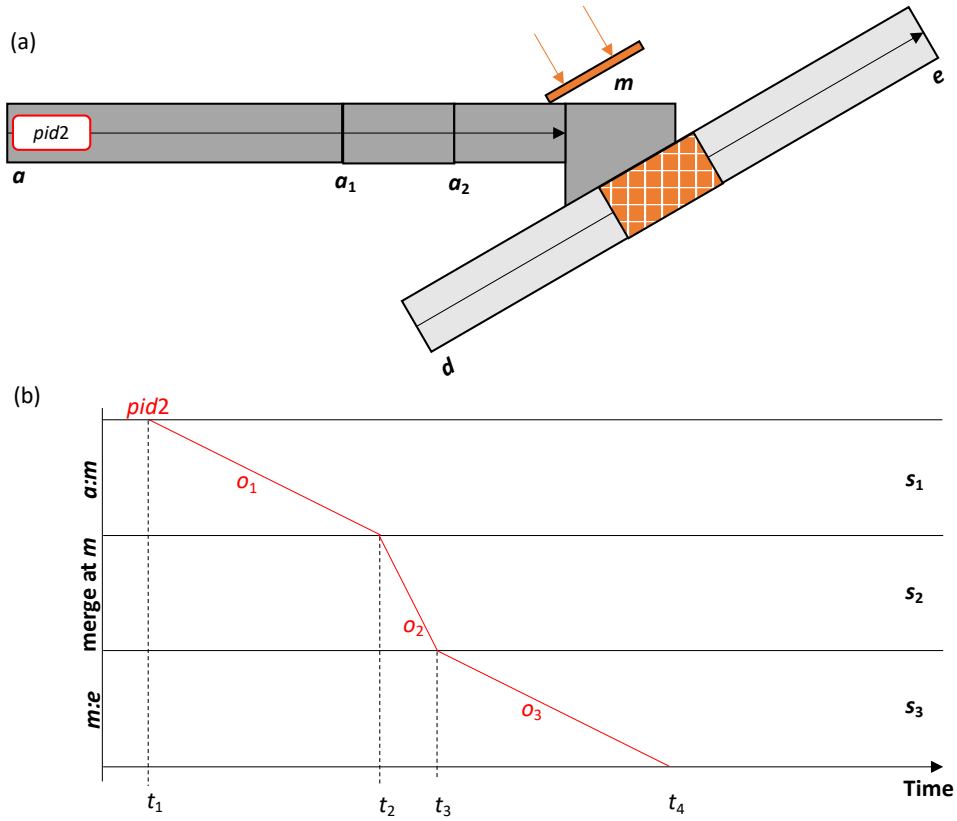


Figure 4.10: TSU *pid*2 merges without delays.

its end. The corresponding performance spectrum is shown in Figure 4.11(b), where the occurrences of segments $a:m$ (s_1) and $d:m$ (s_1') are shown on top.

Step 3. TSU *pid*1 reached merge location at time t_3.

Step 4. A bit later at t_4, *pid*2 reached m as well. However, *pid*1 effectively blocked *pid*2 from merging onto $d:e$ (Figure 4.12(a)). The corresponding performance spectrum is shown in Figure 4.12(b), where occurrence o_1 of $d:m$ describes traveling of *pid*1 from d to m, and occurrence o_5' describes travelling of *pid*2 from a to m till it became blocked by *pid*1. Note, that while o_1 is shown as a single straight line, o_5 for *pid*2 is split into two parts o_5'' and o"$_5$ to distinguish the movement time from waiting time at m. Note, the performance spectrum, as it is defined in Chapter 3, does not allow such compound occurrences; however, we use them to explain better the system behavior.

Step 5. While *pid*2 remained blocked at m, *pid*1 smoothly continued on $m:e$.

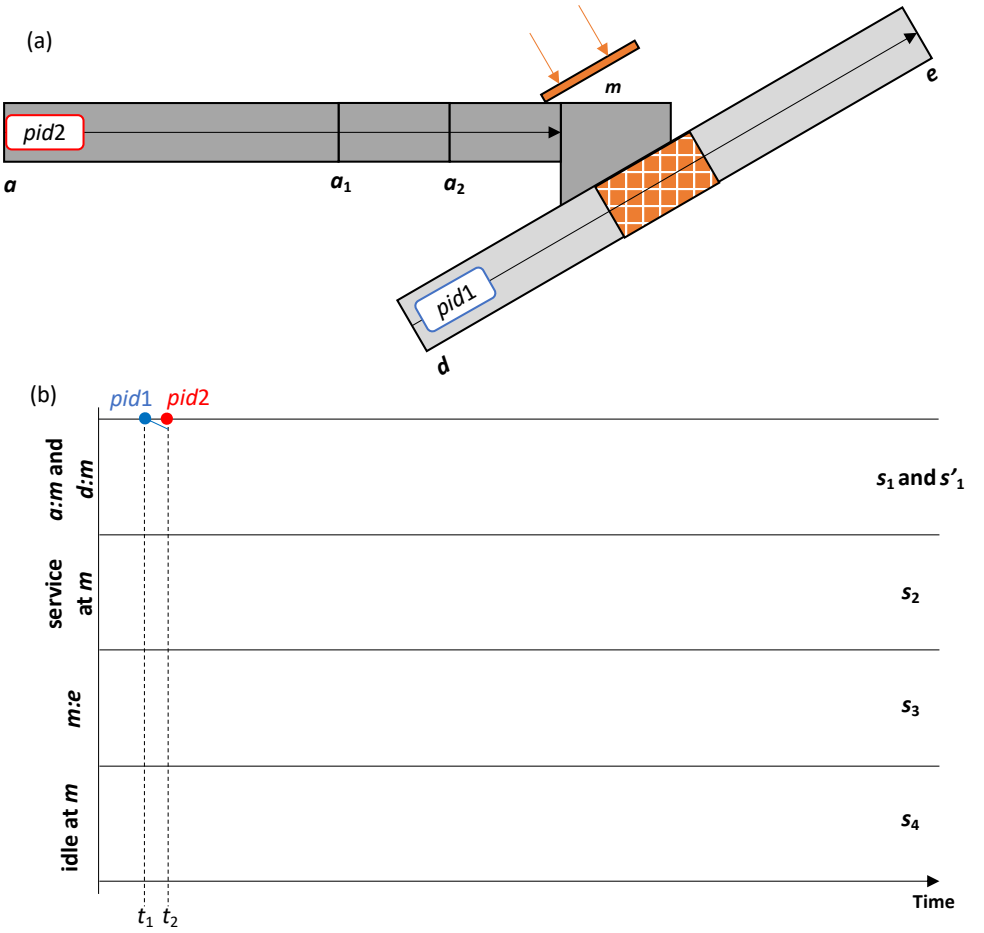


Figure 4.11: TSUs *pid*1 and *pid*2 started their journey at time t_2 (a), the corresponding performance spectrum contains starting occurrences on top (b); segments $a:m$ and $d:m$ are shown on top of each other.

Step 6. Finally at t_6, there was enough free space at m on $d:e$ for merging a TSU, so *pid*2 merged onto $d:e$ at t_7 (Figure 4.13(a)).

Note, to enable merging, all the following conditions must be satisfied:

- no TSUs are presented at merge location m on $d:e$, and
- there is some protective space PrS between a TSU that is the closest to m on $m:e$ (e.g., *pid*1), which we assume to be equal to the TSU average length.

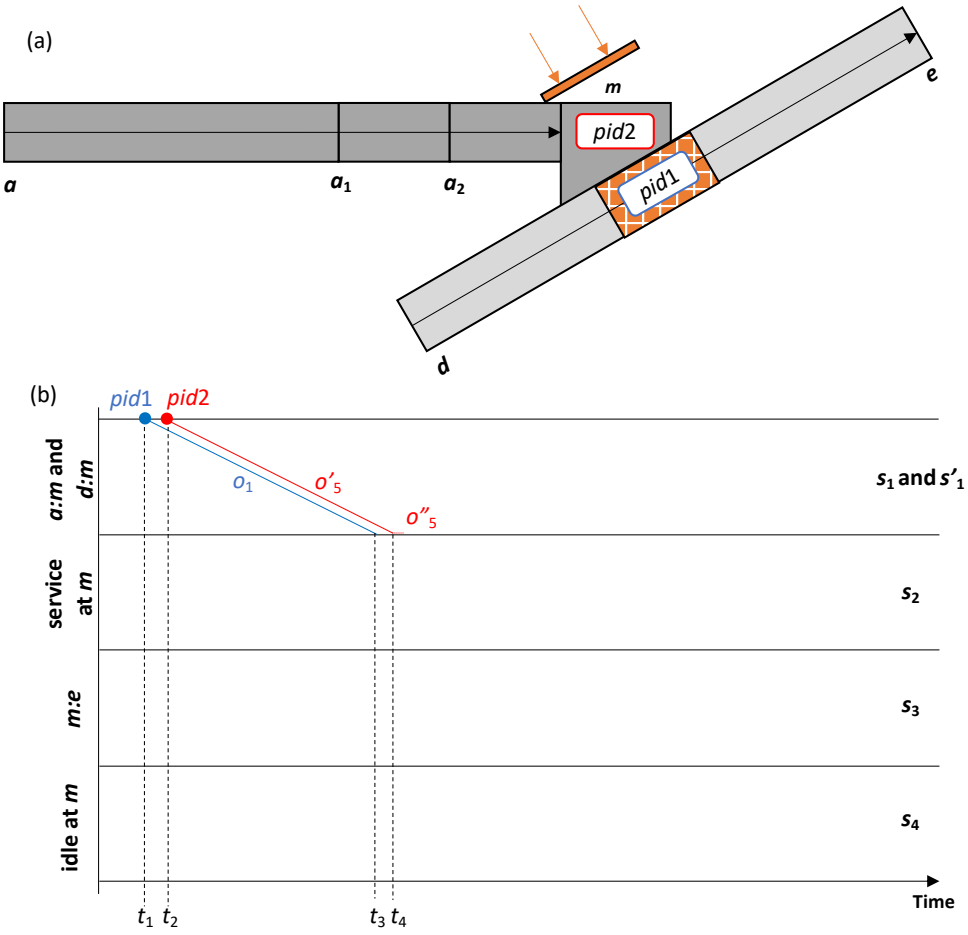


Figure 4.12: Both TSU reached the merge location almost simultaneously by time t_4. However, *pid*2 could not be diverted onto $d:e$ at t_4 because *pid*1 occupied the merge location.

That is, *pid*2 had to wait not only till *pid*1 left the merge location, but also till it traveled far enough to ensure *PrS* in between. The corresponding performance spectrum is shown in Figure 4.13(b), where:

- occurrence o_2 describes how *pid*1 passed m.
- an additional segment idle at m shows how m ensured space *PrS* between *pid*1 and *pid*2 (o_4) by waiting till *pid*1 was moved further from m toward e (the beginning of o_3).
- o_6 describes how *pid*2 merged at m,
- and o_7 describes the future movement of *pid*2 on $m:e$.

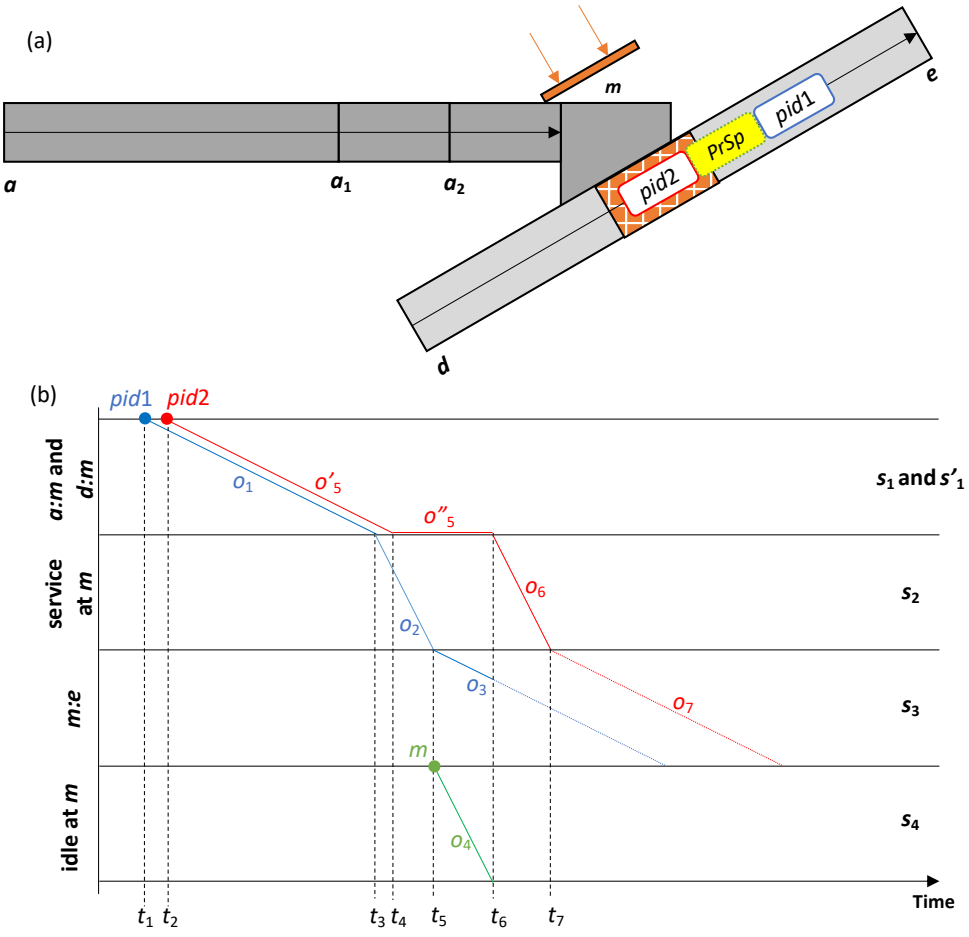


Figure 4.13: *pid*2 had to wait for $t_6 - t_4$ (occurrence o''_5) till *pid*1 left the merge location, and protective space *PrS* was created.

Finally, let us consider the third scenario where the load on conveyor $d:e$ was higher, as shown in Figure 4.14.

Initially, there were TSUs *pid*1 and *pid*2 (Figure 4.14) surrounded by protective spaces *PrS*. There was also a "free" area f' (filled with green squares) between the protective spaces. However, it was too short for a TSU to be merged. At the same time, accumulating conveyor $a:m$ had a queue of three TSUs (*pid*3 – *pid*5) at the end (belts $a_1:a_2$, $a_2:a_3$, and m). Moreover, there was already TSU *pid*6 before accumulating belt $a_1:a_2$, and *pid*7 at the conveyor beginning (location a). As a result,

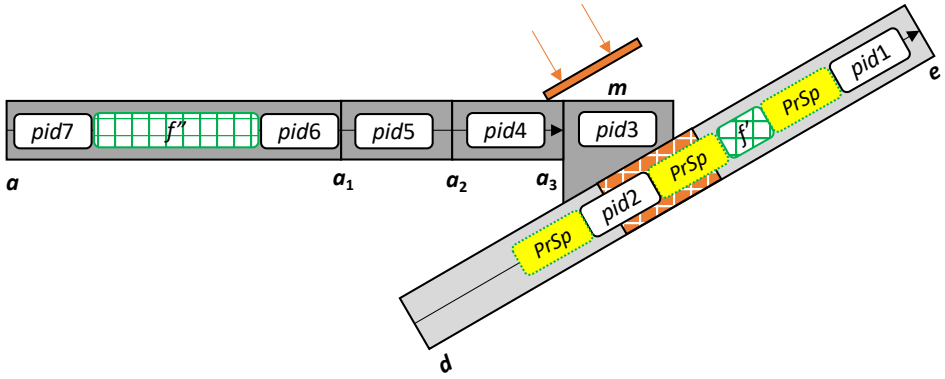


Figure 4.14: Under high load conditions, fewer gaps, long enough to accommodate a TSU and the surrounding protective spaces, were available; at the same time, free but unreachable spaces (e.g., f' and f'') were wasted, decreasing the effective capacity of the conveyors.

- TSUs *pid*1 and *pid*2 did not utilize the capacity of $d:e$ optimally because free space f between them was too small for a TSU to be merged, so it was a "wasted" capacity unit of conveyor $d:e$,
- *pid*6 could not be handed over onto $a_1:a_2$, so belt $a:a_1$ was stopped till $a_1:a_2$ was available. Additionally, free space f" between *pid*6 and *pid*7, i.e., some capacity of $d:e$, was wasted since no TSUs could reach it.

In this scenario, conveyor $d:e$ was not exceeding its maximal capacity, but it was still unavailable because of some non-optimal TSU spatial configuration that did not allow its full utilization.

We conclude that in general, a higher conveyor load increases the chances of a delay, as demonstrated in Figures 4.8 and 4.9, and explained by the examples above. Next, we show how conveyors exposing this effect can be modeled using queueing theory.

4.3.2.2 Modeling Conveyors as State-Dependent Queues

Previously, we explored the speed-density effect for vehicle traffic and TSU flows. If we use the state-independent queue, like *M/G/1/K* queueing system we introduce before, for modeling conveyors with this effect, we fail to reflect the dependency between the speed (or traveling duration) and density (conveyor load). To take this effect into account, the *M/G/c/c* state-dependent models have been described for modeling vehicular and pedestrian traffic flows [21, 27], as well as for modeling MHS conveyors [18]. In Kendall's notation, the *M* stands for stationary Markovian arrival (Poisson) process, *G* stands for General service process, the first *c* stands for *c*

parallel servers, and the last c stands for a queue of capacity c, including the capacity element of jobs in service. Because the capacity element of jobs in service is equal to the server number, it means actually no queue, as the whole capacity is used for jobs in service. $\mu(N)$ stands for the exponential mean service rate depending on the active job number. This queue graphical representation is shown in Figure 4.15.

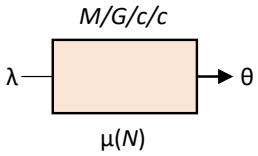


Figure 4.15: *M/G/c/c* state-dependent queue graphical notation: stationary Markovian arrival M (arrival rate λ), general service process G with c parallel servers and capacity (i.e., no queue).

Let us provide some intuition about the idea behind this model. Jobs arrive according to a Poisson process at arrival rate λ. Note when all c servers are busy, arriving jobs (representing TSUs) are lost to the system. The service times are distributed according to a general distribution G. In contrast to *M/G/1/K* system in Section 4.3.1, the service rate depends on the current number n of jobs in the system, i.e., each system server works at rate $f(n)$. So, when a job arrives or departs, the service rate changes to $f(n+1)$ or $f(n-1)$, respectively. A speed-density curve, such as we showed in Figure 4.8 and 4.9, can be linearly or exponentially approximated [21, 27]. For MHS conveyors, an exponential state-dependent delay curve is more accurate [18, 76]. Such an approximation can be incorporated into the model for deriving the probability distribution $P(n)$ of the number of jobs in the system.

Let the state of the system at any time be the amount of work already performed on jobs that are still in the system, i.e., the state is $x = (x_1, x_2, \ldots, x_n), x_1 \leq \cdots \leq x_n$. As argued in [27], if there are n jobs ($n \leq c$) in the system and $x_1, x_2, \ldots, x_n$ is the amount of work already performed on these jobs, the process of successive states will be a Markovian process in the sense that the conditional distribution of any future state will depend only on the present state. In the theorem presented in [27], Equation 8 defines the probability distribution for the *M/G/c/c* state-dependent queue model, depending on the number n of jobs in the system. This equation incorporates an approximated speed-density curve we discuss above. Then, in the corollary to the theorem, it is said that in the *M/G/c/c* state-dependent model, the departure process (including both jobs completing the service and those that are lost) is a Poisson process at rate λ. For further details, we refer to [27, 76]. That corollary is of crucial importance because it allows to *chain M/G/c/c* systems into a queueing network, as both, their arrival and their departure, are Poisson processes.

So, an (accumulating) conveyor of capacity c (TSUs) can be modeled as a *M/G/c/c* queue, in order to take the speed-density effect into account [18]. In the following, we show how the *M/G/1/K* and *M/G/c/c* models can be used for modeling the MHS building blocks, units, and their various configurations.

4.4 Modeling Material Handling Systems with Queueing Networks

Previously, we considered the state-dependent *M/G/1/K* model as a model of the single-resource single-conveyor unit, and state independent *M/G/c/c* queue model as a model of the accumulating conveyor. However, we need to model more complicated merging and diverting units, and connections between conveyors and units as well. For that, queue models can be organized into *queueing networks*. Queueing networks can be classified in different ways. However, in this section, we consider how MHSs can be modeled using *open* and *closed* queueing networks [75] with blocking [17], and how different MHS configurations can be modeled, using queueing networks of these classes.

4.4.1 Modeling with Open Queueing Networks

Open queueing networks assume *external* arrivals and departures, so the number of jobs *varies* over time. Within the network, a job that completed service at one queue can either *leave* the network, or go to another queue, according to the routing. The routing can be either *probabilistic* or *deterministic*. The example of an open queueing network with deterministic routing is shown in Figure 4.16. This queueing network represents the merge unit shown in Figure 4.10 and discussed in Section 4.3. We model it through a single *M/G/1/K* queue for the merge unit resource, and three *M/G/c/c* queues for the incoming and outgoing conveyors, as in [18]. TSUs arrive to either queue $a:m$ or $d:m$ according to a Poisson process with rate $\lambda/2$. After service is finished at either of the queues, it is deterministically routed to the queue m, and then to queue $m:e$, where it finally leaves the system. Note, in our example, the unit resource has a single capacity element (shown in orange in Figure 4.10), so parameter K is actually 1. Parameters c of the *M/G/c/c* queues $a:m$, $d:m$, and $m:e$ are equal to the corresponding conveyor's capacity.

4.4.2 Queueing Networks with Blocking

Queueing theory studies queues of *infinite* and *finite* capacity. A queue with infinite capacity can always receive a job from another queue or the arrival process. Although infinite capacity allows simpler models, it does not reflect the characteristics of many

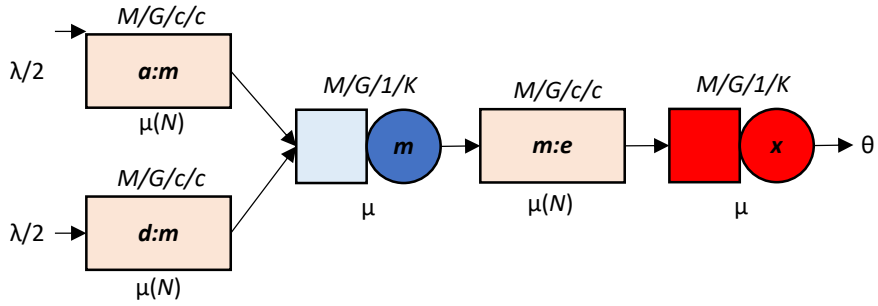


Figure 4.16: Queueing network modeling the merging unit m and its incoming and outgoing conveyors.

real systems. For example, the capacity of MHS conveyors is always finite. In this case, queues with finite capacity can be used. In queueing networks with such queues, if a job is routed to a queue that is currently full, the phenomenon of *blocking* is observed, i.e., the job becomes *blocked* until its queue-receiver is available.

Let us explain the phenomenon of blocking in more detail by example. For that, we consider again the behavior of merge unit m, shown in Figure 4.10. So far, we considered the scenarios where the receiving conveyor $m:e$ is not full, i.e., when it still had some free capacity for receiving TSUs. In reality, this is not always the case. For example, a conveyor can be fully occupied under high load conditions, or when a consequent conveyor/resource is unavailable. Let us consider a scenario when conveyor $m:e$ was fully occupied because some resource x at its end is temporarily unavailable. The corresponding state (at time moment t_6) is shown in Figure 4.17(a), where TSUs *pid*1 and *pid*2 have already been merged onto conveyor $m:e$, but *pid*3 and *pid*4 are still waiting for merging.

The corresponding performance spectrum (with "compound" occurrences) in Figure 4.17(b) describes this scenario in detail. First, *pid*1 and *pid*2 traveled through $a:m$ (occurrences $o^{pid1}_{a:m}$ and $o^{pid2}_{a:m}$), then passed the merge location m (o^{pid1}_{m} and o^{pid2}_{m}), and continued on $m:e$ ($o^{\prime pid1}_{m:e}$ and $o^{\prime pid2}_{m:e}$) until conveyor $m:e$ stopped at time t_6 because of resource x unavailability. We call it a *blockage*. Note that at time t_6, *pid*3 was almost ready to be merged because protective space PrS_2 was almost provided. However, when $m:e$ stopped, conveyor belts of $a:m$, carrying *pid*3 and *pid*4, also stopped as there was no space to hand *pid*3 over. As a result, there was another blockage on the preceding conveyor $a:m$. In such a scenario, we say that a blockage of conveyor $m:e$ *propagated backward* onto conveyor $a:m$. The blockage was observed until time t_7, when resource x resumed, so conveyor $m:e$ could continue. In the performance spectrum, all the occurrences for the time interval $[t_6, t_7]$ are visualized as horizontal lines because there was no progress of the corresponding entities. For example, TSU *pid*3 in occurrence $o''^{pid3}_{a:m}$ was not moving from a to m but standing

still. When conveyor $m:e$ continued its moving, *pid*3 could be merged immediately, and in the following everything went on smoothly.

In this example, we considered the scenario that affected just four TSUs in a tiny system. In real MHSs, a blockage can cause a cascade of other blockages, as well as the propagation of load peaks throughout the system [77, 78].

To model blocking in systems, *queueing networks with blocking* have been extensively studied [17]. Several different blocking types have been proposed in the literature, which describe different types of behaviors on blocking. We assume *Blocking After Service* (BAS) as suggested in [18]. BAS assumes the following blocking sce-

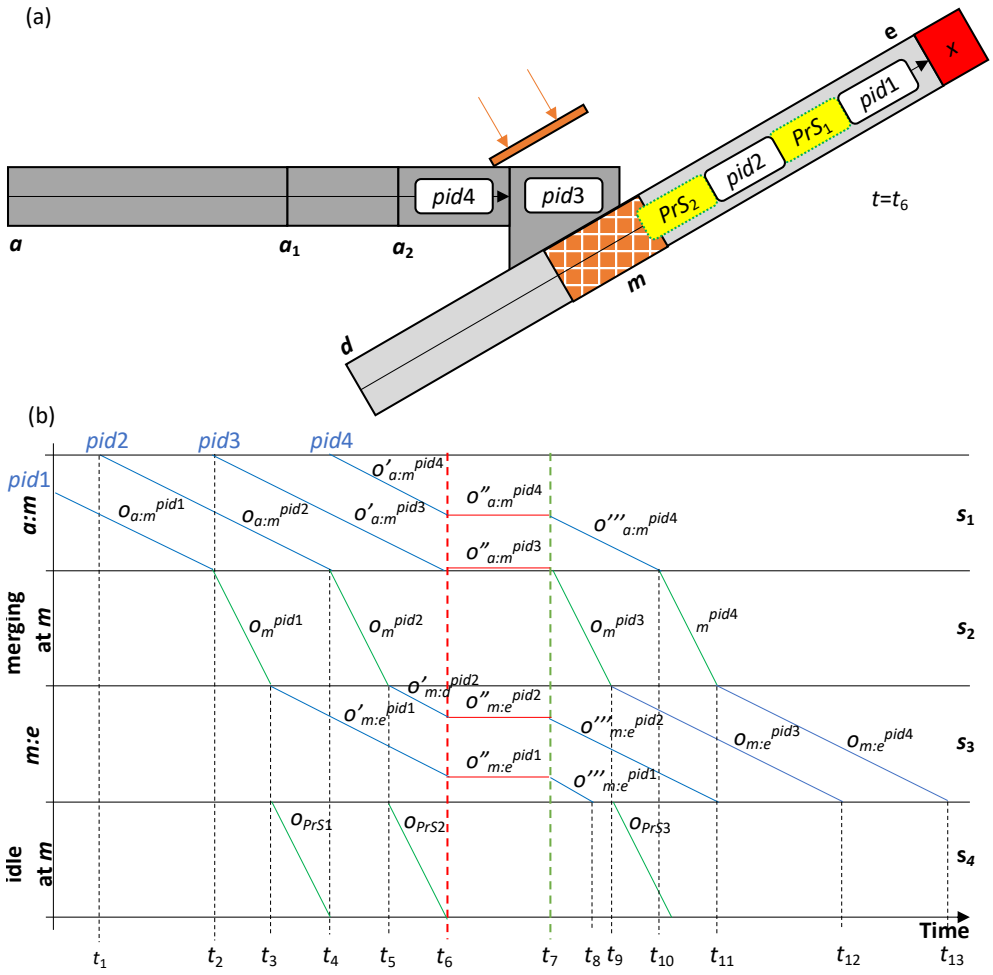


Figure 4.17: Resource x blocks conveyor $d:e$; as a result, $a:m$ gets blocked as well due to inability to hand over *pid*3 onto $a:m$.

nario. When a job on completion of its service at one queue a attempts to enter another full queue b (i.e., the queue b is blocked), it is forced to wait at queue a, occupying the server space, until a space becomes available at destination queue b. As a consequence, server a is forced to stop (i.e., the server is blocked) until its space becomes available for another job waiting in its queue. Server a can resume as soon as the blocking job departures to b.

If we consider the scenario with blockage and assume BAS for the model in Figure 4.16, we can describe it as follows. At time t_6, queue $m:e$ got blocked, so *pid*2 could not depart from resource m (as its protective space was still in the merge location, we considered it occupied). Consequently, queue $a:m$ became blocked. At time t_7, queue $m:e$ got a free capacity element again, so resource m and queue $a:m$ resumed.

The analysis of queueing networks with blocking allows estimating various performance metrics per node, such as utilization, throughput, mean queue length, mean response time, the number of non-empty and non-blocked servers, etc. Additionally, the distribution of various random variables is typically used for the detailed analysis. For example, the distribution of the job number, distribution of job passage time through the node, stationary queue length distribution, i.e., the stationary probability of having n jobs at a node in the steady state (see Section 4.5.1.5) can be used. In principle, by estimating the throughput of a node, **AQ8** can be answered. Using the metrics of individual nodes, average performance metrics for the whole queueing network can be derived, for example, the throughput, mean population, mean residence time, mean passage time from node i to node j, etc. For detailed information about queueing networks with blocking, we refer to [17]. In this thesis, for queueing networks we assume BAS. In the following, we show how various MHS configurations can be modeled using queueing networks with blocking.

4.4.3 Modeling MHSs Using Open Queueing Networks

Previously, we showed how to compose the fundamental building blocks of our choice, i.e., *M/G/1/K* and *M/G/c/c* queues, into an open queueing network to model a merge unit (Figure 4.16). Now, we consider how to model the other typical configurations of conveyors and resources, and finally how to model the whole system.

To model sequentially connected conveyors and resources, *M/G/1/K* and *M/G/c/c* queues can be connected into a series, as shown in Figure 4.18. In the model in Figure 4.18(a), queue *M/G/c/c* is prepended to queue *M/G/1/K*. In this way, for example, a conveyor, moving TSUs *toward* a machine (resource), can be modeled. In Figure 4.18(b), the queues are connected in the opposite order, modeling, for example, a conveyor moving TSUs *from* a machine (resource). In Figure 4.18(c), two *M/G/c/c* are connected. This configuration can be used to describe two connected conveyors with different characteristics, e.g., with different speed-density curves. In

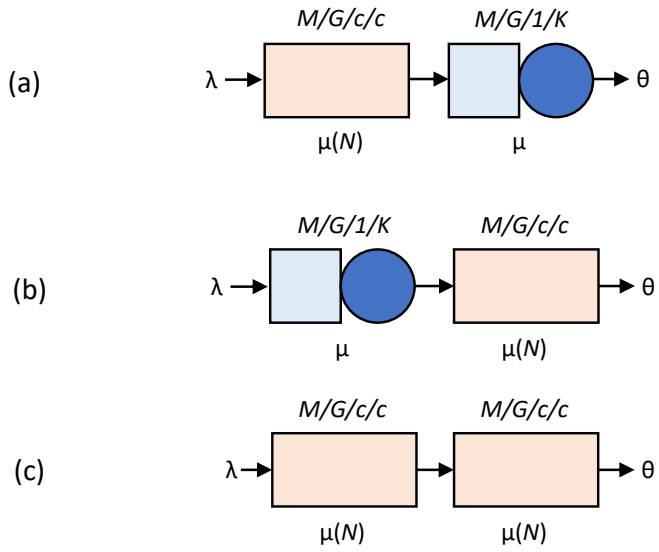


Figure 4.18: Chains of *M/G/1/K* and *M/G/c/c* queues.

principle, such conveyors can be modeled using a single *M/G/c/c* queue. However, introducing two queues can result in a more precise description of the system and more accurate analysis results. Inside each series, routing is deterministic, because each TSU on service completion arrives in the next queue.

Then, the queueing network of a diverting unit model is shown in Figure 4.19. Similarly to the merge unit model in Figure 4.16, the *M/G/1/K* queue, modeling the resource, is connected to the incoming and outgoing conveyors. For this model, a more complicated routing model is needed to support various policies of this unit. A simple routing function would assign equal probability to each possible route, while a more complicated one would receive the overall system state as its argument, for example, to model load balancing.

Finally, we model the whole system we discussed in Chapter 2 (see Figure 6.1) to illustrate how it can be modeled from the building blocks. We assemble it using the models of series and merge units discussed above (Figure 4.20). For that, we assume that one resource is allocated per each check-in counter $a1$ and $a2$. TSUs (bags) arrive in queues $a1$ and $a2$ according to a Poisson process at rate $\lambda/2$ and are served at service rate μ. For the check-in counters, merge unit, and exit unit we assume the exponential service time. On the check-in completion, bags depart to state-dependent queues $a1 : b'$ or $a2 : b$, where their service rate $\mu(N)$ depends on the number of bags in the queue. After the completion of service in queue b, bags go to the state-dependent queue $b : c$ and afterward to queue c, where they leave the system. For this queueing

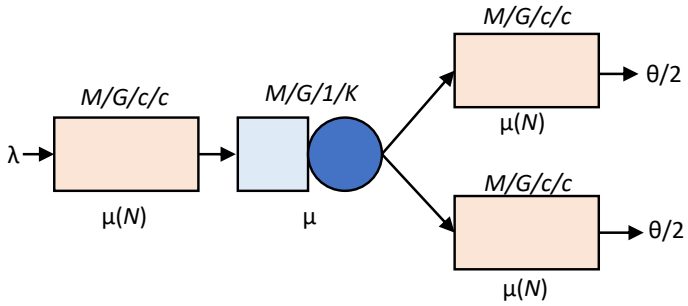


Figure 4.19: Combinations of the *M/G/1/K* and *M/G/c/c* queues model merge (a) and divert (b) units.

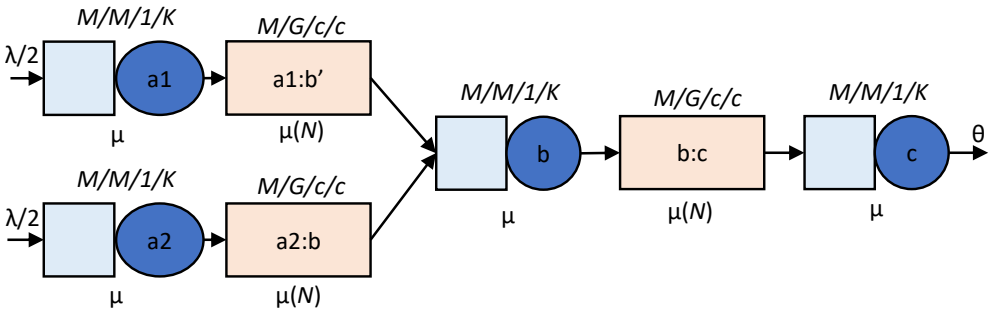


Figure 4.20: Combination of *M/G/1/K* and *M/G/c/c* queues models the system in Figure 6.1(a).

network, we can use deterministic routing, as no alternative routes are possible. We provide the example of a BHS; however, other types of MHSs can be modeled in the same way.

4.4.4 Modeling MHSs using Closed Queueing Networks

For the throughput analysis under constant load conditions, a closed queueing network can be used. In contrast to an open queueing network, there are no external arrivals or departures in a closed queueing network, so some constant number of jobs continually circulates in the queueing network. On service completion, a job is routed to another queue in either a probabilistic or deterministic way similar to open queueing networks, except it cannot leave the system. This way of modeling allows controlling the amount of work.

For example, the conveyors of merge unit m, shown in Figure 4.10, have some finite known capacity, which can be fully utilized under high load. If we assume this condition lasts infinitely long, it means a large (infinite) number of TSUs constantly waiting for arriving to the system. When one TSU exits, another TSU is ready to enter. In closed queueing networks, a job that finished its service is "reused". A closed queueing network, modeling the merge unit, is shown in Figure 4.21, where the output of queue $m:e$ is short-circuited to the input of queues $a:m$ and $d:m$. Closed queueing networks are used for determining the throughput in the out-to-in links, so this "short-circuited" model allows exploring the merge unit under a chosen load.

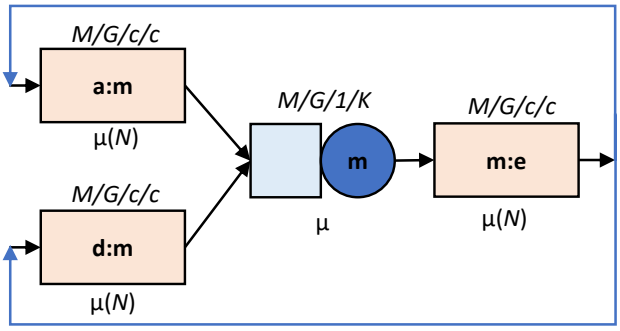


Figure 4.21: Closed queueing network modeling a merge unit.

Similarly, a model of a conveyor series (Figure 4.18), a model of a diverting unit ((Figure 4.19)), and the whole MHS (Figure 4.20) can be short-circuited and converted into a closed queueing network, in order to do the throughput analysis for a chosen amount of work.

In the MHS domain, the throughput analysis of loop sorters is the subject of particular interest, as they are usually the most critical pieces of the whole system. Typically, high-load conditions are the focus of the analyst. The model of a loop sorter of Figure 4.5 is shown in Figure 4.22. In the model, the series of queues $a:b$, b, $b:a$, a model the loop itself, and queues $b:s$ and $s:b$ model the shortcut, where a workstation is installed. The network is short-circuited from queue a to $a:b$.

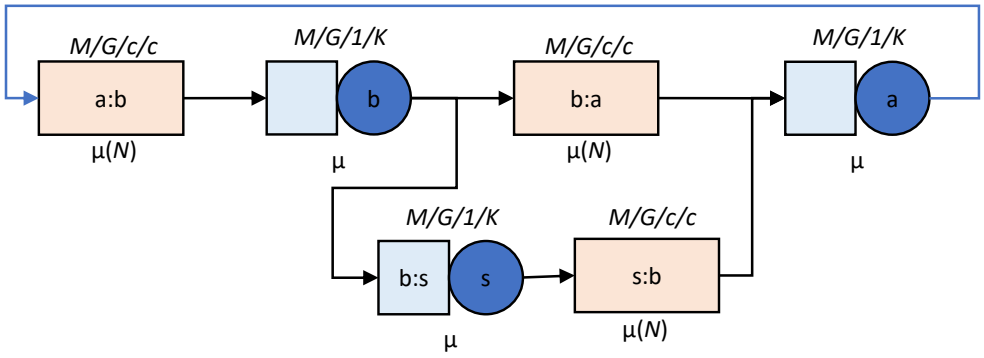


Figure 4.22: Combinations of *M/G/1/K* and *M/G/c/c* queues modeling a loop with a shortcut going through the workstation (see Figure 4.5).

So far, we showed that an MHS can be potentially modeled as a queueing network, by assembling it from the building block. In the next section, we consider the limitations and drawbacks of this approach.

4.5 Analysis Limitations

Previously, we described how MHSs can be modeled with queueing networks, using state-of-the-art approaches. Now, we discuss their limitations for answering our **AQs**. We start by discussing the MHS analysis limitations using queueing networks in general, and the described approach [18] in particular, and conclude by discussing another approach that uses queues for modeling — *queue mining*, aimed to overcome (at least partially) these limitations.

4.5.1 Limitations of Analysis with Queueing Theory

In this section, we discuss multiple factors limiting the analysis of MHSs with queueing theory using analytical methods.

4.5.1.1 Variable Conveyor Capacity Due To TSU-to-TSU Distances

In Section 4.3, we described the merge unit and protective space required between neighboring TSUs on a conveyor, which is required for the correct operation of the system equipment. Depending on the conveyor type, there exist various policies for providing protective space. For example, in Figure 4.23(a) the conveyor has divided

into *trays* of equal size, each tray carries exactly zero or one bag, and at least two trays must be empty between neighboring TSUs (as protective space). While the whole conveyor has eight trays, at most three TSUs can be carried at the same time. If TSUs are placed in a way that more than two trays are empty between two neighboring TSUs, these extra trays are "wasted" unless more TSUs get inserted in between. For example, in Figure 4.23(b) the real capacity is just 66% from the maximum one. If there are only three or four empty trays in between, merging in between is impossible because that would violate the policy. As a result, the actual capacity of the conveyor decreases. In Figure 4.23(c), the conveyor has another policy for protective space: the distance between the *back* sides of one bag and the front side of the directly following bag (on the same conveyor) must be at least δ (δ is shown as a block of black and white bars). This policy takes into account the length of each TSU, thereby maximizing the conveyor utilization. A non-optimal placement of TSUs leads to the decrease of the conveyor capacity in a similar way as for the conveyor in Figure 4.23(a,b).

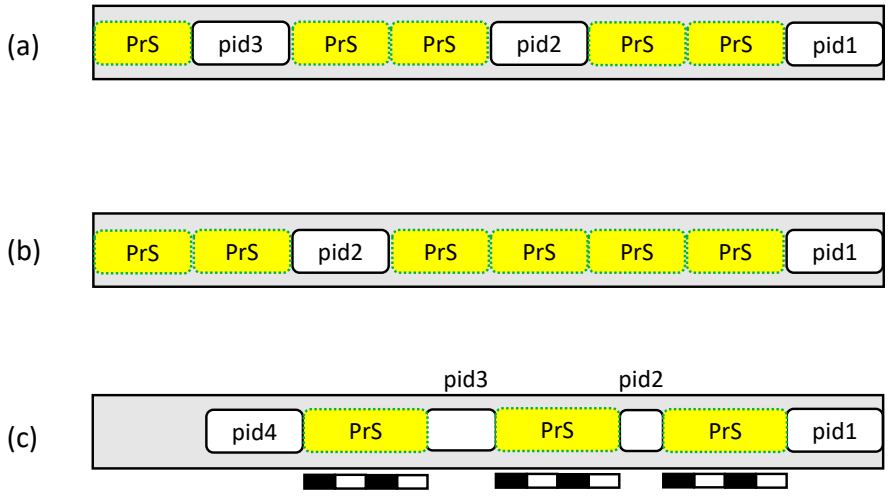


Figure 4.23: Different policies for protective space on MHS conveyors.

As a result, while the maximum capacity can be computed for a conveyor (for example, assuming an average TSU length), its current capacity at each moment is a function of the protective space policy, the length of each TSU placed on a conveyor, and their spatial configuration. Moreover, the policy can be changed dynamically depending on the overall system state, or the system operator's decision. So, a conveyor queue model with some finite capacity c is always an *approximation*, which can significantly differ from the actual one and can cause significant errors during the performance analysis.

4.5.1.2 State-Dependent Queues and Protective Space Policies

Modeling MHS conveyors using state-dependent *M/G/c/c* queues is based on the observation that the TSU travel time depends on the conveyor load. Let us consider it in more detail. Indeed, we observe the dependency between the conveyor load and the travel time, for example, in Figure 4.8 and 4.9. However, does it depend *only* on the load? As we show in Figure 4.14, this effect can be caused by short-term delays occurring when a TSU cannot be smoothly (i.e., without a delay) handed over to the consequent conveyor $d:e$ because this conveyor does not have room at the merge location m. The heavier load on both conveyors, the higher the delay probability for the TSUs being merged. That is, the service time in *M/G/c/c* queue modeling conveyor $a:m$ additionally depends on the state of the consequent queue (conveyor) $d:e$ where TSUs can be routed afterward. In turn, the state of $d:e$ depends on the previous states of $a:m$ because it (at least partially) forms the load and TSU spatial configuration (including protective space) on $d:e$. As a result, the departure process of an *M/G/c/c* queue, modeling an MHS conveyor, actually depends on the previous states of the previous and next queues. So, by taking into account just the speed-density effect, these queues cannot approximate conveyors accurately.

4.5.1.3 Routing

In queueing networks, the routing function usually defines the probabilities of a job to be routed to each queue in the system, and the probability of leaving the system (for open queueing networks). In work we considered [18], stochastic routing is used as well. Nevertheless, let us consider the routing in MHSs in more detail to understand when stochastic routing is the right choice, and when it is not, by the example of the BHS in Figure 1.1.

In this system, the *material handling process* defines the current TSU destination, taking into account

- the time before the actual flight departure,
- TSU location,
- TSU state (e.g., whether it is successfully identified or not),
- overall system state,
- and so on (see Figure 4.24).

Then, to route TSUs to their destinations, a prioritized list of possible routes to each destination is dynamically defined in dynamic routing tables. Normally, a TSU is sent through a route with the highest priority. However, a lower prioritized path can be chosen instead because of the occupancy or unavailability of certain conveyors and/or resources along the way with a higher priority. As a result, routing depends on both system process and system state at each moment of time, i.e., it can be defined as a function $f_{\text{routing}}(n_i, n_j, \textit{process}, \textit{state})$ of the path from node n_i to n_j, system process and

system state. In practice, the routing is typically implemented as a large complicated software component continuously watching the system state in order to compute an optimal route for each TSU in the system.

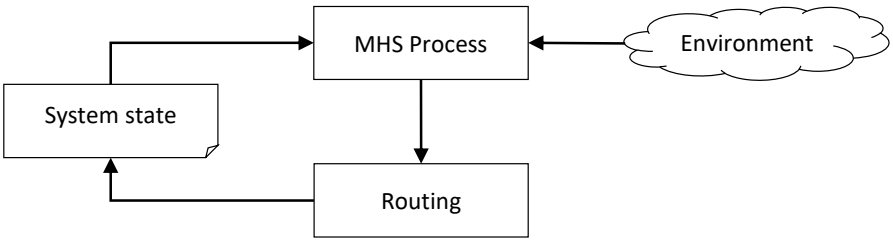


Figure 4.24: Routing in MHSs.

How to model this routing? The answer depends on the analysis goals. For throughput analysis during the system design or extension phase, defining a stochastic routing function that approximates the main behavior under some chosen conditions would be a practical and accurate way to describe routing. However, other analysis types require modeling of outlier behaviors (routing for outliers) as well. For example, post-mortem bottleneck analysis requires a deep understanding of where and why each TSU traveled at each moment of time, especially for outliers. During analysis, the analyst "replays" recorded information about TSU locations in the system, in order to explain how the behavior of many TSUs together resulted in a bottleneck (**AQ3**, **AQ5**, and **AQ6**). For this type of analysis, a stochastic routing function, modeling the general behavior only, would not help because (1) it does not model outliers, and (2) each "turn" on a TSU way has to be explained by facts and not by its probability.

However, defining an accurate stochastic routing function would be extremely useful for answering **AQ8** because it does not require to consider individual TSU paths. In the next section, we discuss whether a queueing network model, assembled from the queue models discussed earlier in this chapter, is capable of modeling the routing of real-world MHSs accurately.

4.5.1.4 The Nature of Conveyor Departure Processes

For a state-dependent *M/G/c/c* queue, modeling an MHS conveyor, the departure process is considered to be Poisson [18, 19, 20, 76]. However, we identified two

reasons that make this approximation accurate only under very limiting conditions. First, in Section 4.5.1.2 we discussed that the queue state depends on its own previous states, as well as on the states of the queue from/where its TSUs are routed, and therefore cannot be considered as a Poisson process. Second, let us consider how the routing affects the nature of the arrival process of queues where TSUs come after a split. For example, in Figure 4.25, TSUs from conveyor (queue) a can be either diverted toward the early bag store through conveyor c, or kept on the loop (b). If we assume that the original process is Poisson process, and if the probabilistic splitting is done in a manner that is independent of the inter-arrival times of the original arrival process, then one can easily show that the processes obtained after splitting are Poisson processes as well. However, as we discussed above, the routing process is not probabilistic. Even if we assume a Poisson process for conveyor A departures, a non-random split makes the arrival processes to those two destination queues non-Poisson. So, the assumption about the Poisson nature of conveyor departures hardly holds for real-world MHSs.

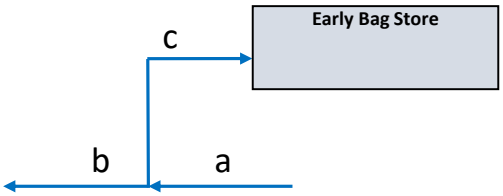


Figure 4.25: TSU flow split.

4.5.1.5 Analysis of Queueing Networks Under Transient and Steady State Conditions

In queueing theory, the *transient* and *steady* states are typically distinguished for a queueing system. The steady state means that the system is in its *equilibrium* state, i.e., the probability $P_n(t)$ of having n jobs at time moment t does not depend on t. Otherwise, the system is in the *transient* state. An illustration of a possible transition from the transient state into a steady state is shown in Figure 4.26, where the system starts operating in a transient state, then eventually reaches the steady state. It is often assumed that after a queue system has been operative for a sufficiently long period of time, probability $P_n(t)$ becomes independent of time t. There also exists

the term *temporary steady state* [79] for referring to a finite, relatively short period of steadiness in the system.

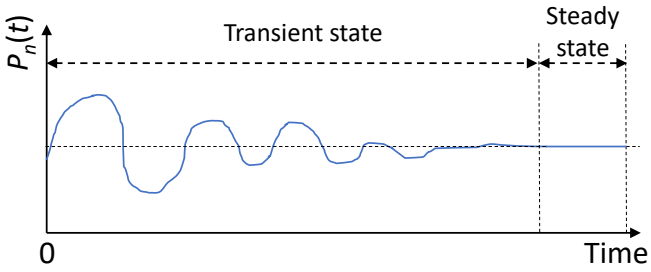


Figure 4.26: Transient and steady states of a queueing system.

Typically, the analysis of queueing systems is done for steady-state conditions for the following reasons [80]:

1. the analysis of a system in equilibrium allows to study its structure, limits and connections between various parameters that characterize it, and
2. many systems come to a steady state soon after their operations start.

When the focus of the analysis is on the system behavior *before* it reaches the steady state, or if the system *never* reaches it, or reaches it in *unknown* period of time, the transient state analysis is needed. However, the problem of the transient analysis is much harder than the problem of the steady-state analysis, so the number of known results and related algorithmic tools is significantly less than for the steady state analysis [81]. As a result, the transient-state analysis is usually done through simulation [82].

However, when is an MHS in the transient state, and when is it in the steady state? Let us consider again the system example shown in Figure 1.1. The material handling process of this system aims to maintain the best performance for the incoming passenger baggage flow, taking into account the flight schedule, occasional flight delays, and correcting operators' actions. The passenger arrival process depends on the flight schedule and information about flight delays, i.e., the probability of having n bags in the system heavily depends on the time. We can assume that the system can be in a temporary steady state from time to time, for example, at night if there are no flights scheduled, but such states are barely interesting for the analyst. So, we assume that MHSs are mostly in the transient state, so their analysis under steady state conditions cannot answer our **AQs**.

Although the approaches we considered before [18, 19, 20] use complicated models of MHS conveyors, they are designed for MHS analysis in the steady state. However, there are *queue mining* approaches, relevant to our **AQ8**, that do not require the system being analyzed to be in a steady state.

4.5.1.6 Limitations of Approximation Techniques and Simulation

So far, we considered the limitations of analytical methods. However, if analytical methods cannot be used, *approximation techniques* can be applied. For example, the individual queues of a queueing network can be analyzed in isolation based on their arrival and departure processes [83], and a network with blocking after service is decomposed into individual nodes with modified arrival processes, to solve it using flow balance and Maximum Entropy techniques in [84]. However, many approximation techniques also have limitations making their application in the MHS domain infeasible, for example:

- analysis under the steady state conditions,
- a small to medium network size,
- a deadlock-free network,
- static routing,
- *first-block-first-served* policy,
- etc.

Moreover, the accuracy of the results cannot be guaranteed [75].

When restrictive assumptions do not allow for system analysis using analytical or approximation methods, *simulation* is usually used. It often allows for reflecting reality better than analytical models or approximation techniques because simulation does not require many simplifying assumptions. For example, the frequent assumptions about Poisson arrivals and exponentially distributed service time are not required for simulation. Additionally, the simulated routing algorithm can closely reflect the real one, even if it is dynamic, as is often the case in MHSs.

On the other hand, the design and implementation of complex processes or systems require deep knowledge of the system and excellent engineering skills. Moreover, it is time-consuming. Speaking from experience obtained at Vanderlande, including simulation of a small-sized BHS that we consider later, simulation is usually used for relatively "small" MHSs to determine a limited set of the equipment parameters during the system design phase and to visualize the system for the customer. However, simulation is infeasible for larger MHSs due to much manpower required for implementing an accurate model, and the high number of computational-intensive experiments required for obtaining comprehensive results. As a result, *emulation systems*, built using real system components, are usually used in practice instead.

4.5.2 Limitations of Analysis with Queue Mining

A particular class of process mining tasks for process performance analysis is called *queue mining* [85]. It establishes a queueing perspective in process mining and considers queues as first-class citizens. Queue mining techniques [34, 85, 86, 87, 88]

are partially grounded in process mining, and partially in queueing theory. For example, their input can be an event log and a process model (given or discovered from the event log). Similarly to the process model, the characteristics of queues, used by a technique, are also derived (mined) from the log. The resulting model can be used to compute the waiting time of cases (i.e., the time interval when a case was waiting in a queue), and the service time of cases (i.e., the time interval when a case was served by a server), at any time moment. These characteristics can be used for revealing bottlenecks, detecting SLA violations, computing descriptive statistics of different performance measures, and calculating various KPIs. For doing visual analytics, a queueing network can be visualized in a graphical form, where each queue and server can be annotated with their state, e.g., the number of waiting cases can be visualized for each queue.

For this thesis, we consider queue mining works to be most relevant for solving **AQ8**. The most recent queue mining work, related to **AQ8**, addresses the case remaining time prediction problem, using so-called *congestion graphs* [87]. This problem can be potentially converted to the problem of aggregate performance KPI prediction. This approach

1. automatically discovers congestion graphs from event logs,
2. extracts features related to congestion in the system for any observed time moment, and
3. enriches existing ML-based approaches with the features related to congestion, derived from congestion graphs.

The congestion graph is built from the process queueing network model. Let us consider this underlying model in more detail. As a queueing network, a *Generalized Jackson Network* (GJN) has been chosen as the most general model in single-server queueing theory [89]. GJN allows renewal arrival processes and independent, identically distributed non-exponential service times. Jobs are assumed to be *indistinguishable*, and routing is assumed to be *Bernoulli distributed*. Each server has only one resource. The state of GJN corresponds to a Markov process, known as the Markov State Representation (MSR) that comprises the following triplet per node [16, 89]:

1. the queue length,
2. the elapsed time since the most recent arrival,
3. the time since the start of the most recent service.

The MSR characterizes congestion at each system queue. The congestion graph is the transformation of a GJN, where each edge corresponds to the GJN queue, and the node labeling is based on the MSR. Given an event log where the activities describe the GJN servers, the MSR can be restored from event logs for any moment of the observed time interval besides the ongoing cases because, for the ongoing cases, the queues where they are waiting are not observed yet. Because of this uncertainty for

the ongoing cases at the time when a prediction is made, the approximation of MSR that does not require knowledge of the next activity of the ongoing cases is used for the node labeling (we refer to [87] for the exact definition). This labeling can be used as the source of congestion-related features for various ML-based time prediction approaches, potentially relevant for answering **AQ8**.

However, what are the limitations of this approach with respect to **AQ8**? As we discussed previously, much less general queueing networks, built form state dependent *M/G/c/c* queues, assuming not general but Poisson processes, cannot provide a good approximation of systems without batching yet has many limitations, caused, for example, by the complexity of routing in MHSs. It comes as no surprise that a more general GJN introduces even more inaccuracy and limitations. We consider the most important ones the following.

- In GJNs, a renewal arrival process is a less strict requirement than a Poisson process in queueing networks of [18] because other than exponential holding times (i.e., intervals between consequent arrivals) are allowed. However, holding times still must be independent, but this is not the case for systems without batching, as we showed in Section 4.5.1.4.
- The MSR does not take into account the TSU-to-TSU distance (protective space), thereby losing crucial information about the TSU state on the conveyors.
- Complicated system routing is not taken into account by assuming Bernoulli distributed routing.

So, while the congestion graph state characterizes congestion in the system over time, it fails to capture the larger part of MHS dynamics and cannot answer **AQ8**.

4.6 Chapter Summary

This chapter addresses the problem of MHS modeling and analysis using queueing theory-based approaches. In the beginning, we distinguished two large classes of MHS — with and without batching and decided to scope this thesis to the former class. Then, we formulated the actual analysis questions, obtained from the study conducted among Vanderlande's process engineers. This study aimed identification of actual problems for solving by any suitable analysis techniques. Using the ability to answer these questions as a criterion for selecting the modeling and analysis technique, we started our exploration from queueing theory. We successfully identified the building blocks queueing theory has for modeling the real-world MHSs and found the relevant work for modeling and analysis of MHSs. We partially verified the finding of these works, including the so-called speed-density effect. However, the MHS behavior is so complex that the existing approaches, despite their sophisticated mathematical models, provide a good approximation only under strict limiting conditions,

thereby making them hardly applicable for the given **AQs**. In contrast, the more simple and more general models behind the queue mining approaches imply even more limitations, therefore having even less applicability. Obtaining the precise model and doing the detailed analysis of a real-world MHS using queueing theory appears to be overwhelmingly hard, and no work allowing that could be found during the literature study.

Nevertheless, the key concepts of the queueing theory-based modeling approaches, such as modeling MHS conveyors as FIFO queues where TSUs wait for the service provided by the servers (MHS resources), and routing between the queues, reflect the nature of MHSs very well. This understanding is the input of crucial importance for reasoning about the behavior of system without batching. In the remainder of this thesis, we use the term *systems with shared resources and queues* instead of *systems without batching* and MHS (when applicable) because it describes this class of systems more accurately. In the next chapter, we review works related to the **AQs**, and in Chapter 6 we adopt these concepts to define the process model of systems with shared resources and queues, using the concept of FIFO queues for modeling conveyors, the concept of servers for modeling resources, and the concept of routing for modeling the material handling process.

Chapter 5

Review of Literature

In this chapter, we review existing works related to answering the **AQs** formulated in Section 4.2 and the methods, outlined in Section 1.3 and proposed in this thesis. Figure 5.1 shows how this chapter is organized.

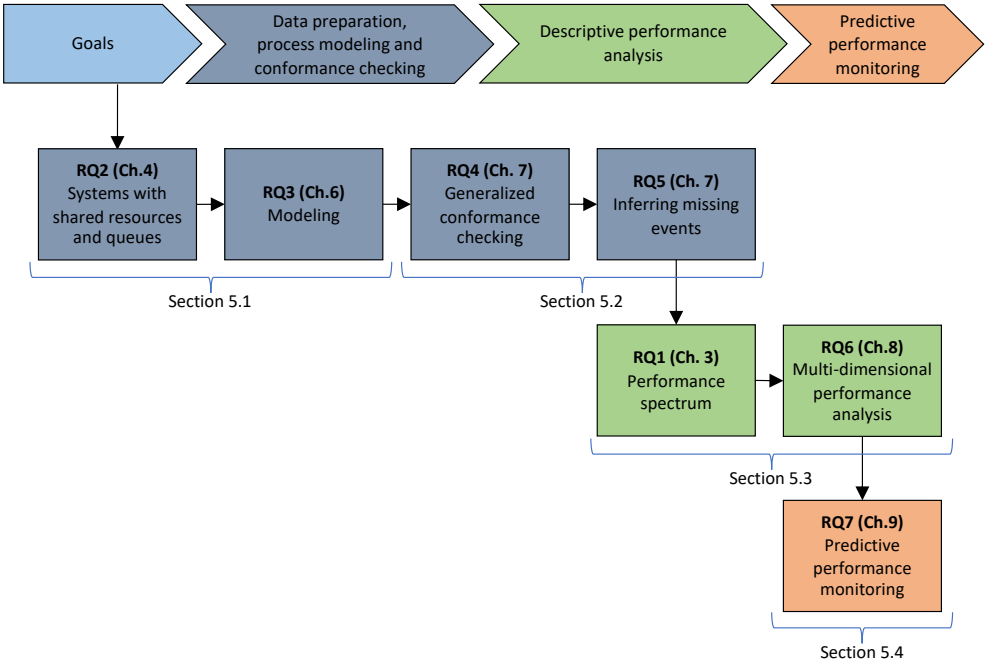


Figure 5.1: Review of literature follows the main flow of this thesis, starting with modeling and going toward predictive performance monitoring.

We consider works related to modeling of material handling processes in Section 5.1 and review works on conformance checking in Section 5.2. We consider works on descriptive process performance analysis and predictive performance monitoring in Section 5.3 and Section 5.4 respectively. Note, we review works related to Chapter 3 and Chapter 8 together in Section 5.3 because both the performance spectrum and method for multi-dimensional performance analysis address the related problems.

5.1 Analytical and Behavior Models

In this section, we consider queueing theory-based analytical models, queue mining approaches, and behavioral process models.

Queueing Models and Networks. There were *deterministic* and *stochastic* queue models developed [17]. In one of the first works on MHS deterministic models [90], the authors considered a system consisting of one loading station, one unloading station, and a conveyor in between, providing insight into a fundamental understanding of a conveyor as the part of a dynamic system, and the way of obtaining variables for a smooth flow between the conveyor endpoints using simulation. In extensions [91, 92, 93] of this work, algorithms calculating these values and conveyor stochastic models were proposed.

Stochastic models are generally more complicated than deterministic ones but reflect reality better. A travel time between the series of stations in a closed loop was considered first as a random exponentially distributed value in work on *discrete* stochastic models [94]. This work was extended for generally distributed values for a system with two stations [95] and for larger systems and multiple job classes [96, 97].

The founding works on *continuous* stochastic models of conveyor systems [98, 99, 100] were extended by the works of other researcher [91, 101, 102, 103] who considered various system topology, the phenomenon of blocking, and so on.

Among recent works related to the MHS domain, analytic solutions for queueing models with *multiple waiting lines*, where either (1) jobs choose servers or (2) servers pick up jobs, were reviewed in [104], where the 2×2 switch can be seen as a model of a combined merge/diverting unit. However, only *isolated* service centers were in the scope, i.e., no queueing networks capable to model an entire MHS were considered. Analytic results for *two-carousel* systems for getting insights for large-scale systems, when analytic studies or simulation are intractable, were presented in [105]. However, this work did not consider the whole system or sorters that have multiple entries and exits. An analytical model to determine the maximum throughput capability of zone picking systems, which is also applicable to BHSs, was proposed in [106]. It extended [107] by additionally considering congestion and blocking at conveyor merges, which can significantly affect the system throughput. The system was mod-

eled as a closed queueing network. The resulting queueing network considers a constant number of jobs (e.g., bags) in the system, thereby targeting its application to the design phase of the MHS lifecycle.

Other works addressed actual problems of business process analysis disregarding the process domain. Thus, advanced queueing models for quantitative analysis of business processes with *many-to-many relations* between activities and resources, and case types with *different performance properties* were proposed in [108], and extended with an approximation for the *parallel construct* in [109]. These works allow translating BPMN models into queueing networks, but assume *block-structured* models [110, 111] (and processes respectively), the FIFO discipline for serving, and Poisson arrivals. All these assumptions do not hold for the MHS considered in this thesis (see Chapter 4). Waiting times in models with multi-server queues where service times are affected by the load were analyzed in [112]. The authors concluded that such differences in service times are inadequate to ignore if they exist in practice, e.g., in call centers.

A dependency between waiting times and load was exploited in the series of works [18, 19, 20, 21, 27]. The authors proposed models based on a *state-dependent M/G/c/c queue* that is capable of modeling a so-called *speed density effect* observed for pedestrian and vehicle traffic [21, 27], when speed drops when the traffic density grows. Further, they showed the same effect in BHSs and applied the same approach [18, 19, 20]. These works are based on the key assumption about Poisson arrivals that does not hold for MHSs considered in this thesis.

Queue Mining. If an analytical solution is unavailable, or its underlying assumptions do not hold, the queue characteristics can be assigned *manually* using a-priori domain knowledge [34, 113], or inferred *automatically* using data recorded from process execution. Thus, the field of *queue mining* [79] uses an event log and process model, either given or automatically discovered from event logs, to derive a queueing network topology, and the characteristics of this queueing network. From the perspective of addressing **AQ1**-**AQ8**, the most relevant work is [87], where a so-called *congestion graph* is built from a generalized Jackson network whose topology is derived from a process model. The states of all the queueing network nodes, annotating the congestion graph nodes, describe the current process state. This information is used as the source of congestion-related features for various ML-based time prediction approaches. In contrast to approaches in [18, 19, 20], this work has fewer assumptions about the distribution of the waiting and service times in the system.

For a detailed discussion about queueing theory- and queue mining-based approaches, we refer to Section 4.5.

Behavioral Models. In the fields of business process management and process mining, *Perti net theory* lays the foundation for process modeling. Over decades, *Petri nets with black tokens*, as well as various model notations based on Petri nets, have been

widely used for describing the control-flow perspective of business processes. For a more detailed discussion, we refer to [1]. However, black tokens do not allow distinguishing the behavior of *multiples instances* (cases) *together* in one run, thereby limiting process analysis that is based on these models. *v-Petri nets* [114] overcome this drawback by introducing *token identities*. Nonetheless, such models usually describe the behavior of a single entity in the control-flow perspective, leaving the behavior of other entities across the process perspectives, as well as their relations, out of scope.

The problem of modeling behaviors of multiple entities across various process perspectives is addressed by various approaches. In the series of works [115, 116, 117], the authors addressed the problem of verification of data and processes. Thus, *DB-nets* were proposed as the marriage of process models and relational databases [115]. For that, a process is modeled with CPNs [63] as a variant of *v*-Petri net with name creation and management, called *v-CPN*, while special *"view" places* serve for retrieving data from the database. Update logic on transitions serves for updating the persistent data. The extension of this work [116] allows getting rid of the view places by using *v*-CPNs with *priorities*. Finally, *catalog nets*, proposed in [117], can be seen as a modeling approach based on "pure" CPNs for the case of read-only persistent data represented as catalogs on places, while read-write data are still partially represented through tokens attributes. Within these approaches, multiple entities can be modeled in a monolithic *v*-CPN, while their relations are modeled in the relational database for [115, 116], and directly in a *v*-CPN in [117]. However, modeling multiple entities in the same net makes it difficult to apply existing analysis techniques to such models.

Process structures [118] integrate relational modeling and behavior modeling but use dedicated behavioral models without existing analysis techniques.

The model of *proclets*, called a *proclet system*, defines one behavioral model (a Petri net) per entity. In proclet systems, entities can interact *asynchronously* via *message exchange* [119] or *synchronously* via *dynamic transition synchronization* [48], allowing to describe *many-to-may relations* with *correlation* and *cardinality* constraints between the entities. When a Petri net with token identifiers, e.g., a *v*-Petri net, is used as a proclet model, same-entity instances are distinguishable in a system run. *Object-centric Petri nets* [120] are the special class of CPNs, which are structured to model the flow and synchronization of different objects (or entities). They correspond to synchronous proclet systems [48] where the synchronization has been materialized in the model structure.

Synchronous proclet systems [48], on the one hand, allow modeling multiple entities and their relations with constraints, but on the other hand, they are too complicated for simpler relations of the process, queues, and resources in MHSs. At the same time, their proclets in the form of *v*-Petri nets cannot describe the token orders in queues.

Instance Spanning Constraints (ISC) and Process Spanning Constraints (PSC) [121] limit ways in which instances of the same or different processes respectively can be

executed. For example, if batching on the execution of a process step *a* is mandatory, it is an ISC. However, if the instances of multiple processes must execute their particular activities simultaneously, it is a PSC. In [121], four categories of ISC/PSC have been considered:

1. simultaneous execution of activities,
2. constrained activity execution,
3. order of activity execution, and
4. non-concurrent execution of activities,

and algorithms for their discovery have been suggested. Formalization of these ISCs and PSCs using proclet systems [119] and CPNs [49] has been proposed in [122].

Interestingly, many phenomena caused by ISCs [121] (e.g., different types of batching) can be observed in performance spectra as instances of various performance patterns (see Chapter 3). However, these instances do not necessarily correspond to ISCs, they can be also ISC exceptions [123] or just arbitrarily happening behaviors.

In MHSs, spanning constraints of the *constrained activity execution* category, such as batching and FIFO order, are usually observed. However, are they instance or process spanning constraints? As we showed in Chapter 4, these phenomena happen because of the behavior of resources and queues. If we consider them as separate processes in terms of [121], the constraints they introduce are PSCs. However, describing these constraints through constrained activity execution may be overwhelmingly difficult. Instead, if the MHS resources and queues are modeled explicitly as separate entities, simpler PSCs — simultaneous execution of activities — can be sufficient to limit behaviors.

PQR-Systems. All things considered, we conclude that none of the existing models can serve as a "backbone" for the methods of this thesis, capable of answering the **AQs**. Having said that, we determine that

- queues, servers, and routing functions of queueing networks [17] can perfectly represent MHS conveyors, machines, and system layouts, and
- synchronous proclet systems [48] can be seen as a model allowing some superset of capabilities required for modeling MHSs.

In Chapter 6, we model MHSs as a dedicated synchronous proclet system, called *PQR-system*, which describes the process, queues, resources, and their relations and synchronization. In the PQR-system, we limit relations cardinality to one-to-one, ensure that correlation constraints are never violated, and extend *ν*-Petri nets to nets with a richer syntax, which is a sub-set of CPN syntax [49] (similarly to *ν*-CPNs of [115, 116]) for modeling queues. "Embedding" queues into a model is similar, to some extent, to congestion graphs [87] of queue mining [79]. However, introducing PQR-systems, we focus mostly on the queueing behavior and *designed* performance characteristics rather than on inferring actual queue performance characteristics from

data, in contrast to queue mining. Additionally, we define the replay semantics [1] for the PQR-systems. It allows addressing the problem of relating data to the model, which we consider next.

5.2 Conformance Checking and Log Repair

As we learned at Vanderlande, event data recorded from MHSs are usually incomplete because of the logging approaches widely used in the industry, while a process model may become incorrect due to concept drift, i.e., changing system parameters over time. As a result, a classical assumption about a fully trusted event log or process model [24] does not work. To deal with it, we adopt the concept of *generalized conformance checking* that does not require trust in either log or data but unites the three tasks of model repair, log repair, and conformance checking under a common roof [44]. In the following, we consider works related to these tasks.

Conformance Checking. *Conformance checking* is one of the main types of process mining [1, 24], and considered in many works. *Alignment* of an event log trace with some "correct" process trace was suggested in [124] but it cannot be used to align a trace with a process model. In contrast, a precise relationship between log events and elements of a Petri net model is established through *replaying* event log traces over the model by computing alignments between them in the founding work [43] and its many extensions [24]. Conformance checking with uncertainty via satisfiability modulo theories [125] is actual when event logs contain uncertain data [126, 127, 128]. However, the direct application of these approaches is limited to black-token Petri nets and the control-flow process perspective that they describe, i.e., they do not address the challenge of conformance checking of multiple process perspectives [24], e.g., the process, queue, and resource ones described by the PQR-system.

In existing works, this challenge is addressed in various ways. Data-aware behavioral compliance checking is addressed in [129, 130] by applying compliance rules *per trace* and not per event log. An approach aligning an event log with a so-called *Petri net with data*, that, besides the control-flow perspective, takes into account the *data*, *time*, and *resource* perspectives, is considered in [52], and a way to obtain extended conformance checking diagnostics for its result was considered in [53]. In [131] authors extended [52] and treated all the process perspectives equally rather than assuming the control flow as the most important. The aforementioned works consider "flat" Petri nets, i.e., not-nested models without any hierarchy. The problems of conformance checking of nested Petri nets are addressed in [132], while conformance checking of artifact-centric process models is addressed in [133, 134] through behavioral conformance checking of artifact models *per model*, and interaction conformance checking of the interactions of artifact instances described by these models.

While existing works address the problems of conformance checking actual for conformance checking over PQR-systems, none of them can be applied directly because the Q- and R-proclets use a more complex language than Petri nets with data [52] or artifact models [133, 134]. However, the way how the time and data perspectives are modeled and checked in Petri nets with data, and how the interactions of different artifacts (modeled by different models) are checked in [133, 134] are indeed relevant for the problem of conformance checking of PQR-systems. We build on these approaches by proposing their extensions and combining them for doing conformance checking of PQR-systems.

Log Repair. In all operational processes in logistics, manufacturing, healthcare, education, and so on, complete and precise event data, including information about workload and resource utilization, are highly valuable since they allow for process mining techniques uncovering compliance and performance problems. Event data can be used for replaying processes on top of process models [1], predicting process behavior [32], and so on. All these techniques rely on the completeness and correctness of given event data. It makes the problem of event log (data) repair actual in the MHS domain, where data are usually incomplete.

Various approaches exist for dealing with incomplete data of processes with non-isolated cases that compete for scarce resources. In call-center processes, thoroughly studied in [135], queueing theory models can be used for load predictions under assumptions about distributions of unobserved parameters, such as customer patience duration [136] while assuming high load snapshot principle predictors show better accuracy [85]. For time predictions in congested systems, the required features are extracted using congestion graphs [87] mined using queuing theory.

Techniques to repair, clean, and restore event data before analysis have been suggested in other works. An extensive taxonomy of quality issue patterns in event logs is presented in [137]. Repairing inadvertent time intervals [137] is considered in [138]. In [139] resource availability calendars are retrieved from event logs without the use of a process model, but assuming the presence of *start* and *complete* life-cycle transitions, as well as case arrival time in the event log. Using a process model, classical trace alignment algorithms [24] restore missing events but do not restore their timestamps. The authors concluded (see [24], p. 262) that incorporating other dimensions, e.g., resources, for multi-perspective trace alignment and conformance checking is an important challenge for the near future.

Data models for event data over multiple entities have been studied extensively in three forms. One type of event log describes entities just as a sequence (or collection) of events [73, 140] where each event carries multiple entity identifier attributes, possibly even having multiple entity identifier values. Behavioral analysis requires extracting a trace per entity, thereby constructing a set of related sequential event logs [140, 141]. Other works construct a partial order over all events using graphs: nodes are events, edges describe when two events directly precede/-

follow each other and are typed with the entity for which this relation was observed [142, 143, 144, 145].

Our log repair approach, proposed in Section 7.3.8, contributes to the problem of reconstructing the behavior of cases and limited shared resources for which the cases compete. Given a PQR-system and event table (see Section 2.4), we reconstruct missing events through classical trace alignments over the proclet of the PQR-system, representing the control flow. The dynamic synchronization [48] of all proclets in the PQR-system allows inferring how and when sequential traces of resource entities must have traversed over the process steps, which we express as a linear programming problem [146] to compute timestamp intervals for the reconstructed events. For the construction of the linear program, we make extensive use of the partial ordering of events.

Enabling conformance checking over PQR-systems, and log repair for inferring missing events with timestamp information, we enable generalized conformance checking [44] based on these approaches.

5.3 Descriptive Performance Analysis

The process performance analysis from *event data* can be divided into descriptive, predictive, and visual analysis, we refer to [36] for an extensive discussion. We summarize descriptive performance analysis and visual analysis here, and predictive performance analysis in the next section.

Descriptive Performance Analysis and Visual Analytics. Commonly, the process performance is described by *enhancing* a given or discovered process model with information about activity durations (nodes in a model), or the waiting time between activities (edges in a model) [1]. In the visualization, each node and edge can be annotated with aggregate performance measures, e.g., average, minimal, or maximal value, for all the cases passing through this node or edge, as illustrated in Figure 3.1(a). The performance visualization on a model is more accurate if the discovery algorithm takes the underlying performance information into account [147, 148]. A non-fitting log can be aligned to a model to visualize the performance information [43]. However, a more detailed visualization of the performance characteristics requires more dimensions. Wynn et al. [68] plotted different process variants (with different performances) into a 3-dimensional space "above" the process model. Transition system discovery allows to split occurrences of an activity based on its context and visualize performance in each context separately [67].

Techniques for *describing the performance of all cases* construct simpler models through stronger aggregation [67]. Also, the recent temporal network representation abstracts non-stationary changes in the performance over time [148].

The representational bias of models, assumptions, and aggregation can be avoided through visualization and visual analytics [72]. *Dotted Chart* [149] plots all events per case (y-axis) over time (x-axis) allowing us to observe arrival rates and seasonal patterns over time. Story graphs [150] plot a case as a poly-line in a plane of event types (y-axis) and time (x-axis) allowing to observe the patterns of similar cases with respect to the behavior and performance over time but convolutes quickly with many crossing lines.

Performance Spectra. In Chapter 3 we proposed the *performance spectrum*, a model and visualization that avoids the problems of [150] by describing the *performance of each process step* without assumptions about the data (except having a log of discrete events). The visualization *reveals where a process violates typical assumptions about the performance* such as non-stationarity or cases influencing each other. Additionally, we provided a *taxonomy* that describes these phenomena (see Section 3.3).

Process Model-Less and Model-Based Descriptive Performance Analysis. Building on phenomena revealed by the performance spectrum [45, 46, 151] (see Chapter 3) and described in the performance patterns taxonomy [45] (see Section 3.3.2), a method for detecting system-level behavior leading to dynamic bottlenecks in MHSs is proposed in [77] for patterns describing congestion and higher load (*blocking* and *high-load system-level events* in terms of [77]). This method is extended into a framework for detecting system-level behaviors of high-level events of any type, configured for the framework, for "classical" business processes in [78].

However, the method of [77], and the framework of [78] are *model-less* approaches, i.e., they cannot benefit from domain knowledge captured by a process model. It results in several drawbacks, among which the incapability of root-cause analysis of undesirable performance pattern instances (e.g., bottlenecks) is the most important. We refer to Section 8.1 for a detailed discussion.

In Chapter 8, we contribute to the problem of root cause process performance analysis by "marring" the performance spectrum and PQR-system. First, we introduce the performance spectra of the queue and resource dimensions described by PQR-systems, and show how the synchronization of the PQR-system proclets can be explored in the spectra of the process, queue and resource dimensions. Then, we propose a method for

1. tracking developments of blockage and high load instances in the performance spectra down toward their origins, i.e., initial pattern instances, using knowledge about the process captured in the PQR-system, and
2. doing their root cause analysis with the queue and resource performance spectra.

Next, we review works on predictive performance analysis.

5.4 Predictive Performance Analysis

For business processes, the remaining processing time for a case can be predicted by regression models [70] or by decorating a transition system with remaining time [69], prior trace clustering improves the prediction [152]. In [36], a Naive Bayes classifier predicts the future path of a single running case and a regression model predicts the transition durations on this path. The likelihood of future activities can be predicted using Markovian models [153], but without providing any time predictions. The completion time of the next activity can be predicted by training an LSTM neural network [39], or by learning process models with arbitrary probability density functions for time delays through non-parametric regression from event logs [154] that can also be used for learning simulation models to predict performance [71, 155]. Competing for shared resources can be taken into account through simulation models or with queuing models [156]. However, using only features of a single case, these models cannot predict PPIs for non-isolated cases. Estimating an aggregate PPI through the outcome of individual cases [157] cannot be used for non-mandatory outcomes of non-isolated cases. Prediction of the remaining time for a single case in processes with non-isolated cases is addressed in [35], where the intra-case features of a running case of interest are coupled with inter-case features of concurrently running cases that are "close" to the case of interest in terms of control-flow and temporal distances. However, in processes with tightly coupled dynamics such as MHS processes, cases influence each other, e.g., congestions propagate through the system, and resource problems affect groups of cases, impacting the overall system performance. In [158], context data are considered as a possibly impactful factor for process outcomes in logistics and manufacturing.

Among MHSs, BHSs are studied extensively. In the BHS domain, relationships between some bag- and system-related properties can be learned by feedforward neural network models [159], but the results reported as just acceptable, even for the fully controllable environment of a simulation model. A risk of baggage mishandling can be predicted with an aggregate probabilistic flow graph as a function of travel durations between system locations [160], while dynamic routing is not supported. Problem-oriented *simulation models* allow identifying of bottlenecks and critical operations for inbound baggage handling [22] and learning dependencies between security policies and time characteristics of manual baggage screening [23]. In [161] the overview of various simulation-based performance prediction techniques for baggage screening is provided. While these simulation models are precise, their design requires in-depth knowledge of a system design and proved to be time-consuming.

Our method, proposed in Chapter 9, contributes to the problem of predicting aggregate PPIs for systems with shared resources and queues. We capture inter-case dependencies by leveraging the performance spectrum and learning the unknown system behavior from performance-related features of the performance spectrum.

Chapter 6

Modeling Systems with Shared Resources and Queues

In Chapter 4, we showed how systems with shared resources and queues, i.e., a class of systems without batching we focus on, can be considered as queues (conveyors) and servers (resources) serving jobs (TSUs) interconnected through a routing function, defined according to the system layout. In Chapter 4, we also identified the limitations of queueing networks related to our analysis questions **AQ1**-**AQ8**, provided in Section 4.2. In this chapter, we propose a model that does not have these limitations. For that, we first formulate the challenges for modeling systems with shared resources and queues. Then, we introduce the main concepts for their modeling, based on *v*-Petri nets [114, 162, 163] and synchronous proclet systems [48, 119]. Finally, we introduce Process-Queue-Resource systems (PQR-systems) as a special class of systems with shared resources and queues that allows the performance analysis of individual MHS entities, as the analysis questions **AQ1**-**AQ8** require. We provide both PQR-systems formal definition and semantics. Additionally, we introduce a simulation model of a BHS and the corresponding PQR-system to demonstrate modeling concepts. Further in the thesis, we use the PQR-system as input for our methods and the simulation model as a source of synthetic event data.

6.1 Challenges for Modeling Systems with Shared Resources and Queues

In this section, we discuss how the interactions of various MHS entities result in complicated system behavior, and in turn, what modeling challenges rise because of this complexity. To provide more concrete examples, we mostly consider BHSs and

the bags they handle, assuming that our reasoning is also applicable to the other types of MHSs.

6.1.1 Complicated Dynamics of BHSs

As we showed in Chapter 4, BHSs can be described and analyzed in terms of *entities* of the following types.

1. *Bags* (jobs in the queueing theory terms) that travel throughout the system and visit locations where various process steps are executed.
2. *Resources* (servers in the queueing theory terms) that perform process steps. While a machine's resource is "wired" to the machine performing a particular process step, e.g., to a scanner performing baggage security screening, a workstation is operated by a *worker* who can also do other process steps at other workstations.
3. *Conveyors* (queues in the queueing theory terms) that move bags throughout the system via its resources, e.g., an accumulating conveyor can move bags from a divert unit of a sorting loop to a security scanner.

Each type of these entities serves a different purpose and therefore behaves differently. In the system, each separate entity follows *its own behavioral rules* to fulfill its purpose but must synchronize interactions of its "interfaces" with the other entities. Let us show that for all the entity types, using a BHS shown in Figure 1.1 as an example.

Bags. Each bag has to reach its final destination on time, undertaking mandatory process steps on the way. Initially, each bag is handed over to the system, then it goes along one of the multiple possible paths leading to its final destination. Depending on the bag's current state, the sequence of process steps to be completed can change on the way. For example, if a bag could not be identified automatically, it must undergo the manual identification process before resuming the "standard" sequence of process steps. Each bag is processed independently, i.e., its sequence of process steps to be completed does not depend on the states of the other bags in the system, transportation means (e.g., accumulating or non-accumulating conveyors), and so on. Finally, a bag journey ends by exiting the system.

Resources. A system resource receives bags from one or multiple incoming conveyors (its input), performs a process step, and hands the bag over to one of the outgoing conveyors (its output). Disregarding the input/output configuration, each resource has the following life cycle:

1. it is *idle* initially,
2. it receives a bag through its input and becomes *busy*,

3. it performs a process step in some *service time*, needed, for example, to transport a bag through the resource location/equipment,
4. it becomes free again when the bag is handed over to the subsequent conveyor. However, as we described in Section 4.1.1, some protective space is typically required between neighboring bags. Because of that, a transition from state *busy* to state *idle* typically takes some non-zero time, which we call the *resource waiting time* t_{wR}. The resource waiting time allows the processed bag to travel a bit further from the resource location, thereby making protective space in between.

In order to maintain this cycle for each individual bag,

1. a resource naturally synchronizes at least twice with a bag: when it takes a bag in and starts working on it, and when it finishes working on a bag and moves it out/forward,
2. all resources in this thesis are assumed to have the capacity limit of one, i.e., when a resource synchronizes with a bag and takes it in, it is busy and cannot take in another bag (i.e., it always synchronizes with *one bag at once* between the start and complete.

This life cycle is valid for both machines and workers. However, there are two aspects that are different:

- the service time of machines is usually similar or the same for each bag, while for workers it depends on their skills, physical state, and so on.
- workers, in contrast to machines, are not permanently attached to a single location and/or process step but can perform different steps at different locations.

Conveyors. Each conveyor has exactly one input and one output. It takes in bags handed over by a resource on one end, then moves them toward the other end, and hands them over to another resource. On a conveyor, the bags stay in the order they entered it, and need non-zero time to pass through. The conveyor properties, such as the speed, capacity, and whether it is accumulating or not, vary. Similarly to resources, conveyors do not operate in isolation but synchronize with the other system entities as follows.

1. Conveyors continuously synchronize with bags: every bag that is put onto a conveyor is tied to this conveyor (i.e., both are synchronized in their movement). The conveyor belt movement naturally determines how the bags on it move forward, so we observe *conveyor to bag influence*.
2. At the end, i.e., while handing the bag over, whether the bag can be handed over impacts the conveyor, because the conveyor may have to stop (see Chapter 4), i.e., we observe *bag to conveyor influence*.

6.1.2 BHS Modeling Challenges

As Chapter 4 shows, to answer our **AQs** (see Section 4.2), a BHS model needs

1. to distinguish the individual behavior of multiple bags, and
2. to describe how bags compete for shared resources (conveyors, machines).

It is needed because most of our **AQs** require the behavioral analysis of either one or multiple *individual* bags that synchronize with various *individual* resources and conveyors. For example, the bottleneck root-cause analysis (see **AQ3** in Section 4.2) requires the analysis of load to individual conveyors and resources, to understand when, where, and why particular bags caused periods of higher load, congestion, and eventually bottlenecks. For such analysis, the behavior of bags, queues, and resources have to be described as distinguishable entities. Moreover, because the entities synchronize (as described in Section 6.1.1), the model needs to describe the interaction of these entities as well.

These requirements make the direct use of classical process models, such as Petri nets, difficult, so more sophisticated process modeling techniques are needed. To describe the behavior of entities of different types, we adopt the model of synchronous proclets [48], which

1. describes different entities as separate *proclets* (e.g., separate Petri nets), and
2. describes how the proclets synchronize using *synchronous channels* between them.

Further, to distinguish individual entities of the same entity type modeled by the same proclet, we use the ideas of unique identifiers of *v*-Petri nets [114, 162, 163]. Using these ideas together, we tailor a dedicated synchronous proclet system, which we call a *PQR-system*, capable to describe BHS entities of these types (bags, resources, queues) with multiple distinguishable instances of the same entity (e.g., multiple different bags), and their synchronization on the level of individual entities.

In the following, in Section 6.2, we recall basic ideas of *v*-Petri nets and synchronous proclets, and illustrate how they can be used to describe systems with shared resources and queues by defining a proclet *P* for processing TSUs according to the system layout, proclets *Q* for queues (conveyors), and proclets *R* for resources. The synchronization of these proclets then allows for describing and decomposing the system behavior from the perspective of all three entity types. In Section 6.3, we design a PQR-system by describing the Q- and R-proclets using a subset of CPNs, and explain PQR-system replay semantics. Finally, in Sections 6.4 and 6.5, we define the PQR-system and its semantics formally.

6.2 Concepts for Modeling Systems with Shared Resources and Queues

Before we present the formal definition of PQR-systems, we first introduce the necessary concepts informally. For that, we introduce

- how to describe different uniquely identifiable entities of systems with shared resources and queues, and
- how to describe the behavior of these entities using the CPN concepts we recap in Chapter 2

Later in this chapter, we build on these concepts to define the PQR-system and its semantics.

6.2.1 Distinguishing Same-Type Entities by Token Identifiers

A classical process model N_E (Section 2.2) describes the processing or behavior of instances of a single entity E, e.g., transportation of a bag in a BHS. This can be formalized as a single labeled net $N_E = (P_E, T_E, F_E, \ell_E)$. The behavior of *one instance* of E, e.g., a concrete bag, then follows from consuming and producing "black" tokens in N_E, which defines an occurrence sequence, i.e., a run of N_E (see Definition 2.4) [164]. Black tokens in such nets are indistinguishable. In Section 2.2, we model a toy BHS that accepts bags at check-in counters $a1$ and $a2$, merges bag flows from these counters at location b, and finally delivers bags at lateral c (Figure 2.1(a)). It is modeled as a labeled Petri net in Figure 2.1(b). Let us consider two black tokens on place $p2$ in the net. It is impossible to say which token corresponds to which bag in the system (Figure 2.1(a)). So process models, based on nets with black tokens, significantly limit the capabilities of analysis of individual instances of an entity when multiple instances are present in a run.

Petri nets with token identities [114, 162, 163] are a way to describe the behavior of *multiples instances* of an entity *together in one* run, e.g., to describe the handling of multiple bags together in a BHS. We can express the ideas of Petri nets with token identifiers in a CPN where each place has a color set of token identifiers, as follows.

The CPN model N_{BHS} in Figure 6.1(b) extends the model in Figure 2.1(b). It describes the handling of individual entities of type *bag*. In the model, each arc has expression `pid`, where variable `pid` has type `ID` (defined as `string`) that models the set $\mathscr{I}$ of identifiers (see Definition 2.7). The initial transitions without pre-places are always enabled. A *new instance* of the entity, described by such a CPN, is created by generating a new identifier $id \in \mathscr{I}$ by a transition with an empty preset. In our example, when transition $t1$ or $t4$ occurs, a new identifier value $id \in \mathscr{I}$ is bound to variable `pid` on arc $(t1, p1)$ or $(t4, p3)$ respectively. For our work, we assume that in

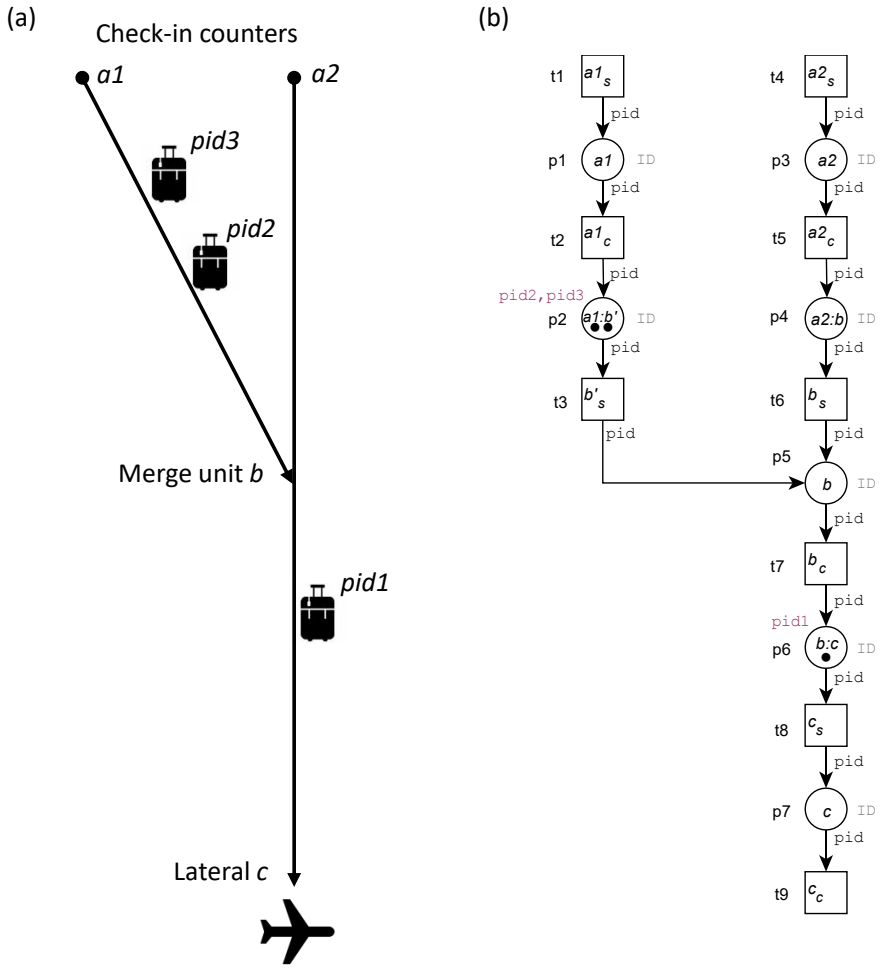


Figure 6.1: MFD (a) and CPN model (b) of a BHS.

a CPN for any transition with the empty preset and outgoing arcs with variable `pid`, variable `pid` is always bound to a new value of the identifier $id \in \mathscr{I}$ never seen before.

Further, net N_{BHS} behaves as usual. The instance *id* advances by an *occurrence* of an enabled transition of the net in that instance *id*: any transition $t \in N_{BHS}$ is *enabled* in instance *id* when each pre-place of t contains an *id* token; firing t in instance *id* then consumes and produces *id* tokens as usual. The distribution of identifier tokens over the places of N_{BHS}, which also have color set `ID`, describes the state of each instance

id. The state of the entire entity N_{BHS} for all its instances is then a distribution of *multiple* tokens from $\mathscr{I}$ over the places of N_{BHS}. That is the state of the entity is described by its current marking. In our example, the current marking (the state of the entity) is:

- $m_{current}(p1) = m(p3) = m(p4) = m(p5) = m(p7) = []$,
- $m_{current}(p2) = [pid2, pid3]$,
- $m_{current}(p6) = [pid1]$.

From marking $m_{current}(p2) = [pid2, pid3]$, we see that bags $pid2$ and $pid3$ are in state $p2$ (at location $p2$), while bag $pid1$ is in state $p6$ (at location $p6$). Note, as a result of assigning unique identifiers to the tokens, the tokens on place $p2$ become distinguishable, in contrast to Figure 2.1(b).

The partially ordered run (Definition 2.4), leading to marking $m_{current}$, is shown in Figure 6.2, where we can see that the events and conditions of each instance are completely independent of the events and conditions of the other instances. The run describes a partial order of events.

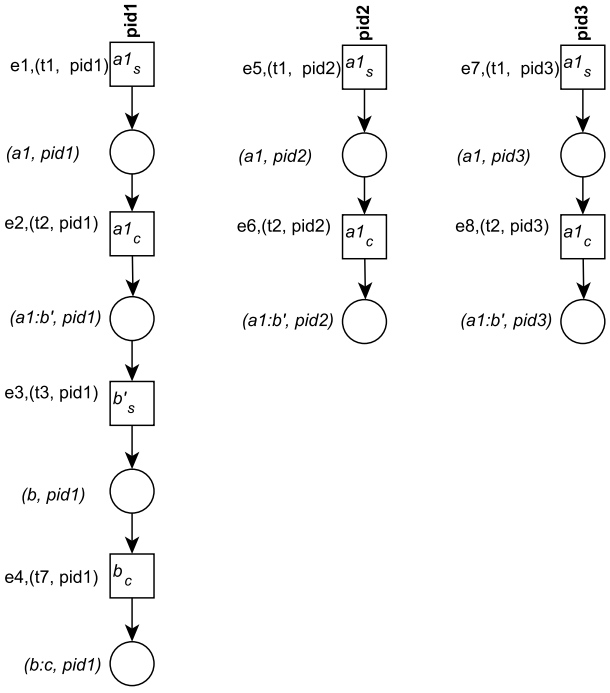


Figure 6.2: Run of the CPN of Figure 6.1(b).

6.2.2 Distinguishing Multiple Interacting Entities by Synchronous Proclets

The previous section showed how to distinguish the instances of the same entity using tokens with identifiers. It also showed that within a single entity, instances are completely independent. That is not true for BHSs. For example, bags interact through shared resources and conveyors. So, to describe the behavior of a BHS more adequately, entities of all three types have to be modeled, as well as their synchronization. Using *synchronous proclet systems* [48], different entities can be modeled as different nets that are called *proclets*, and their synchronization can be described through so-called *synchronization channels*. In the following, we explain the ideas and concepts of synchronous proclet systems as they were introduced in [48] along the BHS example. Later in Section 6.3, we point out how this model needs to be extended to fully describe BHS dynamics.

Bags. This entity type refers to the process of bag handling according to the system layout, i.e., entering the system, visiting various locations for performing process steps (by resources), and leaving the system at the assigned final destination. In our example of a simple BHS, we have two start locations $a1$ and $a2$ where bags can enter the system, one possible path from each start location ($a1 \rightarrow b \rightarrow c$ and $a2 \rightarrow b \rightarrow c$ respectively), and the only exit c. The CPN model of this entity is already described in a detailed manner in Section 6.2.1 and shown in Figure 6.1(b). In the proclet system, this CPN becomes a proclet that we call a *Process proclet* (or a *P-proclet*). There is always just one P-proclet in the proclet system, i.e., only this proclet describes the bag handling process (entity *bag*).

Resources. The second entity type is *resource*. In our example, we have:

1. two check-in counters (workstations) with two resources (workers) $rid1$ and $rid2$ (Figure 6.3(a)),
2. one merge unit with a single resource $rid3$,
3. one lateral (exit) with a single resource $rid4$.

In the following, we show that we model the life cycle of each resource identically.

First, we consider the check-in counters. At their locations ($a1$ and $a2$), the system has two operators (workers), and each operator can work at any counter, for example, to replace the other operator who has a break. We describe them as entity *R-operators* (*R* stands for *Resources*) of type resource, where each operator is an instance of this entity, modeled through a unique resource identifier token. We call proclets that model resource entities *R-proclets*. The model of entity *R-operators* is shown in Figure 6.4(a). Two resources are represented (and easily distinguishable) as tokens $rid1$ and $rid2$ on place $idle^{ops}$. Each of them has the following life cycle.

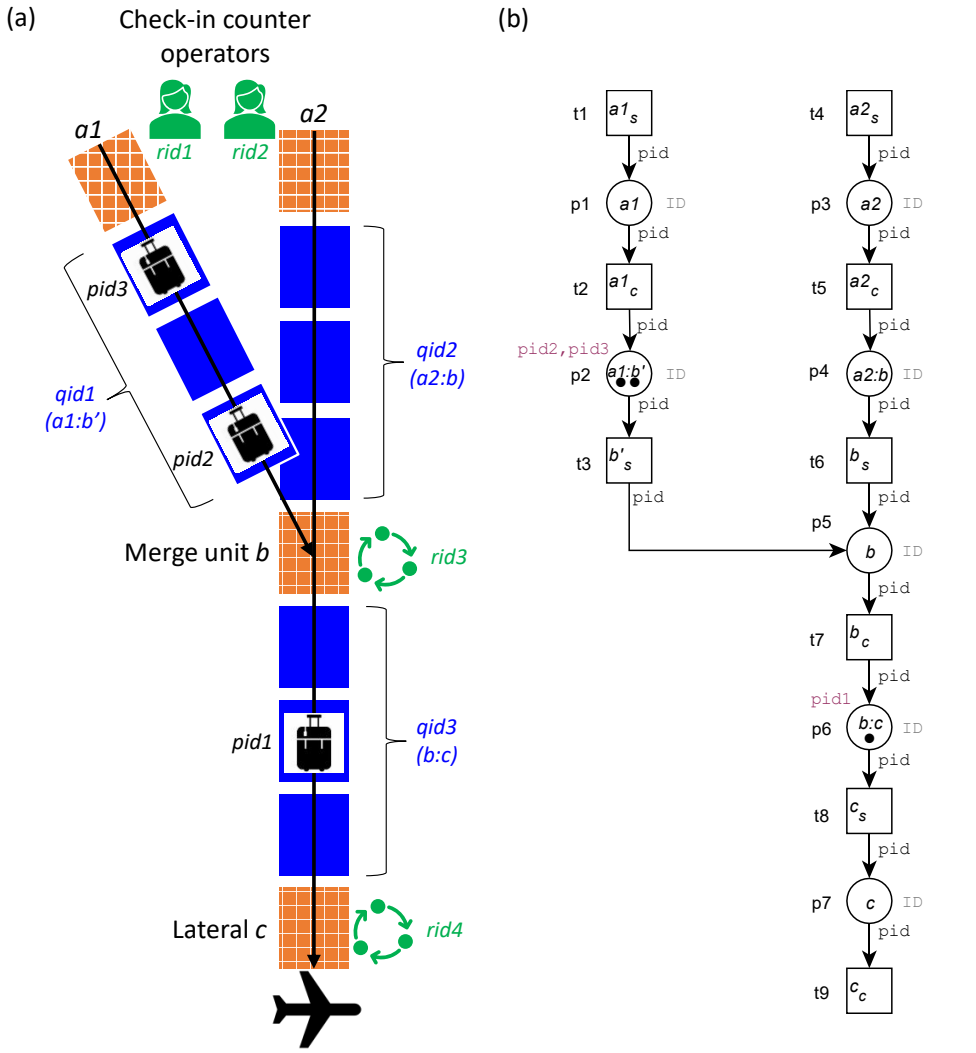


Figure 6.3: Diagram of the BHS of Figure 6.1, including resources and queues (a) and its CPN model (b).

1. When a bag enters one of the workstations, a free token (*rid*1 or *rid*2) on place $idle^{ops}$ is consumed by an occurrence of transition $start^{ops}$, and that resource identifier is produced on place $busy^{ops}$, i.e., the corresponding resource becomes busy and starts the bag handling.
2. After the bag handling is completed and the bag has *completely* left the machine location, the "busy" token on place $busy^{ops}$ (*rid*1 or *rid*2, depending on which one was previously consumed from place $idle^{ops}$) is consumed by an

occurrence of transition $complete^{ops}$, and that resource identifier is produced "back" on place $idle^{ops}$, i.e., the resource becomes free and available for serving the next bag.

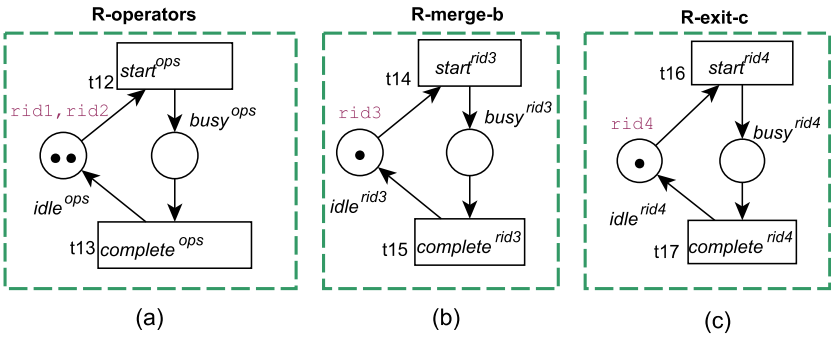


Figure 6.4: Resource proclets.

Next, we consider a different resource – merge unit b that has *two* inputs and one output. It receives bags coming either from check-in counter $a1$ or $a2$ and merges them into the single conveyor $b : c$ going to exit c. We describe it as R-proclet *R-merge-b* (Figure 6.4(b)) as follows. Because a merge unit can handle only a single bag at a time, entity *R-merge-b* has only one instance with resource identifier *rid*3. The merge unit lifecycle is as follows.

1. The resource of the unit is free initially (the only token is on place *idle*).
2. On receiving a bag, resource *rid*3 switches into state *busy* (by an occurrence of transition $start^{rid3}$).
3. After merging is finished and the bag has left the unit, the resource switches back to state *idle* (by an occurrence of transition $complete^{rid3}$ token *rid*3 is consumed from place $busy^{rid3}$ and the same resource identifier is produced on place $idle^{rid3}$).
4. Finally, lateral (exit) c receives bags through its only input and off-loads them out of the system (for example, for further loading onto a dolly-tug and transportation to aircraft). It also has only one resource *rid*4 as it can process one bag at a time.

This resource is modeled as R-proclet *R-exit-c* (Figure 6.4(c)), similarly to the merge unit resource *rid*3. As we can see, the life cycles of all those resources are identical. However, their parameters, such as the service and waiting time, can be different. Later in this chapter, we introduce them in resource models, and we use them in Chapter 7 for our log repair approach.

Conveyors. The third entity type is *conveyor*. A conveyor takes a bag in, moves it forward, and hands it over to the following machine. Depending on the conveyor capacity (proportional to its length), it can move one or multiple bags at the same time, while the bags leave conveyors in the same order they entered them. As we discussed in Chapter 4, we consider conveyors as FIFO queues, where incoming bags are enqueued to the queue, and outgoing bags are dequeued from it. In Section 2.3, we used a CPN model of a FIFO queue as a running example (see Figure 2.7), so we refer to it for the detailed description. In this chapter, we re-use this model for the implementation of conveyor entities as Q-proclets (*Q* stands for *Queue*). Because one conveyor is a single queue, each conveyor entity has only one instance. In our example, conveyor $a1 : b'$ between check-in counter $a1$ and merge unit b (Figure 6.3(a)) is modeled as Q-proclet *Q-a1:b'* (Figure 6.5), where place q^{qid1} contains the queue instance identifier $qid1$, and place $capacity^{qid1}$ has 3 tokens for capacity 3. The other queues *Q-a2:b* and *Q-b:c* are modeled identically, while the values of the queue instance identifiers ($qid2$ and $qid3$ respectively), and queue capacity (3 bags per each queue) are configured per entity. In Figure 6.6, the system with all the P-, Q- and R-proclets is shown.

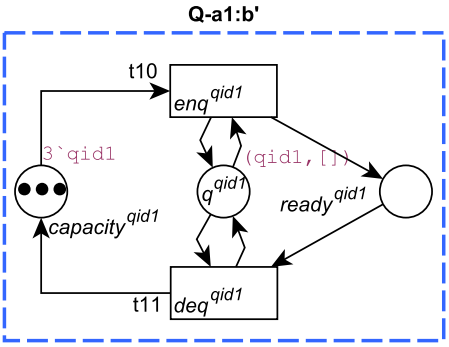


Figure 6.5: Queue proclet.

To summarize, we model a BHS as a single P-proclet for describing different instances of the entity bag, and multiple Q- and R-Proclets for describing different entities of the types of conveyor and resource. Each R-proclet and Q-proclet is the same but their parameters, such as instance identifiers, capacities, etc. can be different. Note, that the models shown in Figure 6.1(b), Figure 6.4, Figure 6.5, and Figure 6.6 still miss some more technical details for a complete formalization, but they illustrate the ideas behind the final formal model what we define in this chapter. In Section 6.3, we introduce the *templates* of Q- and R-Proclets that can be configured by such parameters and used for modeling all the system resources and conveyors uniformly. Next, we show how proclets representing separate entities synchronize in the proclet system.

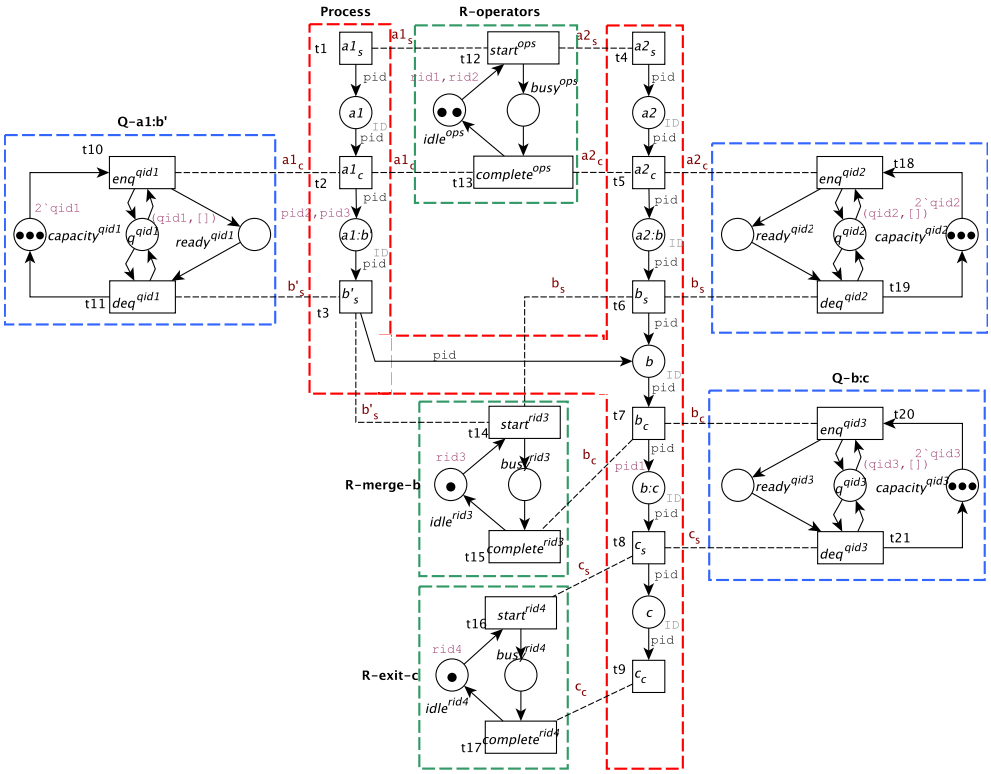


Figure 6.6: Synchronous proclet model of the BHS shown in Figure 6.1(a).

6.2.3 Behavior of BHSs through Synchronous Proclets

So far we described bags, conveyors, and resources as separate P, Q, and R-proclets, and made each entity instance distinguishable through unique token identifiers. To describe how bags interact with resources and queues, a proclet system has *channels* that describe how transitions of different entities can *synchronize*.

Synchronous Channels. In the proclet system, a synchronous channel is defined by the pair of transitions (t_i, t_j), thereby connecting two transitions. Usually, these transitions belong to different proclets. A local transition that is not connected via any channel always occurs on its own. However, two transitions, connected by a synchronous channel, occur *simultaneously*. For example, transitions $t1$ of proclet *Process* and transition $t12$ of proclet *R-operators* in Figure 6.6 are connected by channel $(t1, t12)$, which is shown as a dotted line, and therefore occur simultaneously.

In general, a transition can be connected by channels with multiple transitions, and synchronize with one or multiple different transitions at a time. For example, transition $t12$ synchronizes *either* with transition $t1$ or $t4$, while transition $t2$ synchro-

nizes with $t10$ and $t13$ simultaneously. To distinguish these situations, the channels are *labeled*, and only transitions connected by channels with the same label occur simultaneously.

For example, in Figure 6.6, channels $(t1, t12)$ and $(t4, t12)$ have different labels $a1_s$ and $a2_s$ respectively, so either transitions $t1, t12$ or $t4, t12$ occur simultaneously. In contrast, channels $(t2, t10)$ and $(t2, t13)$ have the same label $a1_c$, so transitions $t2$, $t10$, and $t13$ occur simultaneously.

In the following, we first explain how channels synchronize transition occurrences, then describe how to interpret the run in the terms of baggage handling, and finally discuss the partial-order semantics of synchronous proclet systems.

Semantics. We explain the intuitive semantics of synchronous proclet systems using the run of the system of Figure 6.6, shown in Figure 6.7(a) as follows.

1. In the run, the bordered transition $t1$ ($a1_s$ of the bag process) occurs (event $e1$ in the run) and creates token $pid2 \in \mathscr{I}$ on place $a1$, thereby creating an instance of the entity bag.
   Because transition $t1$ is connected with transition $t12$ (i.e., the start transition of proclet *R-operators*) through channel $(t1, t12)$, $t12$ occurs (event $e1'$ in the run) together at once with $t1$ in a single *synchronized event* $e1*$. We refer to such synchronized events by appending the asterisk symbol '*' to event identifiers.
   Note, transition $t12$ was enabled initially because the initial marking provided resource identifier tokens $rid1$ and $rid2$ on place $idle^{ops}$.
   The occurrence of $t12$ (event $e1'$) consumed (i.e., "engaged") resource $rid1$ for handling bag $pid2$, and produced the same resource identifier on place $busy^{ops}$, making $rid1$ busy, i.e., unavailable for handling other bags.
2. Similarly, the other bordered transition $t4$ ($a2_s$ of the bag process) occurs (event $e7$) and creates token $pid1 \in \mathscr{I}$ on place $a2$, creating another instance of the entity bag.
   Again, together with $t4$, the start transition $t12$ of proclet *R-operators* occurs (event $e7'$) in the synchronous event $e7*$, because $t4$ and $t12$ are connected through channel $(t4, t12)$.
   This time, $t12$ consumed the other resource identifier token $rid2$ from place $idle^{ops}$ and made it busy by producing the same resource identifier on place $busy^{ops}$.
3. Then, $t2$ (transition $a1_c$ for the bag process) synchronizes with $t13$ (the complete transition for *R-operators*) and $t10$ (the entire transition for queue *Q-a1:b'*) in event $e2*$ (synchronization of events $e2$, $e2'$, $e2''$) for identifier tokens $pid2$, $rid1$ and $qid1$ respectively. It means the bag $pid2$ moves from location $a1$ to the queue (conveyor) $a1 : b'$ and resource $rid1$ becomes free (idle) again.

In the run, events of the same instances directly cause each other, e.g., $e2$ is directly caused by $e1$, and $e2'$ is directly caused by $e1'$. As those events occurred as

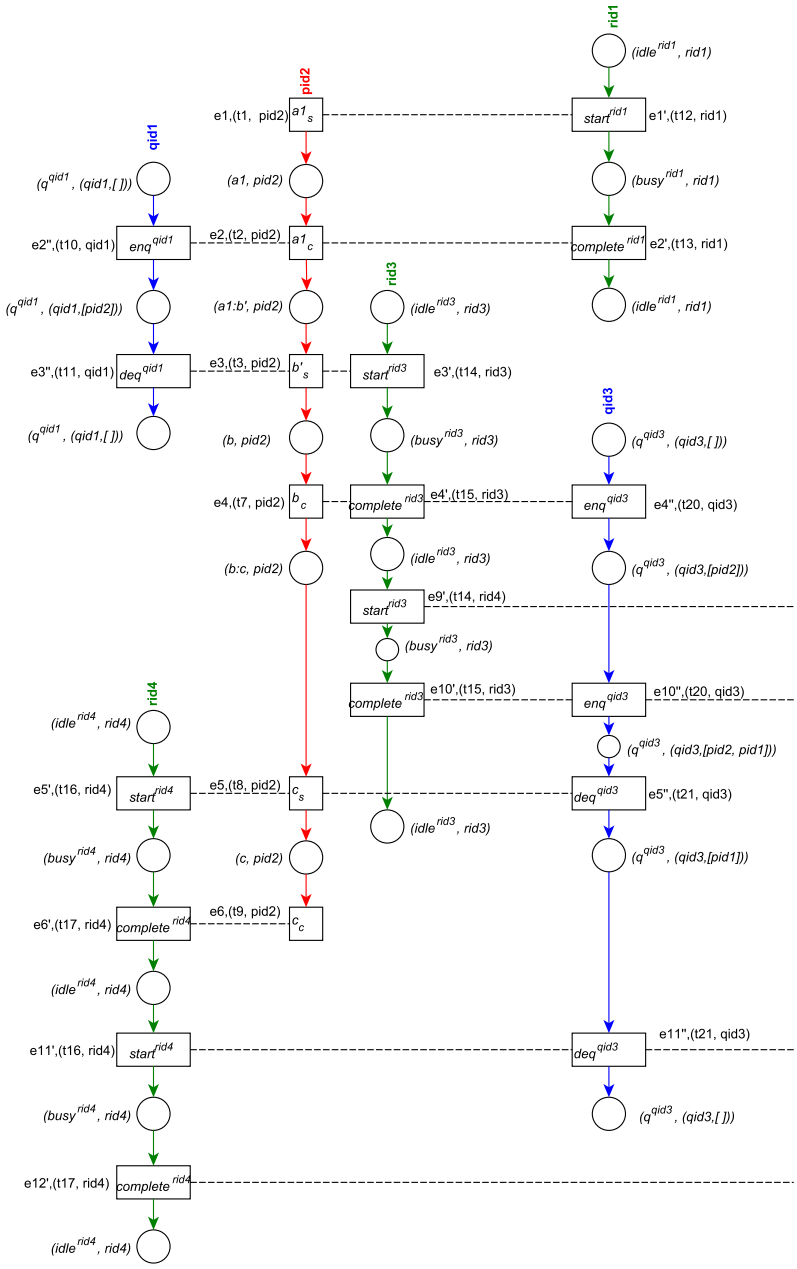


Figure 6.7: The run of the synchronous proclet system of Figure 6.6, part 1. The second part is shown in Figure 6.8.

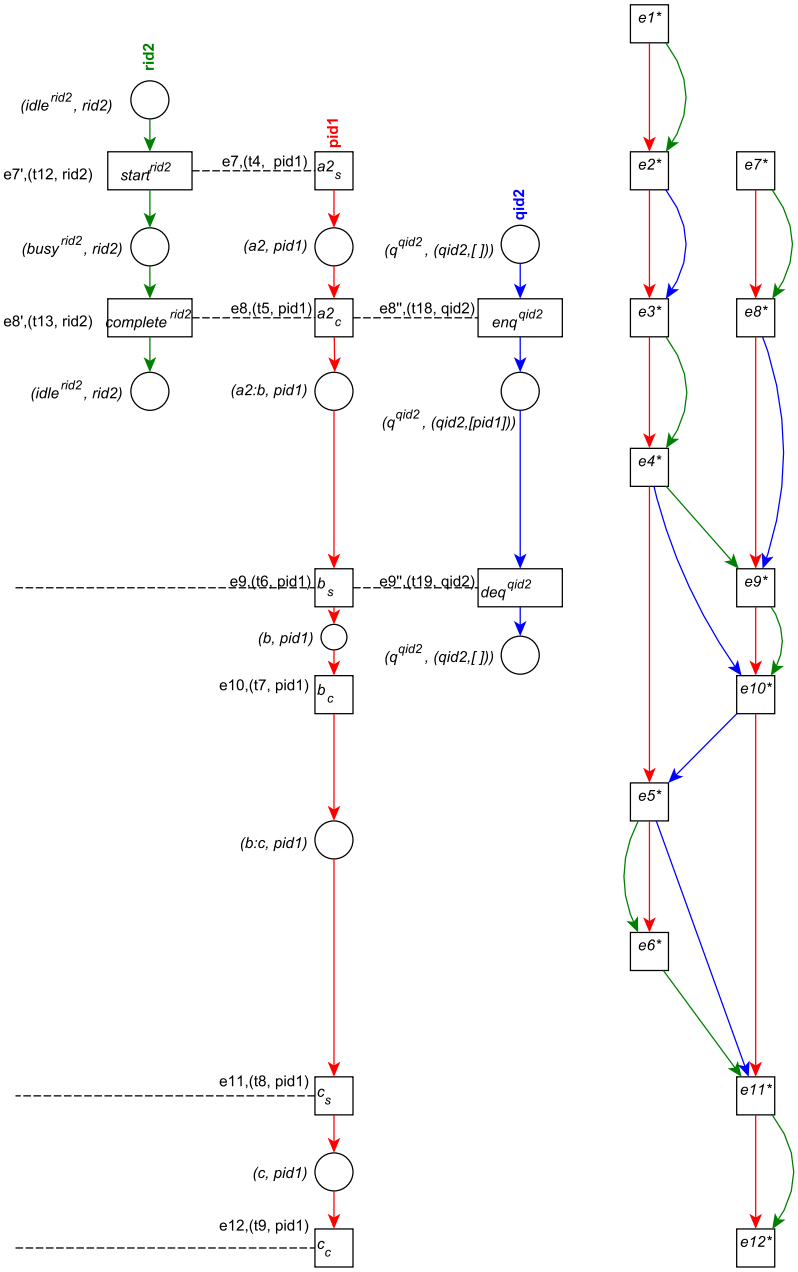


Figure 6.8: The run of the synchronous proclet system of Figure 6.6, part 2. The first part is shown in Figure 6.7.

synchronized events $e1*$ and $e2*$, $e2*$ is directly caused by $e1*$. The labeled partial order in Figure 6.8(b) shows causalities between synchronized events. While $e1*$ and $e2*$ are related to each other, some other events are not. For example, $e1*$ and $e7*$ do not causally depend on each other, so they are unordered in the partial order.

Let us follow the rest of the path of bag *pid*2 through the system.

1. In event $e3*$ three transitions synchronize. An item on queue *qid*1 reaches the end of the queue and is dequeued, i.e., $t11$ fires. Bag *pid*2 starts process step *b*, i.e., $t3$ fires. Free resource *rid*3 of *R-merge-b* starts handling *pid*2 and switches from state *idle* to *busy* ($t14$). As a result, bag *pid*2 starts merging onto the main linear conveyor $b:c$.
2. Afterward, $t15$ (resource *rid*3 finishes handling) synchronizes with $t7$ (the completion of process step *b* for *pid*2) and $t20$ (the enqueue transition for queue *Q-b:c*) in event $e4*$, bag *pid*2 is merged onto conveyor $b:c$.
3. Subsequently, bag *pid*2 leaves queue *Q-b:c* ($e5*$) and gets out of the system by exit *c* ($e6*$). If we continue applying these concepts, we obtain the full run of synchronized events in Figure 6.7. In contrast with the run in Figure 6.2, the sub-runs of instances *pid*1, *pid*2 are not independent but ordered in queue *qid*3 (conveyor $b:c$) at events $e4*, e10*, e5*$ and $e11*$. This can be clearly seen in the labeled partial order of the event in Figure 6.7(b).

Because we model each system entity (bags, conveyors, resources) separately, and distinguish entity instances through unique identifier tokens, the run for the entire system is an interaction of runs of each of the involved instances of the entities, i.e., we can find the run of each entity back in the run of the full system. This property is crucial for defining the semantics of such models for conformance checking and log repair.

Nevertheless, the existing definitions of synchronous proclet systems [48] cannot precisely capture the behavior of systems with shared resources and queues because they have no support for data types with element ordering, such as queues, and no support for modeling time characteristics, such as the minimum waiting time in queues. So in Section 6.4 we provide the exact definition capturing this behavior.

6.3 Approaching Modeling: PQR-Systems

In this section, we present the example of Section 6.2 now in the full formal syntax of PQR-systems and describe the system run example of Section 6.2 using these concepts. We focus on how the identifiers of the process, queue, and resource proclets interact, especially through the queue proclets. For that, we introduce the syntax of the PQR-system and complete CPN models of its proclets, and semantics of the PQR-

system. The section starts by introducing a way in which we label proclet transitions, to make the following discussion clearer.

6.3.1 Labels

Problem and Idea. A BHS comprises many interconnected resources and conveyors, whose entity models have a strict internal "structure". For example, each resource has states *idle* and *complete*, and each queue has transitions for enqueuing/dequeuing, etc. Expressing this structure through unique transition labels serves two goals:

- make the discussion clearer by referring to transitions through unique structured labels, and
- allow the more compact and simple formal definition of the PQR-system.

We use the regular structure of PQR-systems to provide a robust way of transition labeling. As we can see in the example of the PQR-system in Figure 6.11, it consists of the only P-proclet, which forms a "backbone" for connecting the other types of proclets to its transitions. Note the following:

- each process step of the P-proclet has *start* and *complete* life-cycle transitions from the set $LT_{\text{PQR}} = \{start, complete\}$,
- a process step can have multiple *start* and/or *complete* transitions, e.g., merge step b has two start transitions $t3$ and $t6$ because it can start merging a bag coming either from $a1$ or $a2$, so we have additional superscript (tag) $'$ in the label of $t3$, so the label of $t3$ is b'_s, while the label of $t6$ is just b_s, i.e., it does not have any subscript,
- the way how transitions are connected by channels is strict: the *start* transitions of the P-proclet can be connected only with *start* transitions of the R-proclets and *enqueue* transitions of the Q-proclets, while the *complete* transitions of the P-proclet can be connected only with *complete* transitions of the R-proclets and *dequeue* transitions of the Q-proclets.

Using these observations, we introduce an internal structure for transition labels of P-, Q- and R-proclets.

P-Proclet Labels. For P-proclet transition labels, we introduce transition labels as *triplets* containing an activity name, a life cycle transition name, and a tag.

Definition 6.1 (Transition labels of P-proclets)**.** *Let $LT_{\text{PQR}} = \{start, complete\}$ be the set of life-cycle transitions, and let Tags be the set of tags, where $\epsilon \in Tags$ is an* empty tag*. A set $\Sigma_{\text{tP}} = Act \times LT_{\text{PQR}} \times Tags \subset \Sigma$ is the set of P-proclet transition labels.*

While $Act \times LT_{\text{PQR}}$ describes the activity and the life-cycle transition such as $(b, start)$, the tag allows us, for example, to distinguish two different ways to *start* activity b, see $t3$ and $t6$ in Figure 6.6. In this example, $t3$ and $t6$ has label $(b, start, ')$ and $(b, start, \epsilon)$

respectively. We write a label by placing the life-cycle transition name and tag below and above a process step name respectively, e.g., the label of $t3$ (label $(b, start, ')$) is written succinctly as b'_{start} or even shorter as b'_s in Figure 6.6, while for the label of $t6$ we write nothing for tag ϵ (label b'_s in Figure 6.6). In the remainder of this thesis, for P-proclet labels, we write start and *s*, complete and *c* interchangeably for the sake of space and clarity.

Labels of Q- and R-Proclets. The labels of the Q- and R-Proclets are defined in their templates as

- $\Sigma_{tQ} = \{enq, deq, queue, capacity, ready\} \times \mathscr{I} \subset \Sigma$, and
- $\Sigma_{tR} = \{start, complete, idle, busy\} \times 2^{\mathscr{I}} \subset \Sigma$

respectively, where $\mathscr{I}$ is the set of identifiers (Definition 2.7). The use of the unique identifiers $\mathscr{I}$ as the second element makes the labels unique per entity and allows for referring to the corresponding transitions and places via their labels. We write a label of a Q- or R-proclet using its identifier as a superscript for the step in the queue or resource. For example, we write enq^{qid1} for label $(enq, qid1)$ of transition $t10$ (Figure 6.6).

6.3.2 Syntax

Process. We model the baggage handling process as a single P-proclet of the PQR-system. It represents process steps connected according to the logical layout of the system conveyors, which can be typically derived from the system MFD, as discussed in Section 4.4. As each resource proclet life-cycle has *start* and *complete* transitions, we also define these life-cycle transitions in the P-proclet for the process steps. The R-proclets synchronizes their start and completion of serving with the corresponding life-cycle transitions of the process steps. For example, for process step $a1$ of the system in Figure 6.6, transition $a1_s$ models the beginning of the checking-in step, and transition $a1_c$ models its completion. For the P-proclet, the following properties hold.

- No "dangling" *start* or *complete* transitions are possible in the P-proclet, i.e., these transitions always come together in a pair with a place in between. Multiple *start* and/or *complete* transitions of the same process step are possible as well if for each pair of those *start* and *complete* transitions there is always the same place in between.
- The P-proclet net is a *state machine* because in systems with shared resources and queues a bag cannot be divided into pieces, i.e., no parallel process steps are possible for a bag.
- The P-proclet is transition-bordered and new instances are generated by transitions without preset.

- The P-proclet typically has multiple input and output transitions (e.g., $t1$ and $t4$ in Figure 6.6) because a typical BHS has multiple entries and exits.
- The P-proclet can be transformed into a WF-net according to Definition 2.6.

The last property makes existing conformance checking techniques [24, 43] applicable to P-proclets, this is utilized in Chapter 7.

Resources. In Section 6.2.2 we showed how a BHS resource life-cycle can be modeled as a CPN (see Section 2.3), but we intentionally omitted some technical details, such as arc annotations and time characteristics, to make the main ideas clearer. In this section, we transform the model shown in Figure 6.4 into a template for instantiating R-proclets and add three parameters for specifying the resource identifiers, minimum service, and waiting time. Later, we do the same for Q-proclets.

First, we annotate the places and arcs as shown in Figure 6.9. As the R-proclet has tokens carrying resource identifiers, each place has color set `ID`, which we earlier introduced for modeling bag identifiers, and is timed. Correspondingly, we annotate the arcs with variable `rid` of color set `ID` for consuming and producing tokens with resource identifiers. Initial marking $\{RID_1, \ldots, RID_n\}$ provides a multiset of resource identifiers on place $idle^{\{RID_1,\ldots,RID_n\}}$, thereby defining what resources this proclet models. The initial marking on place $busy^{\{RID_1,\ldots,RID_n\}}$ is empty since no case (bag) handling started yet.

Next, we implement the time characteristics. During bag handling, each resource is busy from the moment the bag's front side entered its location (transition *start*) until the moment its back side left the location (transition *complete*). So we assume

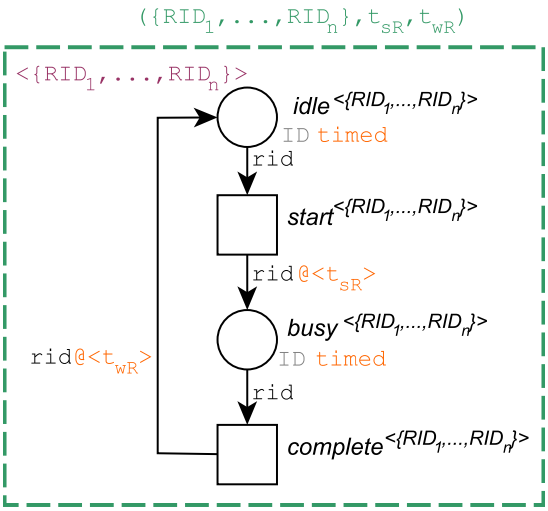


Figure 6.9: R-proclet template.

that the *minimum service time* required for a bag to pass a resource location through completely is t_{sR}. The actual service time can be $\geq t_{\mathrm{sR}}$ but cannot be less. To maintain this characteristic, we append the expression of arc $(start, busy)$ (Figure 6.9) with a delay $\ldots@t_{\mathrm{sR}}$ such that a token producing by transition *start* is ready in time t_{sR} after its occurrence.

Additionally, in BHSs some protective space is typically kept between bags. To take it into account, we assume that the *minimum waiting time* t_{wR} has to pass after a bag has left the resource location to let it be transported further, thereby making this protective space. To maintain this characteristic, we append the expression of arc $(complete, idle)$ with a delay $\ldots@t_{\mathrm{wR}}$ so a token producing by transition *complete* is ready in time t_{wR} after its occurrence.

Finally, we want to use the template to instantiate and model multiple resource entities separately. For that, the template, shown in Figure 6.9, has parameters $(\mathscr{I}_R, t_{\mathrm{sR}}, t_{\mathrm{wR}})$, where

1. parameter $\mathscr{I}_R$ is a set of resource identifiers,
2. parameter t_{sR} is the minimum service time,
3. parameter t_{wR} is the minimum waiting time.

Note, the parameter $\mathscr{I}_R$ is used also to generate the superscript of the proclet transition and place labels to make them *unique* within the proclet system. For that, the set of identifiers is used as the label superscript, as we explained in Section 6.3.1.

To instantiate this template, each parameter should be replaced with a value directly in the CPN definition, for example, to instantiate R-proclet *R-merge-b*:

1. the initial marking $\mathscr{I}_R = \{RID_1, \ldots, RID_n\}$ on place *idle* is replaced with the set $\{rid3\}$,
2. the minimum service and waiting time t_{sR} and t_{wR} on arcs $(start, busy)$ and $(complete, idle)$ are replaced with values 1 and 2 respectively,
3. and the set $\mathscr{I}_R = \{RID_1, \ldots, RID_n\}$ of resource identifiers for all the label superscripts is replaced with $\{rid3\}$.

We write the tuple $(\mathscr{I}_R, t_{\mathrm{sR}}, t_{\mathrm{wR}})$ to refer to a copy of the R-proclet template with concrete values for $\mathscr{I}_R, t_{\mathrm{sR}}$, and t_{wR}, e.g., $(\{rid3\}, 1, 2)$ for proclet *R-merge-b*.

Queues. Similarly to the R-proclet, we add missing implementation details to the queue model in Figure 6.5. In Section 2.3 (Figure 2.7) we modeled a FIFO queue with the time characteristics. In this section, we adopt this model for modeling BHS conveyors and convert it into a template. The resulting template is shown in Figure 6.10. The template has parameters $(QID, size, t_{\mathrm{wQ}})$, where

1. parameter $QID \in \mathscr{I}$ is the unique queue identifier,
2. parameter *size* is the queue (conveyor) capacity, i.e., the maximal number of elements the queue can contain,

3. parameter t_{wQ} is the minimum waiting time, i.e., the minimum time required to pass since enqueuing of an element before its dequeuing.

Note, parameter *QID* is additionally used as the superscript of the proclet transition and place labels to compose *unique* transition and place labels within the proclet system (see Section 6.3.1).

To instantiate this template, each parameter should be replaced with a value directly in the CPN definition, for example, to instantiate Q-proclet *Q-a1:b'*:

1. parameter *QID* in the markings on places *capacity* and *queue* is replaced with identifier *qid*1,
2. parameter *size* is replaced with capacity 2,
3. parameter t_{wQ} is replaced with duration 10,
4. and the superscripts of each label is replaced with *qid*1 (i.e., the value of parameter *QID*).

Similarly to R-proclets, we write the tuple $(QID, size, t_{\text{wQ}})$ to refer to a copy of the Q-proclet template with concrete values for *QID*, *size* and t_{wQ}, e.g., $(qid1, 2, 10)$ for proclet *Q-a1:b'*.

In contrast to the implementation of the queue in Section 2.3, this template does not model an arrival process, so variable `pid` on the arcs $(t1, p2)$ and $(t1, p3)$ is not bound by any input arc, i.e., it is unbound. Instead, the arrival process to the queue emerges when we synchronize the enqueue transitions of the Q-proclet with a process step of the P-proclet. We explain that in more detail in Section 6.3.3.

System. So far, we described the system building blocks as the P-, Q- and R-proclets. An example of the PQR-system of the BHS of Figure 6.1(a) is shown in Figure 6.11. To keep the focus on the system structure, i.e., on the proclets and channels, the

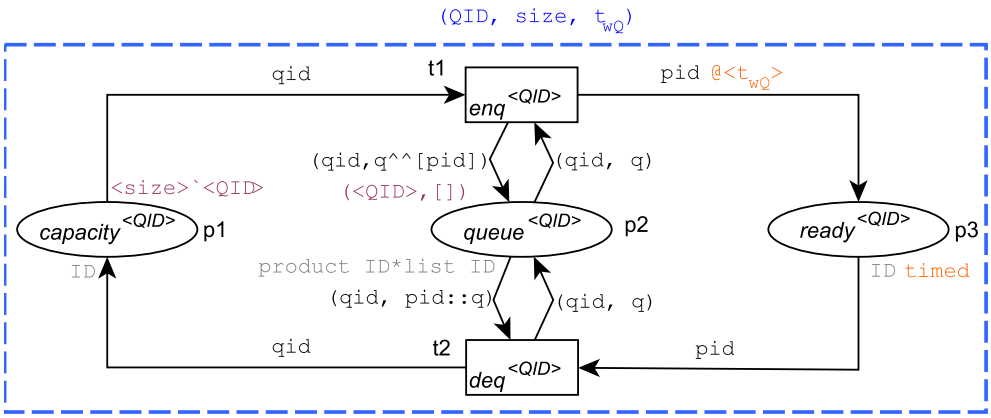


Figure 6.10: Q-proclet template.

implementation details of the Q- and R-Proclets are hidden. The model provides connections between the building blocks as follows:

1. *Start* and *complete* transitions of each process step are connected with transitions of at least one R-proclet so that all *start* transitions of each P-proclet process step are connected to the *start* transition of this R-proclet, and all *complete* transitions of that process step are connected to the *complete* transition of this R-proclet. As a result, each process step has at least one resource that can perform it (completely).
2. Each transition of the P-proclet, except bordered transitions without preset or postset, is connected with one Q-proclet such that a *complete* transition of the P-proclet is always connected to the enqueue transition of this Q-proclet, and the dequeue transition of the Q-proclet is connected to the *start* transition of one of the directly following process steps (of the P-proclet). That is, on completion of a non-terminal process step, a bag is always handed over to a directly following process step *through a queue* (conveyor).

6.3.3 Semantics

Now, we describe how the building blocks, i.e., the P-, Q- and R-proclets, synchronize through synchronous channels. For that, we explain how identifiers from different proclets interact, and how the process identifiers of the P-proclet synchronize with variable `pid` of the Q-proclet. Then, we explain the replay semantics of PQR-systems.

Binding. In Section 6.2.3 we explained the semantics of a proclet system, which is similar to Figure 6.11, in terms of a system run. However, that example does not explain how variable `pid` of the Q-proclets is bound to a value during synchronization. Let us now extend this explanation for the PQR-system of Figure 6.11, starting from event $e4*$ of the run in Figure 6.7, because this part of the run can demonstrate the idea the best.

First, we briefly recap how the situation was developing before the occurrence of $e4*$. For bag *pid*2,

1. channel $a1_s$ (transitions $t1, t12$) fires synchronously,
2. then channel $a1_c$ ($t10, t2, t13$) fires,
3. and eventually channel b'_s ($t11, t3$, and $t14$) fires synchronously as well .

As a result, *pid*2 is on place *b*, queue *qid*1 is , resource *rid*1 is in state *idle*, and resource *rid*3 is in state *busy*. The marking (see Section 2.3.2) at time $time_3$ *after* event $e3*$ occurred, is as follows.

- $m_3(b) = [pid2@time_3]$,
- $m_3(idle^{rid3}) = []$,

- $m_3(busy^{rid3}) = [rid3@time_3]$,
- $m_3(queue^{qid3}) = [(qid3, [])@time_3]$,
- $m_3(ready^{qid3}) = []$,
- $m_3(capacity^{qid3}) = [qid3^{capacity^{qid3}}@time_3]$.

In another part of the system, bag $pid1$ reached place $a2:b$ after firing channels $a2_s$ (transitions $t12, t4$) and $a2_c$ ($t13, t5, t18$). Correspondingly, queue $qid2$ is not yet, resource $rid2$ is in state *idle* since it already handed $pid1$ over (to the conveyor modeled by $qid2$). The marking at time $time_8$ is as follows.

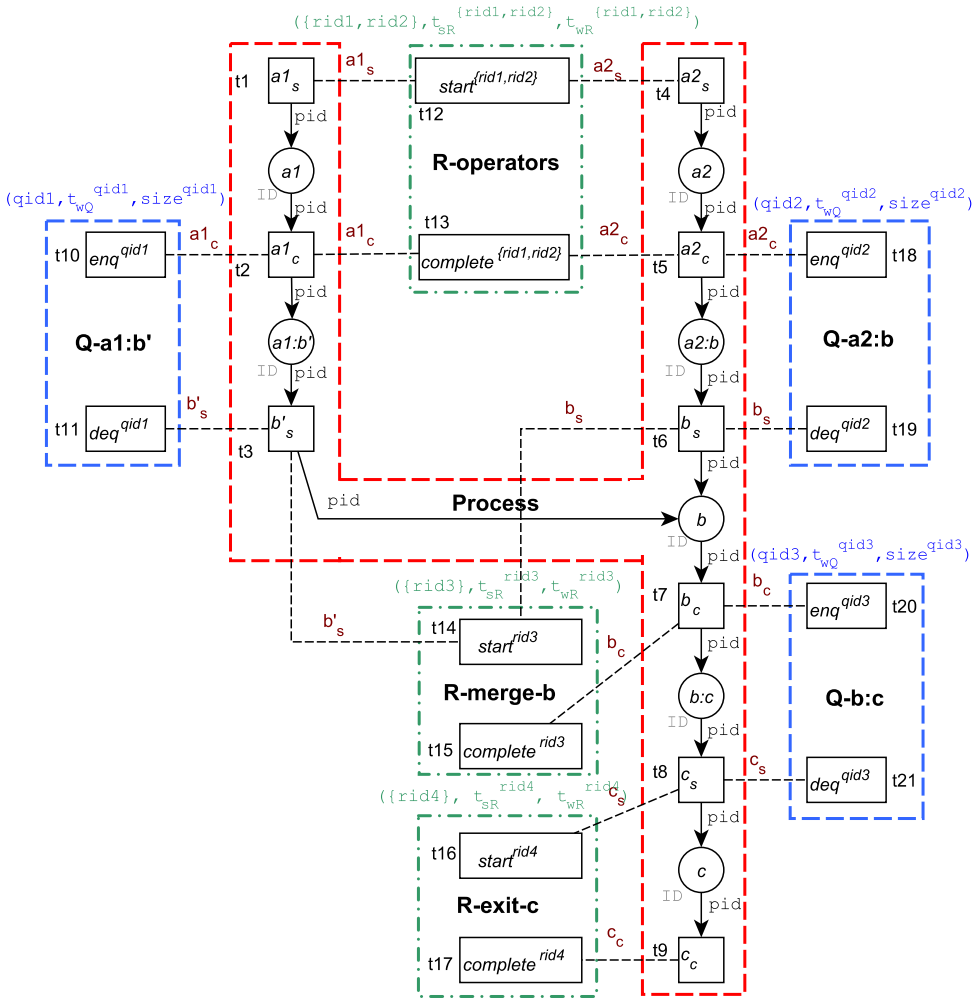


Figure 6.11: Example of a PQR-system modeling the BHS in Figure 2.1(a).

- $m_8(a2 : b) = [pid1@time_8]$,
- $m_8(idle^{rid2}) = []$,
- $m_8(busy^{rid2}) = [rid2@time_8]$,
- $m_8(queue^{qid2}) = [(qid2, [pid1])@time_8]$,
- $m_8(ready^{qid2}) = [pid1@time_8]$,
- $m_8(capacity^{qid2}) = [qid2^{capacity^{qid3}-1}@time_8]$.

Now, transitions $t7, t15$ and $t20$ are enabled. In the system, these transitions synchronize by channels $(t7, t15)$ and $(t7, t20)$ (labeled b_c) and occur simultaneously, for example, in event $e4*$ at time $time_4$ (Figure 6.7). For all three transitions to synchronize in $e4*$, they have to *agree* on a binding, i.e., the variables that occur at the arcs of $t7$, $t15$, and $t20$ have to be bound to the same value. In a PQR-system, this is only variable `pid`. Thus, to fire $t7$ (b_c) in the P-proclet, we have to bind `pid` to an identifier that is on the input place b of $t5$ in the P-proclet, i.e., for instance bag $pid2$. Transition $t20$ in the Q-proclet has `pid` only on its outgoing arcs. Transition $t15$ in the R-proclet *R-merge-b* does not have `pid` on any arc. Thus, binding `pid` to $pid2$ and synchronizing all three transitions

- consumes $pid2$ from place b (in the P-proclet),
- produces $pid2$ on place $b : c$ (in the P-proclet),
- enqueues $pid2$ into the queue (in Q-proclet $qid3$).

As a result of enqueuing, transition enq^{qid3} (1) "appends" the value of `pid` to the queue on place $queue^{qid3}$, , and (2) produces the bag identifier $pid2$ (received through `pid`) on place $ready^{qid3}$. The current marking of these places is as follows:

- $m_4(queue^{qid3}) = [(qid3, [pid2])@time4]$,
- $m_4(ready^{qid3}) = [pid2@(time4 + t_{\mathrm{wQ}}^{qid3})]$,

where t_{wQ}^{qid3} is the minimum waiting time of the queue $qid3$. As we show, distinguishable tokens on these places allow maintaining the FIFO ordering and the mandatory minimum waiting time t_{wQ} of the queue.

When later transitions $t7, t15$ and $t20$ occur as a synchronized event $e10*$ at time $time_{10}$ for the other instance $pid1$ of the P-proclet, variable `pid` on the arcs of proclets *Process* and *Q-b:c* is bound to $pid1$ correspondingly. The synchronous firing of all three transitions for this binding

- consumes $pid1$ from place b (in the P-proclet),
- produces $pid1$ on place $b : c$ (in the P-proclet),
- enqueues $pid1$ into the queue (in Q-proclet $qid3$).

Now, transition enq^{qid3} "appends" the value of `pid` ($pid1$) to the queue and produces this bag identifier on place $ready^{qid3}$. As shows the resulting marking

- $m_{e10*}(queue^{qid3}) = [(qid3, [pid2, pid1])@time_{10}]$,

- $m_{e10*}(ready^{qid3}) = [pid2@(time_4 + t_{\text{wQ}}^{qid3}), pid1@(time_{10} + t_{\text{wQ}}^{qid3})]$,

(1) the queue $qid3$ contains a list of two bags identifiers, ordered according to the FIFO discipline (i.e., in the order of enqueuing), and (2) place $ready^{qid3}$ has the corresponding bag identifier tokens, which become available for consumption at $time_4 + t_{\text{wQ}}^{qid3}$ and $time_{10} + t_{\text{wQ}}^{qid3}$ in the order of enqueuing as well.

Interestingly, transition $t20$ of queue $qid3$ essentially works as the initial transitions of the P-proclet with a ν-semantics, but the difference is that where the P-proclet (and ν Petri nets in general) generate completely new identifiers, the Q-proclet gets one from an arrival process.

In event $e5*$, transitions $t8$, $t16$, $t21$ synchronize with binding $\texttt{pid} = pid2$ for variable `pid`. Note, that here transition $t8$ in the P-proclet and $t21$ in the Q-proclet both have variable `pid` on the input arc, so for $t8$ and $t221$ to synchronize, the following must hold:

- the bag $pid2$ must be at the *head* of the queue in place *queue*,
- the timeout token $pid2$ on place $ready^{qid3}$ must be ready, i.e., at least time t_{wQ}^{qid3} (the minimum waiting time of the queue $qid3$) has to pass since the occurrence of event $e4*$, and
- the bag must be available in the input place of $t8$.

This happened for event $e5*$ for $pid2$ at time $time_4 + t_{\text{wQ}}^{qid3}$. So, bag $pid2$ is dequeued first (but not earlier than the minimum waiting time t_{wQ}^{qid3} since its enqueuing ($e4*$)). Afterward, bag $pid1$ is dequeued by $t11$ at $time_{11} \geq time_{10} + t_{\text{wQ}}^{qid3}$. So, the FIFO order and the queue minimum waiting time are preserved.

Replay Semantics. So far, we explained the ideas behind the semantics of PQR-systems. We do not define operational semantics, but define the semantics of replaying an event table (see Definition 2.8) over a PQR-system for conformance checking and log repair that states when a set of events together forms an execution that can be described by the PQR-system (i.e., can be replayed by the PQR-system). Concretely, for a given event table ET we provide conditions required for ET to be the *execution* of the PQR-system.

The example of a possible execution in the form of the event table $ET = (\{\text{pid}, \text{qid}, \text{rid}\}, \{e1*, \ldots, e12*\}, \#)$ of the system in Figure 6.11 is shown in Table 6.1. Additionally, we assume the following parameters for all the Q- and R-Proclets of the system:

- $t_{\text{sR}} = 1$,
- $t_{\text{wR}} = 2$,
- $t_{\text{wQ}} = 10$.

In this event table (Definition 2.8), case notions pid and rid are global (see Section 2.4), i.e., defined for each event, while qid is undefined for events $e1*, e6*, e7*$ and $e12*$. When this event table is considered as the execution of the PQR-system,

Event	Activity	Time	pid	qid	rid
$e1*$	$a1_s$	0	$pid2$	-	$rid1$
$e2*$	$a1_c$	1	$pid2$	$qid1$	$rid1$
$e3*$	b'_s	11	$pid2$	$qid1$	$rid3$
$e4*$	b_c	12	$pid2$	$qid3$	$rid3$
$e5*$	c_s	24	$pid2$	$qid3$	$rid4$
$e6*$	c_c	25	$pid2$	-	$rid4$
$e7*$	$a2_s$	1	$pid1$	-	$rid2$
$e8*$	$a2_c$	11	$pid1$	$qid2$	$rid2$
$e9*$	b_s	21	$pid1$	$qid2$	$rid3$
$e10*$	b_c	22	$pid1$	$qid3$	$rid3$
$e11*$	c_s	34	$pid1$	$qid3$	$rid4$
$e12*$	c_c	35	$pid1$	-	$rid4$

Table 6.1: Execution of the PQR-system of Figure 6.11 corresponding to the run of Figure 6.7.

Trace #	Event log	Proclet	Trace
σ_1	L^{ET}_{pid}	*Process*	$\langle e7*, e8*, e9*, e10*, e11*, e12* \rangle$
σ_2	L^{ET}_{pid}	*Process*	$\langle e1*, e2*, e3*, e4*, e5*, e6* \rangle$
σ_3	L^{ET}_{qid}	*Q-a1:b'*	$\langle e2*, e3* \rangle$
σ_4	L^{ET}_{qid}	*Q-a2:b*	$\langle e8*, e9* \rangle$
σ_5	L^{ET}_{qid}	*Q-b:c*	$\langle e4*, e10*, e5*, e11* \rangle$
σ_6	L^{ET}_{rid}	*R-operators*	$\langle e1*, e2* \rangle$
σ_7	L^{ET}_{rid}	*R-operators*	$\langle e7*, e8* \rangle$
σ_8	L^{ET}_{rid}	*R-merge-b*	$\langle e3*, e4*, e9*, e10* \rangle$
σ_9	L^{ET}_{rid}	*R-exit-c*	$\langle e5*, e6*, e11*, e12* \rangle$

Table 6.2: Event logs from the event table shown in Table 6.1.

for each event, the process step name refers to the transition label of the P-proclet that should occur. Case notion attributes pid, rid, and qid (if defined) refer to the corresponding variables `pid`, `rid`, and `qid` of P-, R- and Q-proclet respectively, the timestamp refers to the time when the synchronized event occurred.

Interpreting the event table in Table 6.1 in this way for each case notion pid, qid and rid (according to Definition 2.12) yields three event logs, shown in Table 6.2. Each trace of these event logs describes the *trajectory* of an instance of the corresponding process, queue, or resource entity through the system and its synchronization with other entities.

We now provide the conditions by which an event table is an execution of a PQR-system. First, we require traces $\sigma_1 - \sigma_9$ to be the traces of the corresponding proclet

instances, as defined for CPNs (see Section 2.3). For example, trace σ_2 is a trace of the P-proclet, because it can be generated by the sequence $\langle t4, t5, t6, t7, t8, t9\rangle$ of transition occurrences with corresponding bindings of the P-proclet. Likewise, trace σ_8 is a trace of instance *rid*3 of R-proclet *R-merge-b* because

1. it can be generated by the sequence $\langle t14, t15, t14, t15\rangle$ of transition occurrences with corresponding bindings of proclet *R-merge-b*,
2. the service time of each bag $\geq t_{\text{sR}}$ (e.g., $\#_{\text{time}}(e4*) - \#_{\text{time}}(e3*) = 1 \geq t_{\text{sR}} = 1$),
3. the waiting time between each consequent bags $\geq t_{\text{wR}}$ (e.g., $\#_{\text{time}}(e9*) - \#_{\text{time}}(e4*) = 9 \geq t_{\text{wR}} = 2$).

In the same way, we can confirm that also traces $\sigma_1, \sigma_3 - \sigma_7$, and σ_9 can be generated by the respective proclets and replayed on them. The details of the replay semantics are considered in Section 6.5.

Second, we require all the traces *together* not to contradict the synchronization declared in the PQR-system through channels as follows.

1. **Events with multiple case notions correspond to synchronized transitions of the respective proclets.** We check if each event in *ET* corresponds to the occurrences of transitions that indeed synchronize, i.e., for each event $e*$ the corresponding P-proclet transition t with $\ell(t) = \#_{\text{act}}(e*)$ has

   - a channel (t, t_R) to a transition t_R in an R-proclet that contains resource instance $\#_{\text{rid}}(e*)$, and
   - a channel (t, t_Q) to a transition t_Q in an Q-proclet that contains queue instance $\#_{\text{qid}}(e*)$ if $\#_{\text{qid}}(e*) \neq \perp$.

   For example, event $e2*$ in Table 6.1 has the defined values of attributes $\#_{\text{qid}}(e2*) =$ *qid*1 and $\#_{\text{rid}}(e2*) =$ *rid*1, that corresponds to the instances of proclets *Q-a1:b'* and *R-operators* respectively. Both proclets indeed synchronize with $t2$ through channels $(t2, t10)$ and $(t2, t13)$ labeled $a1_c$.
2. **The start and complete transitions of the same process step synchronize with the same R-proclet instance.** For each event e_c* of a *complete* transition in an instance *pidX* of the P-proclet, in which the instance of the synchronizing R-proclet has identifier *ridX*, the directly preceding *start* event e_s* of *pidX* must synchronize with *ridX* as well, i.e., $\#_{\text{rid}}(e_s*) = \#_{\text{rid}}(e_c*)$. For example, start event $e3*$ in Table 6.1 directly precedes complete event $e4*$ along the trace of instance *pid*2, according to the order defined by the timestamps, and these events have the same value for case notion rid, i.e., $\#_{\text{rid}}(e3*) = \#_{\text{rid}}(e4*) =$ *rid*3.
3. **For any two consequent process steps, the complete transition of the first step and the start transition of the second one synchronize with enqueue and dequeue transitions of the same Q-proclet instance respectively.** For each event e_s* of a *start* transition with preset (i.e., a non-bordered transition)

in an instance *pidX* of the P-proclet, in which the instance of the synchronizing Q-proclet has identifier $qidX \neq \perp$, the directly preceding *complete* event e_c* of *pidX* must synchronize with *qidX* as well, i.e., $\#_{\text{qid}}(e_c*) = \#_{\text{qid}}(e_s*)$. For example, event $e2*$ in Table 6.1 directly precedes event $e3*$ along the trace of instance *pid*2, according to the order defined by the timestamps, and these events have the same value for case notion qid, i.e., $\#_{\text{qid}}(e2*) = \#_{\text{qid}}(e3*) = qid1$.

If all the requirements are satisfied, we can conclude that *ET* is the execution of the PQR-system.

In the following, Section 6.4 and Section 6.5 provide the formal definitions for syntax and semantics explained in the current section.

6.4 The Formal Model of PQR-systems

In the following, we define first each of the P-, Q- and R-proclets, and then the PQR-system formally.

6.4.1 P-Proclet

The definition of a P-proclet reads as follows.

Definition 6.2 (P-proclet)**.** *A* P-proclet *is a CPN* $N = (P, T, F, \ell, Var, colSet, m_0, arcExp)$*, where* $\ell = (T \to \Sigma_{\text{tP}}) \cup (P \to \Sigma)$*, that has the following properties:*

1. *each process step* $a \in Act$ *is modeled by one or more alternative* start *transitions producing on place* p_a *with label* $\ell(p_a) = a$ *and one or more alternative* complete *transitions consuming from* p_a *as shown in Figure 6.12:*
   (a) p_a *has one or more input transitions, i.e.,* ${}^\bullet p_a \neq \emptyset$*, and* p_a *has one or more output transitions, i.e.,* $p_a{}^\bullet \neq \emptyset$*,*
   (b) *any input transition is labeled* $(a, start, tag)$ *and any output transition is labeled* $(a, complete, tag)$*,*
   (c) *all transitions bordering* p_a *are labeled differently, i.e., all input and all output transitions differ in their tags.*
2. *each* complete *transition* $t_c^a \in T$ *of process step* $a \in Act$ *is either a bordered transition, i.e.,* $t_c^{a\bullet} = \emptyset$*, or directly followed by exactly one place* p_{ab} *that is directly followed by only one* start *transition* $t_b^s \in T$ *of process step* $b \in Act$*, i.e.,* $\forall p_1 \in t_c^{a\bullet}, p_2 \in {}^\bullet t_s^b, p_1 = p_2 = p_{ab}$*,*
3. *each place* $p \in P$ *has color set* $\mathscr{I}$ *of token identifiers, i.e.,* $colSet(p) = \mathscr{I}$*,*
4. *each arc inscription of the net has variable* `pid` *as an expression. This variable carries a token identifier* $pid \in \mathscr{I}$*, i.e., for each* $(x, y) \in F, arcExp(x, y) = \texttt{pid}, colSet(\texttt{pid}) = \mathscr{I}$*,*

5. *the net is transition-bounded , i.e.,* $\forall p \in P \ {}^\bullet p \neq \emptyset \wedge p^\bullet \neq \emptyset$,
6. *the WF-net* N_{WF}*, derived from* N *according to Definition 2.6, is sound.*

In the following, we write T_N for the transitions of proclet N. The net N is transition-bound. For creating new identifiers, as the semantics in Section 6.5 defines, transitions without preset, which are always enabled, generate new fresh identifiers (i.e., create instances) as described in Section 6.2.1 and shown in Section 6.2.3 for the PQR-system run. As a consequence of points 1, 2, and 5 in Definition 6.2, net N can produce at most one token to its post-place, i.e., the net is a state machine and can be in exactly one of a finite number of states at any given time with respect to each unique identifier $pid \in \mathscr{I}$.

An example of a P-proclet is shown in Figure 6.13, it is identical to the P-proclet of the system in Figure 6.1(b) except the marking: in Figure 6.13 the model is in its initial state (i.e., without tokens on its places).

As a consequence of point 1 of Definition 6.2, a *start* transition can be only followed by a *complete* transition of the same process step, and a *complete* transition can be only followed by a *start* transition of a different process step. We define these pairs as follows.

Definition 6.3 (Start-complete pair)**.** *Let* N *be a P-proclet, according to Definition 6.2. A pair* $(t_1, t_2), t_1, t_2 \in T_N$ *is a* start-complete pair *in* N *iff:*

1. $t_1{}^\bullet = {}^\bullet t_2$,
2. $\ell(t_1) = (a, start, tag_1), a \in AN, tag_1 \in Tags$,
3. $\ell(t_2) = (a, complete, tag_2), tag_2 \in Tags$.

For example, in Figure 6.13, (t_3, t_7) is a start-complete pair because:

- $t_3{}^\bullet = {}^\bullet t_7 = \{p5\}$, i.e., transitions t_3 and t_7 are connected place $p5$,
- $\ell(t_3) = (b, start, ')$ and $\ell(t_7) = (b, complete, \epsilon)$, i.e., transitions t_3 and t_7 are *start* and *complete* transitions respectively of the same process step b.

Definition 6.4 (Complete-start pair)**.** *Let* N *be a P-proclet, according to Definition 6.2. A pair* $(t_1, t_2), t_1, t_2 \in T_N$ *is a* complete-start pair *in* N *iff:*

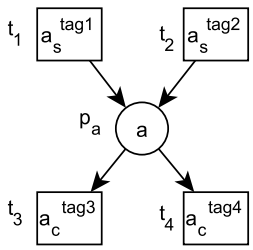


Figure 6.12: Labeling of the input and output transitions of process step a.

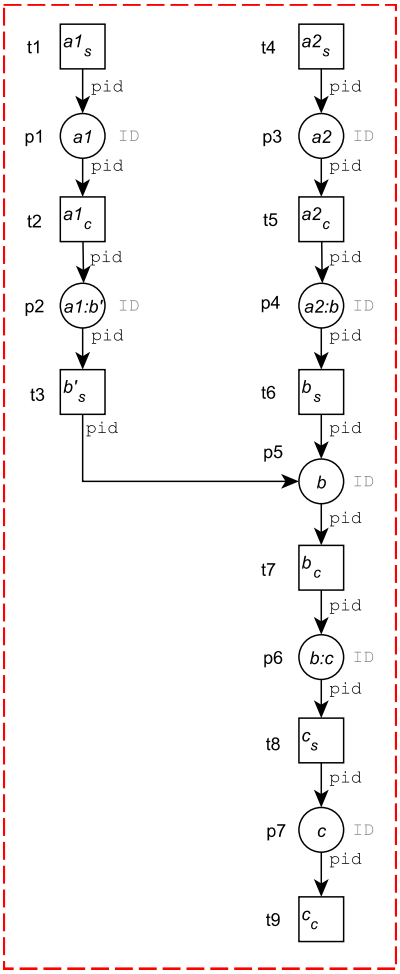


Figure 6.13: P-proclet in the initial state.

1. $t_1{}^\bullet = {}^\bullet t_2$,
2. $\ell(t_1) = (a_1, complete, tag_1), a_1 \in AN, tag_1 \in Tags$,
3. $\ell(t_2) = (a_2, start, tag_2), a_2 \in AN, a_1 \neq a_2, tag_2 \in Tags$.

For example, in Figure 6.13, (t_7, t_8) is a complete-start pair because:

- $t_7{}^\bullet = {}^\bullet t_8 = \{p6\}$, i.e., transitions t_7 and t_8 are connected via place $p6$,
- $\ell(t_7) = (b, complete, \epsilon)$ and $\ell(t_8) = (c, start, \epsilon)$, i.e., transitions t_7 and t_8 are *complete* and *start* transitions respectively of two different process steps b and c.

6.4.2 Q-Proclet

As the name implies, Q-proclets model FIFO queues with a known finite non-zero capacity and minimum waiting time $t_{\mathrm{wQ}} \in \mathbb{T}$, e.g., a conveyor of a BHS. We define the Q-proclet by referring to its template presented in Section 6.3 as follows.

Definition 6.5 (Q-proclet)**.** *A* Q-proclet *is a CPN* $N = (P, T, F, \ell, Var, colSet, m_0, arcExp)$*, where* $\ell = (T \to \Sigma_{\mathrm{tQ}}) \cup (P \to \Sigma)$*, instantiated from the* template *shown in Figure 6.10 with the following parameters:*

1. *the* queue instance identifier $QID \in \mathscr{I}$*,*
2. *the* maximal queue size $size \in \mathbb{N}$*,*
3. *the* minimum waiting time $t_{\mathrm{wQ}} \in \mathbb{T}$*.*

We call the transition pair (enq^{QID}, deq^{QID}) a *complete-start pair* of the Q-proclet because we later synchronize enq^{QID} with a *complete* transition of the P-proclet and deq^{QID} with a *start* transition of the P-proclet. Each copy of the Q-proclet template has such a complete-start pair.

6.4.3 R-proclet

An R-proclet models one or multiple resources of the same entity with the minimum waiting time t_{wR} and the minimum service time t_{sR}, e.g., a merge unit b of the BHS of Figure 6.1(a). We define the R-proclet by referring to its template presented in Section 6.3 as follows.

Definition 6.6 (R-proclet)**.** *A* R-proclet *is a CPN* $N = (P, T, F, \ell, Var, colSet, m_0, arcExp)$*, where* $\ell = (T \to \Sigma_{\mathrm{tR}}) \cup (P \to \Sigma)$*, instantiated from the* template *shown in Figure 6.9 with the following parameters:*

1. *the non-empty* set of instance identifiers $\mathscr{I}_R \subseteq \mathscr{I}$*,*
2. *the* minimum service time $t_{\mathrm{sR}} \in \mathbb{T}$*,*
3. *the* minimum waiting time $t_{\mathrm{wR}} \in \mathbb{T}$*.*

We call the transition pair $(start^{\mathscr{I}_R}, complete^{\mathscr{I}_R})$ a *start-complete pair* of the R-proclet. Each copy of the R-proclet template has such a start-complete pair.

6.4.4 PQR-system

In the previous sections, we defined the P-, Q- and R-proclets. Now, we define a PQR-system consisting of one P-proclet and multiple Q- and R-Proclets. Note, the initial marking of a P-proclet is the empty multiset $[]$ but the initial markings of Q- and R-Proclets are non-empty, so the initial marking of the PQR-system is the union of the initial markings of its Q- and R-Proclets.

Definition 6.7 (PQR-system)**.** *Let N^P be a P-proclet, $\mathcal{N}^Q$ and $\mathcal{N}^R$ be sets of Q- and R-Proclets respectively, which we enumerate as $N^P = N_0$, $\mathcal{N}^Q = \{N_1, \ldots, N_m\}$ and $\mathcal{N}^R = \{N_{m+1}, \ldots, N_n\}$. A* PQR-system *$S = (\{N_0, \ldots, N_n\}, m_v, C, \ell^C)$ has:*

1. *all transitions in $\{N_0, \ldots, N_n\}$ have unique transition labels, i.e., $\forall 0 \leq i \neq j \leq n$, $\forall t \in T_i, \forall t' \in T_j, t \neq t'$, and $\ell(t) \neq \ell(t')$,*
2. *an* initial marking *$m_v : \bigcup_{i=1}^{n} P_i \to \mathbb{N}^{\mathcal{G}}$ assigns on each place $p_i \in P_i, i > 0$ of the Q- and R-Proclets a multiset $m_v(p_i)$ of token identifiers that do not occur in a place $p_j \in P_j$ of any other proclet $N_i \neq N_j$, i.e., $\forall 1 \leq i < j \leq n, \forall p_i \in P_i, p_j \in P_j$: $m_v(p_i) \cap m_v(p_j) = \emptyset$,*
3. *a* set *C* of channels *where each channel in C is defined between the P-proclet and one of the Q- or R-Proclets, i.e., $\forall (t_0, t_j) \in C, t_0 \in T_0, t_j \in T_j$, $1 \leq j \leq n$, such that:*
   (a) *for each start-complete pair (t_1, t_2) of N_0, there exists channels $(t1, t1')$ and $(t2, t2')$ so that $(t1', t2')$ is the start-complete pair of an R-proclet $N_j \in \mathcal{N}^R$,*
   (b) *for each complete-start pair (t_1, t_2) of N_0, there exists exactly one pair of channels $((t1, t1'), (t2, t2'))$ so that $(t1', t2')$ is the complete-start pair of a Q-proclet $N_i \in \mathcal{N}^Q$,*
   (c) *for any two complete-start pairs $(t_1, t_2) \neq (t_3, t_4)$ of N_0 with channels $(t1, t1')$, $(t2, t2')$, $(t3, t3')$, $(t4, t4')$, the complete-start pair $(t1', t2')$ of Q-proclet N_k and the complete-start pair $(t3', t4')$ of Q-proclet N_l are from different proclets, i.e., $N_k \neq N_l$.*
4. *a* channel labeling function *$\ell^C : C \to \Sigma$ that provides a label for each channel in C.*

An example of a PQR-system is shown in Figure 6.11. It has:

- the only P-proclet *Process*,
- three Q-proclets *Q-a1:b'*, *Q-a2:b* and *Q-b:c*,
- three R-proclets *R-operators*, *R-merge-b* and *R-exit-c*.

All the labels of these proclets are unique, e.g., the enqueuing transition of *Q-a1:b'* has label enq^{qid1}, while the enqueuing transition of *Q-a2:b* has label enq^{qid2}. The Q- and R-Proclets of the system have the initial markings, thereby composing the initial marking m_v of the whole system. Although the places of the Q- and R-Proclets are not shown in Figure 6.11, we can derive their markings from the provided template parameters, using the templates in Figures 6.9 and 6.10, as follows:

- $m_v(capacity^{qid1}) = [qid1^{size^{qid1}}]$ (proclet *Q-a1:b'*),
- $m_v(queue^{qid1}) = [(qid1, [\,])]$ (proclet *Q-a1:b'*),
- $m_v(idle^{\{rid1, rid2\}}) = [rid1, rid2]$ (proclet *R-operators*),
- and so on.

Note, the places of the P-proclet do not have any tokens initially, because its tokens are generated by the transitions without preset ($t1$ and $t4$).

Further, each start-complete pair of the P-proclet is connected with a start-complete pair of one R-proclet, e.g., $(t1, t2)$ is connected with $(t12, t13)$ by channels $(t1, t12)$ (labeled as $a1_s$) and $(t2, t13)$ (labeled as $a1_c$) respectively. Similarly, each complete-start pair of the P-proclet is connected with a complete-start pair of one of the Q-proclets, e.g., $(t2, t3)$ is connected with $(t10, t11)$ by channels $(t2, t10)$ (labeled as $a1_c$) and $(t3, t11)$ (labeled as b'_s) respectively.

As a consequence of point 3 of Definition 6.7, each R-proclet is connected (via channels) with at least one start-complete pair of the P-proclet, while each Q-proclet is connected with exactly one complete-start pair of the P-proclet. Note, in each channel (t, t') the first element t is always a transition of the P-proclet, and the second element is always a transition of a Q- or R-proclet, according to point 3 of Definition 6.7. That is required to avoid defining multiple channels between the same pair of transitions, e.g., (t, t') and (t', t).

6.5 Semantics of PQR-Systems

In this section, we formally define replay semantics of PQR-systems that states when an event table (Definition 2.8) forms an execution that is also described by the PQR-system, i.e., can be replayed by the PQR-system. For that, we first discuss the replay semantics for the P-, Q- and R-proclets, and then define it for the PQR-system.

6.5.1 Replaying a Trace Over a CPN

As each of P-, Q- and R-proclets is a CPN, we first consider when a trace can be replayed over a CPN. For example, when trace $\sigma = \langle e3*, e4*, e9*, e10* \rangle$, derived from the event table *ET*, shown in Table 6.1, for case notion *RID* and identifier *rid*3, can be replayed over the R-proclet *R-merge-b* (Figure 6.6). For a CPN N and an event sequence of the trace, we assume that the attribute names of each event match the names of free variables of the corresponding transition in the CPN. The trace can be replayed if event attribute names and values match a binding of this transition occurrence. To be able to match an event sequence in a trace to a sequence of transition occurrences by matching activity names in event sequence to transition labels, we define a *labeled* transition occurrence and labeled transition occurrence sequence. Then, we define how to interpret a single event as a labeled transition occurrence, and a sequence of events in a trace as a labeled transition occurrence sequence. Finally, we define that a trace can be replayed over the CPN if its interpretation as a labeled transition occurrence sequence under certain conditions *is* a labeled occurrence sequence of the CPN.

6.5.1.1 Labeled Transition Occurrences

In Chapter 2, we introduced occurrences of CPN transitions. However, such occurrences do not preserve transition labels, which are needed to match events to transitions by the activity attributes and the transition labels. Now, we define *labeled* transition occurrences and their sequences. The definition of a labeled transition occurrence reads as follows.

Definition 6.8 (Labeled transition occurrence)**.** *Let N be a CPN, and let $m \xrightarrow{(t,bind)@time} m'$ be an occurrence of transition t with binding $bind = \langle (var^1, val^1), \ldots, (var^n, val^n) \rangle$ from state $(m, time)$ with marking m to state $(m', time)$ with marking m'. We call $m \xrightarrow{(\ell(t),bind)@time} m'$ a* labeled transition occurrence *of the transition labeled $\ell(t)$ with binding $bind$ from state $(m, time)$ to state $(m', time)$.*

Now, we can lift Definition 6.8 to a sequence of transition occurrences by turning each of its occurrences into a labeled one by replacing transition identifiers with transition labels.

Definition 6.9 (Labeled transition occurrence sequence)**.** *Let N be a CPN, and let $\sigma = m_1 \xrightarrow{(t_1,bind_1)@time_1} m_2 \ldots m_m \xrightarrow{(t_m,bind_m)@time_m} m_{m+1}$ be transition occurrence sequence with markings $m_1 \ldots m_{m+1}$ and bindings $bind_1 \ldots bind_m$ from state $(m_1, time_1)$ to state $(m_{m+1}, time_m)$. A sequence $m_1 \xrightarrow{(\ell(t_1),bind_1)@time_1} m_2 \ldots m_m \xrightarrow{(\ell(t_m),bind_m)@time_m} m_{m+1}$ is a* labeled transition occurrence sequence *with markings $m_1 \ldots m_{m+1}$ and bindings $bind_1 \ldots bind_m$ from state $(m_1, time_1)$ to state $(m_{m+1}, time_m)$.*

6.5.1.2 Replaying Traces Over CPNs

In processes, executing a process step generates an event. We can interpret a generated event e (Definition 2.8) as a labeled transition occurrence. For replaying, we can consider the attribute-value pairs (Section 2.4) defined by an event as a binding of values to variable names.

Definition 6.10 (Binding from event)**.** *Let ET be an event table and e be an event with attributes $an^1 \ldots an^n$ defined, i.e., $\#_{an^i}(e) \neq \perp$. We call a sequence of all the pairs of the attribute names and the corresponding attribute values $\langle (an^1, \#_{an^1}(e)), \ldots, (an^n, \#_{an^n}(e)) \rangle$ a* binding from event *e. We write $bind_e$ for the binding from event e.*

Using such a binding from events, we can interpret an event as an occurrence of a labeled transition as follows.

Definition 6.11 (Event as a labeled transition occurrence)**.** *Let N be a CPN, let m and m' be markings of this CPN, let e be an event in $\mathcal{E}$, and $bind_e$ be the binding from e according to Definition 6.10. Let the state $(m, \#_{\text{time}}(e))$ of N with marking m. Let function $\beta : Act \to \Sigma$ be a mapping between activity names and transition labels of*

Event	Activity	Time	pid	qid	rid
$e4*$	b_c	12	$pid2$	$qid3$	$rid3$
$e10*$	b_c	22	$pid1$	$qid3$	$rid3$
$e5*$	c_s	24	$pid2$	$qid3$	$rid4$
$e11*$	c_s	34	$pid1$	$qid3$	$rid4$

Table 6.3: Trace σ_5 of the queue with identifier $qid3$ of Table 6.1

N. The interpretation of event e as a labeled transition occurrence is the occurrence $m \xrightarrow{(\#_{\text{act}}(e), bind_e)@\#_{\text{time}}(e)} m'$ *of a transition labeled* $\#_{\text{act}}(e)$ *(according to Definition 6.8) with binding* $bind_e$ *from state* $(m, \#_{\text{time}}(e))$ *to state* $(m', \#_{\text{time}}(e))$ *with marking* m'.

The above definition uses a mapping β between activity names and transition labels and assumes that the other attribute names exactly match the variables. In reality, systems record events whose attributes rarely exactly match free variables of CPNs modeling those systems, so we assume there is a mapping $anToVars : AN \rightarrow Var$ between the attribute names and free variables. Note, we also can ignore variables in bindings that do not occur in the arc expressions of the firing transition (the events can have more attributes than needed for the transition).

Next, we lift Definition 6.11 to an event sequence by interpreting each event as a labeled transition occurrence.

Now, we define when a trace can be replayed over a CPN, using Definition 6.9.

Definition 6.12 (Replaying a trace over a CPN)**.** *Let* σ *be a trace* $\langle e_1, \ldots, e_n \rangle$. *Let N be a CPN, let* $m_1 \ldots m_{m+1}$ *be its markings. Let function* β *be a mapping between activity names and transition labels of N, as defined in Definition 6.11. We can replay trace* σ *on CPN N iff the sequence*
$(m_1, time_1) \xrightarrow{(\beta(\#_{\text{act}}(e_1)), bind_{e_1})@time_1} (m_2, time_1) \ldots$
$(m_m, time_m) \xrightarrow{(\beta(\#_{\text{act}}(e_m)), bind_{e_m})@time_m} (m_{m+1}, time_m)$ *is a labeled occurrence sequence of N from state* $(m_1, \#_{\text{time}}(e_1))$ *to state* $(m_{m+1}, \#_{\text{time}}(e_m))$ *with markings* $m_1 \ldots m_{m+1}$ *and bindings* $bind_{e_1} \ldots bind_{e_m}$, *where* $\forall 1 \leq i \leq m, time_i \leq \#_{\text{time}}(e_i)$.

We explain the definitions above by an example. The event table in Table 6.3 contains the subset of events of the event table in Table 6.1, related to queue $qid1$. These result in the following trace σ_5 for case notion qid in instance $qid1$: $\sigma_5 = \langle e4*, e10*, e5*, e11* \rangle$.

We now replay trace σ_5 over the Q-proclet *Q-b:c* of the PQR-system of Figure 6.7, where we assume that *Q-b:c* was instantiated from the template in Figure 6.10 with the following parameters: $QID = qid3$, $capacity = 3$, $t_{\text{wQ}} = 10$. Let us define the mapping $\beta = \{(b_c, enq^{qid3}), (c_s, deq^{qid3})\}$

Let us assume state $\psi_1 = (m_1, 12)$ with marking

i	t_{rep}	Event	$m_i(queue^{qid3})$	i^{th} labeled occurrence	$m_{i+1}(queue^{qid3})$
1	12	$e4*$	$[(qid3, \langle\rangle)]$	$\langle enq^{qid3}\langle(\text{qid}, qid3), (\text{act}, b_c), (\text{pid}, pid2)\rangle@12$	$[(qid3, \langle pid2\rangle)]$
2	22	$e10*$	$[(qid3, \langle pid2\rangle)]$	$\langle enq^{qid3}\langle(\text{qid}, qid3), (\text{act}, b_c), (\text{pid}, pid1)\rangle@22$	$[(qid3, \langle pid2, pid1\rangle)]$
3	22	$e5*$	$[(qid3, \langle pid2, pid1\rangle)]$	$\langle deq^{qid3}\langle(\text{qid}, qid3), (\text{act}, c_s), (\text{pid}, pid2)\rangle@24$	$[(qid3, \langle pid1\rangle)]$
4	32	$e11*$	$[(qid3, \langle pid1\rangle)]$	$\langle deq^{qid3}\langle(\text{qid}, qid3), (\text{act}, c_s), (\text{pid}, pid1)\rangle@34$	$[(qid3, \langle\rangle)]$

Table 6.4: Labeled occurrences and markings.

1. $m_1(queue^{qid3}) = [(qid3, \langle\rangle)]$,
2. $m_1(capacity^{qid3}) = [qid3^3]$,
3. $m_1(ready^{qid3}) = [\,]$.

Assuming start in state ψ_1, the events of σ_5 define the following transition occurrences according to Definition 6.9 (the markings are omitted):

- $\langle enq^{qid3}\langle(\text{qid}, qid3), (\text{act}, b_c), (\text{pid}, pid2), (\text{rid}, rid3), (\text{time}, 12)\rangle@12$,
- $enq^{qid3}\langle(\text{qid}, qid3), (\text{act}, b_c), (\text{pid}, pid1), (\text{rid}, rid3), (\text{time}, 14)\rangle@14$,
- $deq^{qid3}\langle(\text{qid}, qid3), (\text{act}, c_s), (\text{pid}, pid2), (\text{rid}, rid4), (\text{time}, 24)\rangle@24$,
- $deq^{qid3}\langle(\text{qid}, qid3), (\text{act}, c_s), (\text{pid}, pid1), (\text{rid}, rid3), (\text{time}, 34)\rangle@34\rangle$.

Let us now derive a labeled occurrence sequence from state ψ_1, using the transition occurrence order and bindings (for the free variables) in σ_5. Table 6.4 shows the occurrences and markings for place $queue^{qid3}$ before and after each occurrence (columns m_i and m_{i+1} respectively). The CPN state before and after each occurrence can be derived as the marking before or after the occurrence, and the occurrence time, e.g., the state before the occurrence in the top row is $([(qid3, \langle\rangle)], 12)$. Note, in the table the attributes rid and time, and the markings of places $capacity^{qid3}$ and $ready^{qid3}$ are omitted for clarity.

Event $e4*$ describes the binding $bind_{e4*} = \langle(\text{qid}, qid3), (\text{act}, b_c), (\text{pid}, pid2)\rangle$, according to Definition 6.10. By binding variable `pid` to value $pid2$, variable `qid` to value $qid3$, and variable `q` to value $\langle\rangle$ (available through marking m_1), transition $\beta(b_c) = enq^{qid3}$ becomes enabled and fires at time $\#_{\text{time}}(e4*) = 12$, resulting in marking $m_2(queue^{qid3}) = [(qid3, \langle pid2\rangle)]$.

Next, we bind variable `pid` to value $pid1$, variable `qid` to value $qid3$, and variable `q` to value $\langle pid2\rangle$ (according to binding $bind_{e10*}$), transition enq^{qid3} becomes enabled and fires at time 22, resulting in marking $m_3(queue^{qid3}) = [(qid3, \langle pid2, pid1\rangle)]$.

At time 22, token $pid2$ becomes ready on place $ready^{qid3}$. The binding from $e5*$ provides values $qid3$ and $pid2$ for variables `qid` and `pid` respectively, so transition deq^{qid3} becomes enabled and fires (event $e5*$), removing element $pid2$ from the queue on place $queue^{qid3}$.

Similarly, at time 32, token $pid1$ becomes ready on place $ready^{qid3}$. The binding from $e11*$ provides values $qid3$ and $pid1$ for `qid` and `pid`, so deq^{qid3} fires, removing

the last element $pid1$ from the queue on place $queue^{qid3}$, and resulting in marking $m_5(queue^{qid3}) = [(qid3, \langle\rangle)]$.

As the derived occurrence sequence is indeed the labeled transition occurrence, and each occurrence time (column t_{rep}) is less or equal to the corresponding event timestamp $\#_{\text{time}}(e_i*)$, trace σ (Table 6.3) can be replayed over proclet *Q-b:c*. Similarly, we can show that each trace of the event table, presented in Table 6.1, can be replayed over the corresponding proclet of the PQR-system.

6.5.2 Replaying an Event Table Over a PQR-System

Previously, we introduced the semantics of the P-, Q- and R-proclets through the replay of event traces. Now, we introduce the overall semantics of PQR-systems by combining the semantics of its proclets and adding correlation constraint semantics. In the following, we define that a given event table ET can be replayed over all the proclets of a PQR-system iff:

1. all traces in the event table can be replayed over all the P-, Q-, and R- proclets in the PQR-system individually (Definition 6.13), and
2. all the traces together satisfy the synchronization and correlation constraints of the PQR-system (Definition 6.14).

Definition 6.13 (Replaying an event table over a set of P-, Q- and R-proclets)**.** *Let CN_{PQR} be the set of case notions of PQR-systems, $CN_{\text{PQR}} = \{\text{pid}, \text{qid}, \text{rid}\} \subseteq CN$. Let $ET = (CN_{\text{PQR}}, E, \#), E \subseteq \mathscr{E}$ be an event table. Event table ET can be* replayed *over the P-, Q- and R-proclets of the PQR-system $S = (\{N_1, \ldots, N_n\}, m_v, C, \ell^C)$ iff:*

1. *case notions* pid *and* rid *are global in event table ET according to Definition 2.8, i.e., each event $e_i \in E$ is related to a process with some identifier $pid \in \mathscr{I}$ and a resource with some identifier $rid \in \mathscr{I}$,*
2. *for each trace $\sigma_{\text{pid}} = \langle e_1, \ldots, e_n \rangle$ in an event log L^{ET}_{pid} derived from ET for case notion* pid *(Definition 2.12), all events $e_2, \ldots, e_{n-1}$ are related to a queue with some identifier $qid \in \mathscr{I}$, i.e., $\#_{\text{qid}}(e_i) = qid \neq \perp, 1 < i < n$,*
3. *for each value $p \in \mathscr{I} \mid \exists e \in E \mid \#_{\text{pid}}(e) = p$, each trace $\sigma_{\text{pid}} \in L^{ET}_{\text{pid}}$ with identifier p can be replayed in the P-proclet of system S. For each event e_j, we write $t_P(e_j)$ for the P-proclet transition that fires when replaying e_j,*
4. *for each value $q \in \mathscr{I} \mid \exists e \in E \mid \#_{\text{qid}}(e) = q$, each trace σ_{qid} in an event log L^{ET}_{qid} derived from ET for case notion* qid *can be replayed in a Q-proclet of S. For each event e_j, we write $t_Q(e_j)$ for the Q-proclet transition that fires when replaying e_j,*
5. *for each value $r \in \mathscr{I} \mid \exists e \in E \mid \#_{\text{rid}}(e) = r$, each trace σ_{rid} in an event log L^{ET}_{rid} derived from ET for case notion* rid *can be replayed in an R-proclet of S. For each event e_j, we write $t_R(e_j)$ for the R-proclet transition that fires when replaying e_j.*

For example, the event table ET in Table 6.1 can be replayed over the P-, Q- and R-proclets of the system in Figure 6.11 because:

1. case notions pid and rid are global for ET,
2. all the events of traces σ_1 and σ_2 of case notion pid (Table 6.2), except events $e1*, e6*, e7*, e12*$ of the bordered transitions, are related to a queue, i.e., they have defined values of attribute qid,
3. each trace can be replayed over the corresponding proclet, as we showed in Section 6.5.1 for trace σ_5. The same can be shown for the other traces as well.

If event table ET can be replayed over a set of P- Q- and R- proclets, then any event $e \in ET$ by Definition 6.13:

1. defines a channel $c_R = (t_P(e_j), t_R(e_j))$,
2. and if additionally ${}^\bullet t_P(e_j) \neq \emptyset$ or $t_P(e_j)^\bullet \neq \emptyset$ it also defines channel $c_Q = (t_P(e_j), t_Q(e_j))$ so that labels of c_R and c_Q are the same, i.e., $\ell^{\mathrm{C}}(c_R) = \ell^{\mathrm{C}}(c_Q)$.

Definition 6.14 (Execution of a PQR-system)**.** *Let $ET = (CN_{\mathrm{PQR}}, E, \#), E \subseteq \mathscr{E}$ be an event table. Event table ET is the* execution *of the PQR-system $S = (\{N_1, \ldots, N_n\}, m_v, C, \ell^C)$ iff (1) it can be replayed over the P-, Q- and R-proclets of S according to Definition 6.13, and (2) the following holds:*

1. *in any trace $\sigma_{\mathrm{pid}} = \langle e_1, \ldots, e_n \rangle$ in an event log L^{ET}_{pid} derived from ET for case notion* pid *(Definition 2.12) if there exists e_j with $\ell_P(t_P(e_j)) = (a, complete, tag)$ then for the directly preceding event e_{j-1} holds*

   (a) $\#_{\mathrm{rid}}(e_j) = \#_{\mathrm{rid}}(e_{j-1})$,
   (b) $\#_{\mathrm{time}}(e_{j-1}) < \#_{\mathrm{time}}(e_j)$.

   That is, step a is completed by the same resource that started the step in the previous event.
2. *in any trace $\sigma_{\mathrm{pid}} = \langle e_1, \ldots, e_n \rangle \in L^{ET}_{\mathrm{pid}}$ if there exists $e_j, 2 < j < n$ with $\ell_P(t_P(e_j)) = (a, start, tag)$ then for the directly preceding event e_{j-1} holds*

   (a) $\#_{\mathrm{qid}}(e_j) = \#_{\mathrm{qid}}(e_{j-1})$,
   (b) $\#_{\mathrm{time}}(e_{j-1}) < \#_{\mathrm{time}}(e_j)$,

   i.e., there is a queue $\#_{\mathrm{qid}}(e_j)$ between the resource that completed the previous step $\#_{\mathrm{act}}(e_{j-1})$, and the resource that completed step ($\#_{\mathrm{act}}(e_j)$) (i.e., step a).

By Definition 2.8, in event table $ET = (CN_{\mathrm{PQR}}, E, \#)$, at most one identifier of each case notion in CN_{PQR} is presented for each event. This means that each synchronized event e^*_i synchronizes at most one proclet of each proclet type of the PQR-system. That corresponds to $1:1$ cardinality constraint, defined in [48].

Let us show that the event table ET presented in Table 6.1 can be replayed over the PQR-system of Figure 6.11. In the event table, case notions pid and rid are global,

while case notion qid is defined for all the events of instances $pid1, pid2$ of case notion pid, except the first and last events in their traces. That is, the event table satisfies points 1 and 2 of Definition 6.13. Further, as shown in Section 6.5.1, each trace can be replayed over the corresponding proclet, i.e., *ET* satisfies points 3-5 in Definition 6.13, thereby satisfying point (1) in Definition 6.14. Then, for each *complete* event of the traces $pid1$ and $pid2$, there exists a directly preceding *start* event with the same value of attribute rid, e.g., for event $e2*$, there exists the directly preceding event $e1*$ such that $\#_{\text{rid}}(e1*) = \#_{\text{rid}}(e2*) = rid1$ (point (2)1 of Definition 6.14). Finally, for each *start* event of the traces $pid1$ and $pid2$ (except the events of bordered transitions $t1$ and $t4$, i.e., $e1$ and $e6*$), there exists a directly preceding *start* event with the same value of attribute qid, e.g., for event $e3*$, there exists the directly preceding event $e2*$ such that $\#_{\text{qid}}(e2*) = \#_{\text{qid}}(e3*) = qid1$ (point (2)2 in Definition 6.14). So, we conclude event table *ET* can be replayed over the PQR-system, i.e., it is an execution of the PQR-system in Figure 6.7.

6.6 Properties of PQR-Systems

In the previous sections, we defined the PQR-system and its semantics. Now, we provide some intuition on the PQR-system properties.

A "classical" marked Petri-net (see Definition 2.2) has well-studied properties. For example, it can be *bounded*, *safe*, and *deadlock free*. Its transitions can be *live*, and markings can be *reachable*, and be explored via a *reachability graph*. A workflow net (see Definition 2.5) is *sound* if it has properties of safeness, *proper completion*, *option to complete*, and *absence of dead parts*. Knowing these and other properties allows us to understand how well a model captures the modeled process and if the model contains errors. We refer to [1] for details.

Understanding such properties of a PQR-system is undoubtedly valuable for modeling MHSs. The P-proclet can be converted into a sound workflow net by Definition 6.2. However, does it make the entire PQR-system sound? No, it does not because it also consists of the timed CPNs of the Q- and R-proclets, and the synchronization channels. So, the properties of the entire PQR-system cannot be "inherited" from its P-proclet only. Moreover, the meaning of soundness may be different than soundness of marked Petri nets.

Problems of checking the properties of various "non-classical" process models have been addressed in many works. Thus, the properties of various workflow nets with resources have been studied in [165, 166, 167], *time-soundness* of timed CPNs has been suggested in [168], and soundness of object-centric Petri nets [120] has been proposed in [169].

Works [168] and [169] are particularly interesting because (1) soundness is perhaps the most critical property, and (2) these works study nets with characteristics, of which some are close to the PQR-system ones. That is,

- the Q- and R-proclets are timed CPN, and
- each PQR-system proclet can be seen, to a certain extent, as a sub-net of an object-centric Petri net with a colorset corresponding to the entity that the proclet model, but without the ability to distinguish the objects on the places.

In [168, 169], soundness has been redefined to address the model analysis purposes. Thus, time-soundness guarantees the deterministic behavior of the system, disregarding the time of transition occurrences, and soundness of object-centric Petri nets is checked for a single object but allows for the presence of tokens of the other objects on the net places.

Building on these works, how can soundness of the PQR-system be understood? Let us provide some intuition, assuming the P-proclet is converted into a workflow net.

- Similarly to soundness of object-centric Petri nets, proper completion and option to complete must hold per individual *token of interest* of the P-proclet, yet allowing other tokens on the P-proclet places after completion.
- Each resource of the R-proclets, engaged in handling the token of interest, must be released afterward, i.e., the resource token must return on place *idle*.
- Each capacity element of the Q-proclets, engaged in handling the token of interest, must be released, i.e., the capacity token must return on place *capacity*.
- Proper completion and option to complete for the token of interest must hold, disregarding the actual waiting and service time of the Q- and R-proclets.

Another property of crucial importance for PQR-systems is deadlock free, which guarantees that at least one transition is enabled for each reachable marking. However, a reachability graph for a PQR-system is infinite due to countless combinations of timestamps for markings of the Q- and R-proclets.

To conclude, the definitions and decidability of the PQR-system properties are unknown yet, and subject to future work.

6.7 Demonstration

In the previous sections, we define the PQR-system and its semantics formally. In this section, we demonstrate how:

1. a BHS can be modeled using a PQR-system, and
2. what kind of event data it generates,

by introducing a *simulation model* of an airport terminal BHS. Besides demonstration purposes, its fully controlled environment is used to evaluate our methods, presented in the next chapters, on synthetic datasets so that the ground truth is known.

The MFD of the BHS we modeled is shown in Figure 6.14. It includes the preliminary sorting loop, the most critical (performance-wise) part of medium- and large-sized BHSs, that works as follows.

- Incoming baggage enters the system at locations $a_0^1 - a_0^7$. In particular, bags from the terminal checked-in counters enter the system at $a_0^1 - a_0^4$, transferred bags enter it at enter at $a_0^5 - a_0^6$, and bags from the other terminals come via a_0^7.
- After entering the system, the bags travel toward the preliminary sorting loop (the red circle in the middle of the MFD), where they merge it at $a_4^1 - a_4^7$.
- The preliminary sorting loop serves for:
  - diverting bags for screening to scanners at zone S,
  - arranging a queue of bags when diverting toward S is impossible,
  - diverting bags toward their destinations $b_1^1 - b_1^6$ via $b_0^1 - b_0^6$,

  For example, if a bag cannot be diverted toward S because there is no free space, it starts recirculating on the loop till diverting is possible.
- For evaluation purposes, we model the sorting loop not as a solid belt but as a circle conveyor consisting of three independent belts, connected at locations x, y, and z.
- At destinations $b_1^1 - b_1^6$, bags leave the system.

Note, in this system, we do not model many system areas, such as check-in counters and islands, early bag store, final sorting loops, and so on for simplicity.

Using this MFD, we designed a corresponding PQR-system, providing the minimum resource service time t_{sR}, minimum resource waiting time t_{wR}, and minimum queue waiting time t_{wQ} parameters for the Q- and R-proclets. These parameters are chosen such that they repeat the proportions of the travel time of a real BHS, not revealing their exact values due to confidentiality.

The resulting PQR-system is quite large, so we present it here in a compact way. Figure 6.15(a) shows a "zoomed-out" model, where the Q- and R-proclets are collapsed into the blue and green boxes, and no place and transition labels are shown. The model layout repeats the MFD layout, to make its reading easier.

Its "zoomed-out" fragment for process step IN_5_3-x (a_4^5 in the MFD) is shown in Figure 6.15(b). Note, the simulation model historically uses prefixes IN and OUT instead of a and b respectively, and underscores in label names to make it easier to record textual event logs. Additionally, labels include x, y, and z for merging and diverting units, to show what part of the sorting loop is connected. Figure 6.15(c) illustrates this labeling for the bottom model part.

Using this PQR-system as input, we developed

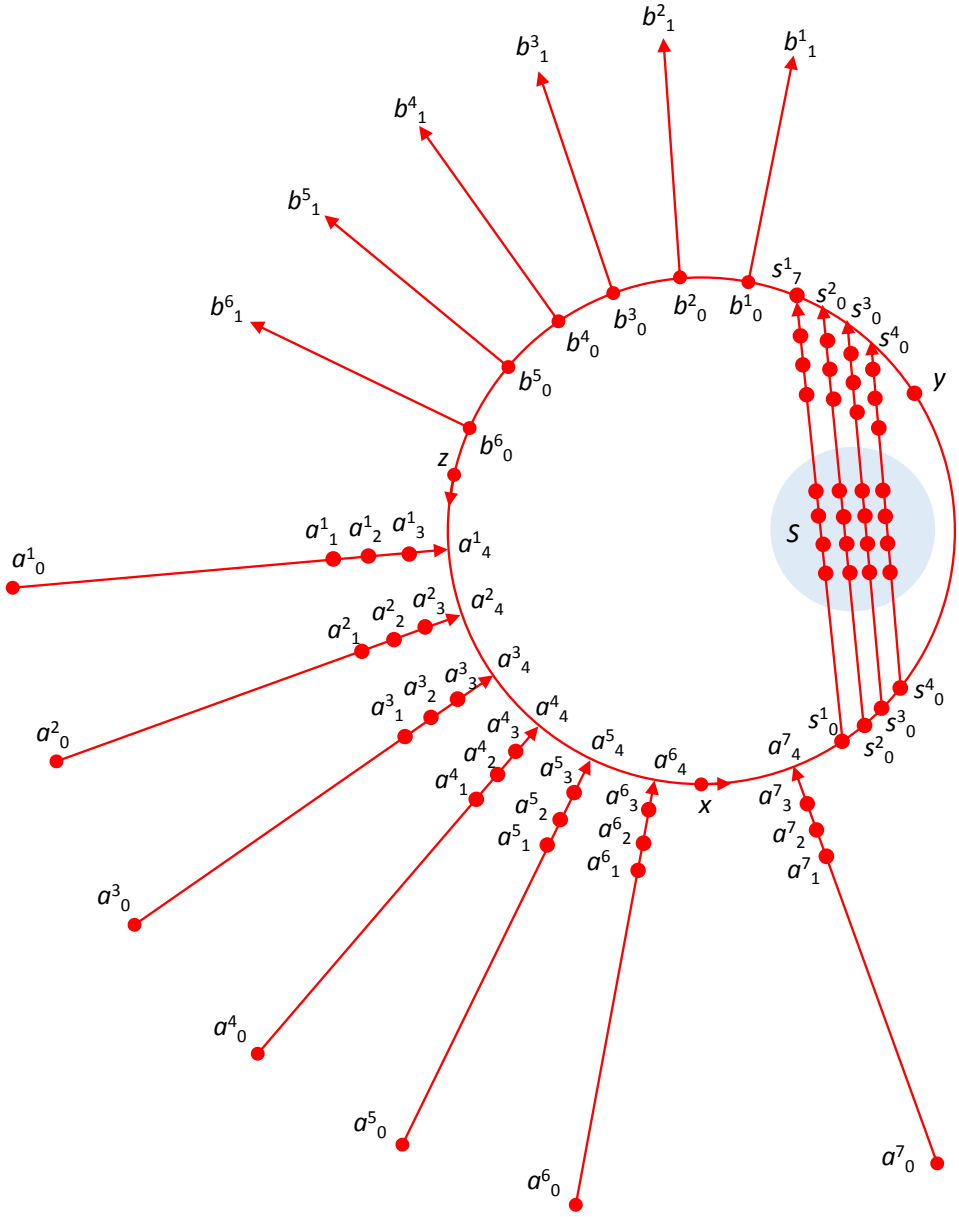


Figure 6.14: Material flow diagram.

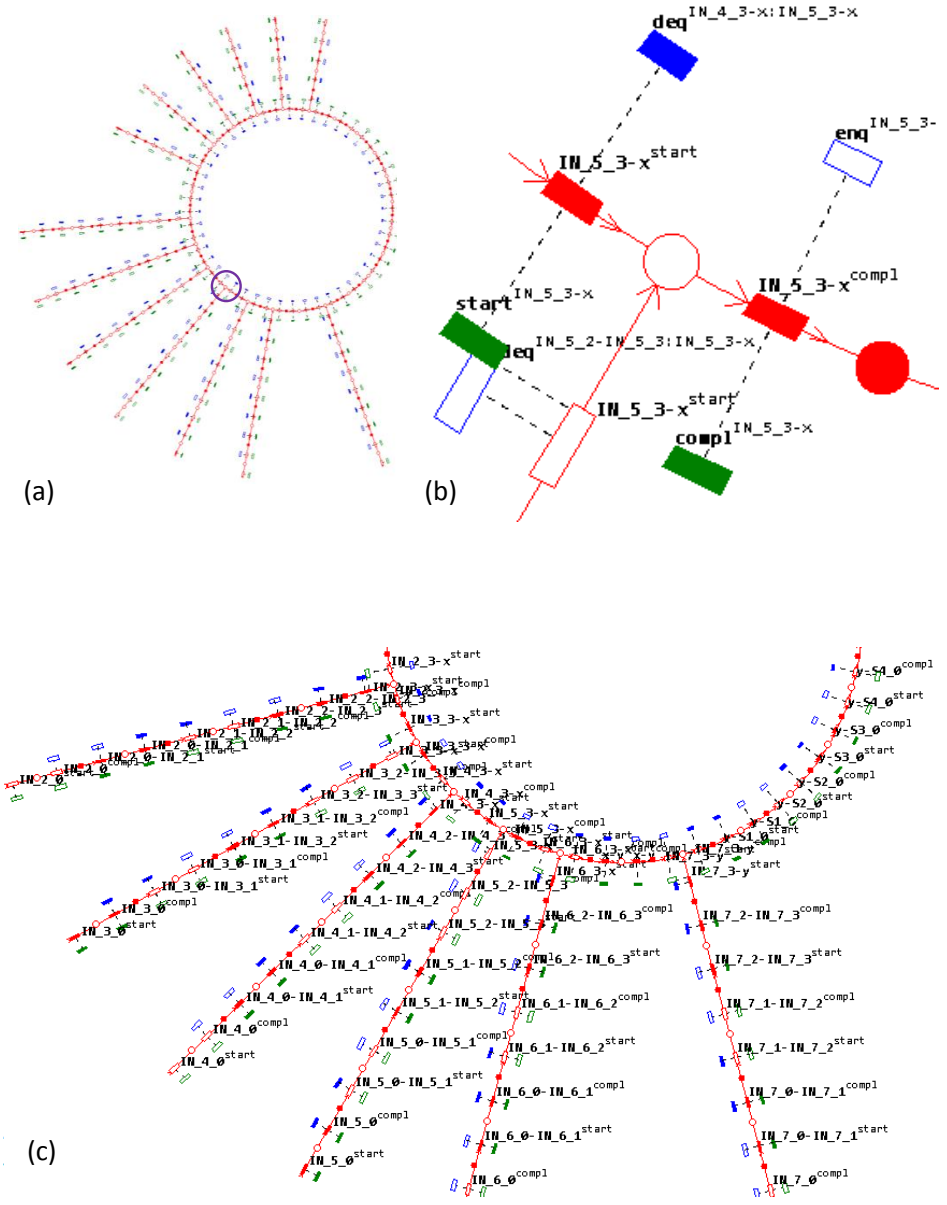


Figure 6.15: PQR-system visualization.

1. a simulation model that implements a BHS modeled by the PQR-system in Figure 6.15,
2. an API to define a simulation scenario,
3. and a control panel that
    - visualizes the system stated in real-time, and
    - allows interactively modifying the running scenario, e.g., to introduce a delay.

The control panel is shown in Figure 6.16(a), where

- sliders (1,2) allow horizontal and vertical zooming,
- field (3) shows the current simulation time (from the start of the epoch),
- button (4) pauses/resumes simulation,
- text field and button (5) allow to send a command to the simulation engine to interactively block/unblock conveyors[1], and
- checkboxes (6) allow to show/hide case identifiers and final destination.

Figure 6.16(b) and (c) show bags on the conveyors when they are moving (b), and blocked (c). The bags are color-coded to show their current destination[2].

The model records both complete and incomplete event logs. In incomplete logs, some events are unobserved in the same way as they are unobserved in real event logs. It is needed for the evaluation using incomplete event data. In Figure 6.17(a), the complete event log has an event (inside the red box) that is unobserved in the corresponding incomplete log, shown in Figure 6.17(b).

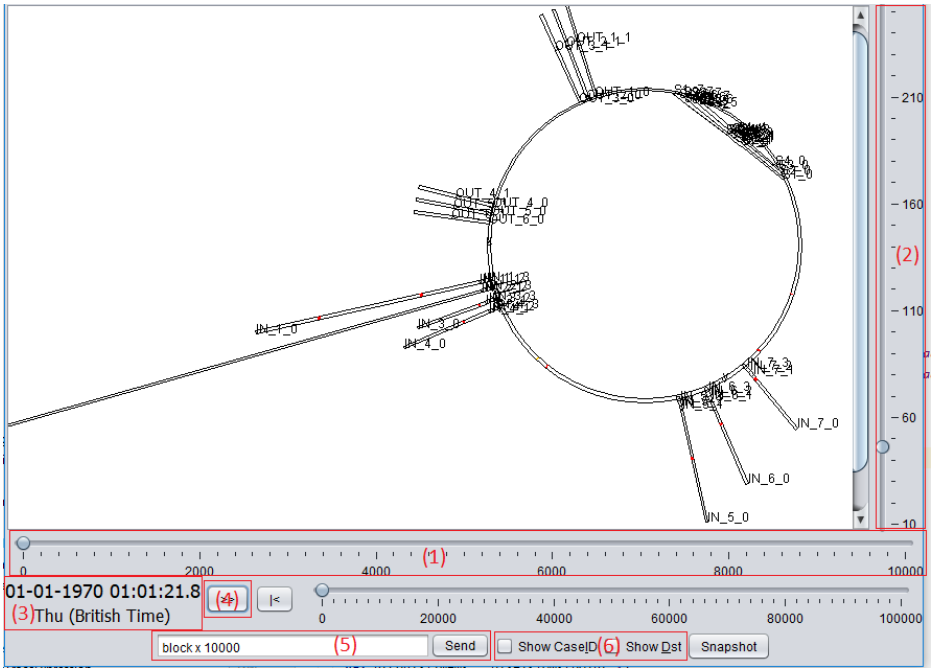
In the next chapters, we design different scenarios to generate event complete and incomplete event logs and evaluate our methods for log repair, descriptive performance analysis, and predictive process monitoring using these synthetic logs (as well as real data).

6.8 Chapter Summary

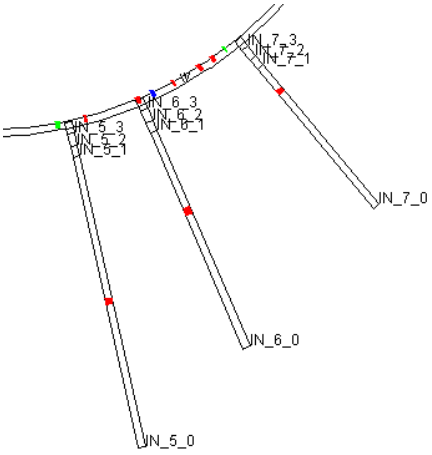
In this chapter, we considered the modeling of BHSs. We highlighted the key aspects of the complex BHS behavior and formulated the most important modeling challenges they imply. Then, we transformed the concepts and insight about the BHS behavior, obtained in Chapter 4 through queueing model- and network-based view on BHSs, into process-model level building blocks and concepts, which we informally introduced. We introduced the concepts of the synchronous proclet system [48] and showed how it could be modified to model not only the process but queue and

[1]The source code and documentation are available on `https://github.com/processmining-in-logistics/psm/tree/pqr`.

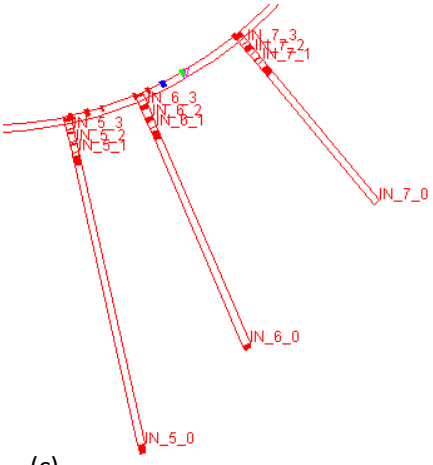
[2]The screencast showing the model animation is available on `https://youtu.be/O0_tjfRInFo`.



(a)



(b)



(c)

Figure 6.16: Simulation model control panel.

(a)

id	timestamp	activity
18	01-01-1970 01:00:46.000	IN_1_0
19	01-01-1970 01:00:51.300	IN_1_0
20	01-01-1970 01:00:54.800	IN_1_0
21	01-01-1970 01:00:57.800	IN_1_0
13	01-01-1970 01:00:59.000	IN_7_3:y
12	01-01-1970 01:00:59.500	IN_3_3:x
12	01-01-1970 01:01:01.600	IN_4_3:x
22	01-01-1970 01:01:02.700	IN_1_0
11	01-01-1970 01:01:04.400	IN_6_3:x
23	01-01-1970 01:01:04.600	IN_4_0

(b)

id	timestamp	activity
18	01-01-1970 01:00:46.000	IN_1_0
19	01-01-1970 01:00:51.300	IN_1_0
20	01-01-1970 01:00:54.800	IN_1_0
21	01-01-1970 01:00:57.800	IN_1_0
13	01-01-1970 01:00:59.000	IN_7_3:y
12	01-01-1970 01:00:59.500	IN_3_3:x
22	01-01-1970 01:01:02.700	IN_1_0
11	01-01-1970 01:01:04.400	IN_6_3:x
23	01-01-1970 01:01:04.600	IN_4_0
24	01-01-1970 01:01:04.600	IN_6_0

Figure 6.17: Complete (a) and incomplete (b) event logs.

resource dimensions of BHSs as well by extending its proclets from ν-Petri nets to CPNs, thereby introducing the process, queue, and resource synchronous proclet system (called a PQR-system). We explained its semantics through the system run. We showed how the sub-runs of the PQR-system proclet instances form the partial order in the overall system run, thereby ordering the otherwise isolated sub-runs of individual process instances (bags). Finally, we defined the PQR-system formally, providing the definitions of all its proclet types, meaning the process, queue, and resource proclets, the system itself, including synchronization between its entities, and constraints, and the PQR-system replay semantics.

The resulting formal process model serves as a backbone for the process mining techniques in the remainder of this thesis, concretely, for conformance checking and log repair in Chapter 7, for multi-dimensional process performance analysis and monitoring in Chapter 8, meaning the process, queues, and resources as the dimensions, and finally, for predictive performance monitoring in Chapter 9.

Chapter 7

Conformance Checking for Systems with Shared Resources and Queues

The preceding chapter introduced the PQR-system for modeling systems with shared resources and queues and its replay semantics. This chapter considers the problem of relating event data to the PQR-system. In process mining, conformance checking is used to identify inconsistencies between a process model and data. Typically, identified inconsistencies are used to repair either the process model or data, i.e., one of them is assumed to be fully trusted. However, here we go beyond this approach and adapt the concept of generalized conformance checking [44]. It assumes that neither model nor the log can be fully trusted. That fits our needs better because we usually trust neither while analyzing material handling processes.

To enable generalized conformance checking in our setting, we propose a PQR-system-based conformance checking pipeline in Section 7.1, consider PQR-system-based conformance checking in Section 7.2, and our novel approach for inferring unobserved events in systems with shared resources and queues in Section 7.3.

7.1 Motivation for Generalized Conformance Checking

Conformance checking is one of the main process mining tasks [1]. It addresses the problem of relating event data to the process model. Conformance checking aims to detect all situations when the process behavior, captured through the event data,

deviates from the behavior described by the process model. Each detected deviation is potentially an input for other process mining tasks, such as:

- *model repair* that aims to fit the process model better in the behavior described by the data,
- *log repair* that aims to remove noise from the event data, and reconstruct missing events,
- *local diagnostics* to investigate the nodes in the model where model and log disagree (e.g., to detect desirable and undesirable deviations for prescriptive models),
- *explaining outliers* with decision mining [170],
- and so on.

As a result, detected outliers can be used for obtaining a more accurate model, cleaner event data for further process analysis, explaining outliers, etc.

Let us consider the classical conformance checking settings and approach (see Figure 7.1(a)). The input is:

- an event log,
- a Petri net with black tokens.

During conformance checking, the event log is analyzed *trace by trace*. As Petri nets have token replay semantics, each trace is replayed over the model. If it can be replayed, there is no deviation. Otherwise, trace alignment [43] is used to identify whether events in the log and/or transitions in the model are missing. However, how to determine if the detected deviations require model or log repair? For that, one of the following interpretations of the given process model is usually assumed.

1. **The event log is fully trusted.** In this case, a process model is considered as *descriptive* [24], and the problem of conformance checking reads as follows.
    - How to determine if a process model S *describes* the behavior of an actual process/system $\mathscr{S}$ represented by an execution EX, i.e., if the model describes the observed (through the data) behavior accurately?
2. **The process model is fully trusted.** In this case, the process model is considered as *prescriptive* [24], and the problem of conformance checking reads as follows.
    - How to determine if an execution EX fits into the behavior of an actual process/system $\mathscr{S}$ *prescribed* by a process model S, i.e., if these data can be generated by the actual (modeled) process/system?

So, the detected deviations can be used for either model repair or log repair tasks for a descriptive and prescriptive model respectively. In Figure 7.1(a), arrows (1) and

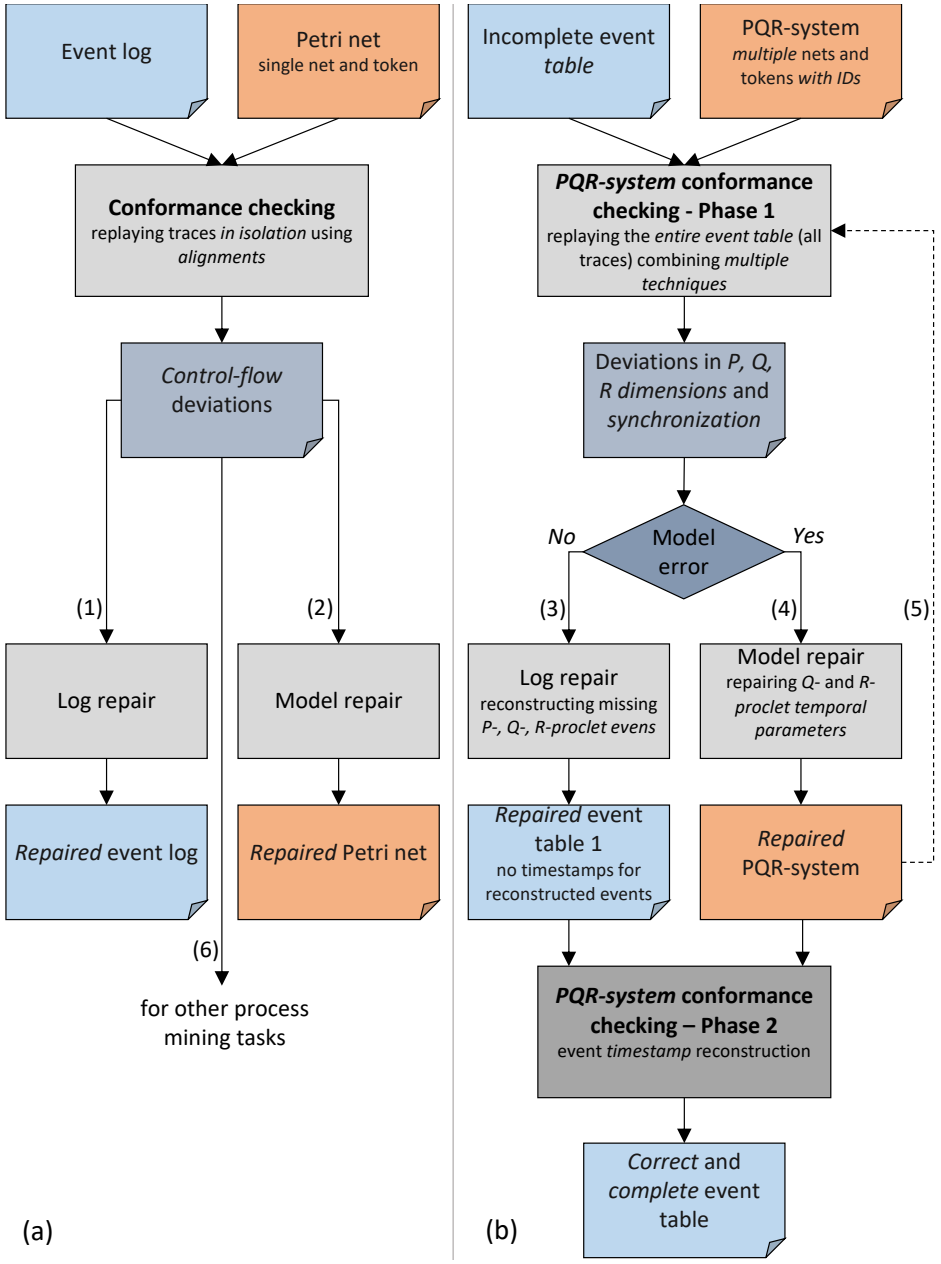


Figure 7.1: Classical conformance checking (a) and generalized conformance checking for the PQR-system-based setting (b).

(2) from Control-flow deviations point to Log repair and Model repair respectively, resulting in a Repaired event log or Repaired process model.

Generalized Conformance Checking. In practice, the analyst rarely has a fully trusted event log or process model. Often, neither can be fully trusted. To overcome this situation, the concept of *generalized conformance checking* is suggested in [44]. Generalized conformance checking goes beyond the detection of deviations, it *categorizes* them as:

- event log anomalies,
- process model errors,
- unsolvable inconsistencies.

That is, generalized conformance checking unites the three tasks of model repair, log repair, and conformance checking under a common roof. So, it aims at altering both model and log, so paths (1) and (2) in Figure 7.1(a) can be followed in *parallel*, leading to a "better" process model and event log *together*.

In the following, we consider which type of conformance checking is actual for the MHS domain, and what conformance checking problems for systems with shared resources and queues are actual. In the remainder of this chapter, we use the term BHS for clarity of examples.

Conformance Checking of Systems with Shared Resources and Queues. Material handling processes can be modeled using the PQR-system that we introduced in Chapter 6. Later in this thesis, we use PQR-systems, as well as event data in the form of an event table (see Definition 2.8) as input for approaches answering **AQ1**-**AQ8** (Section 4.2). So, a more accurate process model or event table would lead to more accurate analysis results. However, for what relating event data, recorded by a BHS, to the corresponding PQR-system can be used in practice? Let us consider several use cases.

For instance, when a bag's trace cannot be replayed over the control-flow model (P-proclet), this situation can indicate a sensor malfunctioning, manual intervention in the process, or information loss due to incomplete logging. For these use cases, the process model can be considered prescriptive (due to the physical constraint for possible process step sequences), and log repair can be done to reconstruct missing events.

In contrast, other situations can indicate model errors. For instance, if a resource trace cannot be replayed over the model (R-proclet), it can indicate *concept drift*. That is, the configuration of the modeled machine is changed, e.g., because a new protective space policy (see Section 1.1.2) is engaged. So, the model no longer describes the system accurately. In this case, the process model can be considered descriptive, and model repair can be done to, for example, change the R-proclet waiting time parameter t_{wR}.

Last but not least, the performance analysis problems in our **AQs** require the presence of the time information in event attributes. For example, the load on a particular machine can be predicted accurately (**AQ8**) only if the information about the load on incoming conveyors, i.e., the bag locations at various time moments, is known. So, event time information reconstruction is an important part of the log repair task. However, the state-of-the-art log repair approach [43] does not reconstruct timestamps. Since the PQR-system describes temporal parameters of queues and resources, the task of timestamp reconstruction can be potentially done for logs with events reconstructed without timestamps.

To summarize, both model interpretations are useful for answering the **AQs** for the same model and data. So, we conclude that the concept of generalized conformance checking is more suitable for our setting than the classical one. However, how to adapt generalized conformance checking [44] to our setting? All things considered, we do it for systems with shared resources and queues as follows.

- The input is a possibly incomplete event table with multiple case notions defined and PQR-system, as shown at the top of Figure 7.1(b) (instead of a single-case notion and complete event log and Petri net with black tokens in Figure 7.1(a)).
- Generalized conformance checking has two phases (two larger boxes in Figure 7.1(b)):
  - **Phase 1**. Model and log repair excluding timestamp reconstruction.
  - **Phase 2**. Reconstructing timestamps of missing (and repaired) events.
- During Phase 1, all deviations are detected and categorized as a data or model error (see the decision box in Figure 7.1(b)), and the corresponding task, i.e., log or model repair, is done accordingly.
- Domain knowledge is used to categorize deviations. We assume if a deviation cannot be categorized as a model error, it is categorized as a log error.
- During log repair (Phase 1), all missing events of the P-, Q- and R-proclets are reconstructed (arrow (3) in Figure 7.1(b)), still without timestamps but in the correct ordering with respect to the control flow (P-proclet's case notion pid).
- During model repair of Phase 1, the parameters of the Q- and R-proclets (case notions qid and rid) are repaired (arrow (4) in Figure 7.1(b)).
- Whenever model repair can be (at least partially) done *before* log repair, the repaired model should be used for log repair (arrow (5) in Figure 7.1(b)) for improving log repair accuracy.
- Note, *both* log repair (arrow 3) and model repair (arrow 4) are performed during generalized conformance checking, in contrast to classical conformance checking.
- When the missing events of the given event table are reconstructed (Event table 1 in Figure 7.1(b)), and the given PQR-system is repaired, Phase 2 starts.

- During Phase 2, the timestamp information of all the reconstructed events of Event table 1 is reconstructed *together* so it becomes (time) *complete correct*, i.e., it can be replayed over the repaired PQR-system according its replay semantics (Definition 6.14).

Note, we focus on the tasks of model and log repair because most deviations in our setting can be only caused by either model or log errors.

To address the problem of generalized conformance checking for systems with shared resources and queues, we formulate two RQs for Phases 1 and 2 respectively. The former reads as follows.

- **RQ-4**. *Given event data generated by a process with non-isolated cases in the form of an event table, and the PQR-system, how to relate the data and model to determine if the model correctly describes both the process for individual cases and case interaction observed in the event data, and if the event table fits into the behavior described by the PQR-system?*

To address **RQ-4**, we systematically explore various practical problems (use cases from the BHS domain), and whether and how existing techniques can solve them. For that, we build on [131], and ideas behind the PQR-system replay semantics. That is, we exploit the module structure of PQR-systems by decomposing the general problem into multiple simpler problems related to a particular proclet of process, queue, or resource, and applying existing techniques for these problems. We consider how deviations can be categorized as log and model errors using domain knowledge. Finally, we consider the synchronization of all the system proclets. To summarize, for each smaller problem of **RQ-4** we

1. provide a practical motivation,
2. explain how existing techniques can solve this problem,
3. check whether these techniques can solve the problem completely, and
4. identify what is still missing, i.e., how these existing techniques can be extended to fully solve the problem,
5. provide reasoning of how detected deviations can be categorized.

To address the problem of log repair with timestamp reconstruction (i.e., Phase 2), we formulate the following RQ:

- **RQ-5**. *Given event data generated by a system with shared resources and queues in the form of an event table where some events are* unobserved *(missing), and the PQR-system, how to reconstruct the unobserved events, including their timestamps, so that the resulting event table can be replayed over the given PQR-system?*

To solve this problem, we suggest an approach that takes an *incomplete* event table where unobserved events are missing, and a PQR-system as input, to reconstruct miss-

ing events with time information. This approach uses existing techniques for event reconstruction and our novel approach for reconstructing timestamp information.

As a result, we "assemble" our approach for generalized conformance checking as a combination of existing and novel techniques. Having said that, we admit that we do not provide complete implementation and evaluation for Phase 1. Historically, we approached **RQ-5** first as it was needed for addressing the other RQs of this thesis. As a result, we used solely alignments [43] for the P-proclet and not PQR-system-based conformance checking that we suggest in this chapter. We made this choice to provide implementation and evaluation of **RQ-5** timely within the project schedule. In the future, the Phase 1 approach can be used instead without changing the log repair approach.

In the remainder of this chapter, we formulate the conformance checking problem in Section 7.2.1. We consider the conformance checking problem, motivation, and approach for P-proclets, Q- and R-proclets, and proclets synchronization in Section 7.2.2, Section 7.2.3, and Section 7.2.4 respectively. Finally, we suggest the log repair method in Section 7.3.

7.2 Conformance Checking of PQR-Systems

In this section, we address **RQ-4**. For that, we formulate the problem over an event table and PQR-system, and use the PQR-system semantics to determine if a given event table can be replayed over the given PQR-system. As the semantics (Definitions 6.14) is defined as a composition of requirements for the proclets and their synchronization, we decompose the problem into simpler ones along this definition, motivate the practical use of each problem, and consider how they can be solved using existing techniques.

7.2.1 Problem Statement

As we mentioned above, the classical conformance checking settings assume as input [1]:

- an event log recorded for a single case notion,
- and a Petri net with black tokens.

During conformance checking, each trace of the event log is related to the model in isolation, i.e., as if no other cases exist. For that, a trace is *replayed* over the given model [43]. If it can be replayed, it means that either

- the model describes the piece of the behavior, observed through the trace events, accurately (descriptive models),
- the trace can be generated by the actual system/process (prescriptive models).

However, our setting is different. Instead of a single case notion event log, we have an event table (Definition 2.8) whose events are related to the process (case notion pid), queues (qid) and resources (rid). This event table can be seen as a set of three event logs derived for case notions pid, qid and rid (Definition 2.12), whose traces are not isolated but describes the interplay of the corresponding entities. Then, instead of a single Petri net we have the PQR-system, i.e., multiple CPNs whose tokens (corresponding to process, queue, and resource instance) are not replayed in isolation but synchronize.

In the following, we formulate the conformance checking problem for PQR-systems that aims to address the problem of multi-dimensional conformance checking for dimensions of the process, queues, and resources, as well the consistency of synchronization of all the traces in the execution together. Let us describe the input first.

Input. Let a system $\mathscr{S}$, which records its execution EX, i.e., the occurrences of its process step life-cycle transitions with labels Σ_{tP} (Definition 6.1) as events with attributes in an event table $ET = (CN_{\text{PQR}}, E, \#)$ (Definition 2.8) such that:

1. case notions pid and rid are global, exactly as point 1 in Definition 6.13 says,
2. in each non-empty trace $\sigma = \langle e_1, \ldots, e_n \rangle$ of an event log L^{ET}_{pid}, derived from event table ET for case notion pid according to Definition 2.12, all the events but the very first and last are related to a queue with some identifier $qid \in \mathscr{I}$, exactly as point 2 in Definition 6.13 says, i.e., $\forall 1 < i < n, \forall e_i \in \sigma, \#_{\text{qid}}(e_i) \neq \perp$.

Let model S be a PQR-system.

Given this input, the problem reads as follows.

Problem. Can the entire execution EX, i.e., all the events together, be replayed over the PQR-system S according to Definitions 6.14? If not, why it cannot be replayed, i.e., which violations of the replay semantic are detected while relating PQR-system S and event table ET?

This problem statement is explained in the diagram in Figure 7.2, where PQR-system S presumably models the actual system $\mathscr{S}$, which generated execution EX in the form of even table ET. By taking PQR-system S and event table ET as input, a conformance checking approach is to determine whether ET can be replayed over S, and if not, how ET violates the PQR-system replay semantics.

To approach this problem, we use the PQR-system replay semantics (see Definition 6.14), i.e., we check if an event table can be replayed over the given PQR-system. For that, we build on [133] and exploit the module structure of PQR-systems, decomposing the problem into several simpler problems for different proclet types, and for system-level synchronization. Thus, we address separately:

1. *behavioral conformance* [133], that we call *trajectory conformance* in the light of the PQR-system semantics,

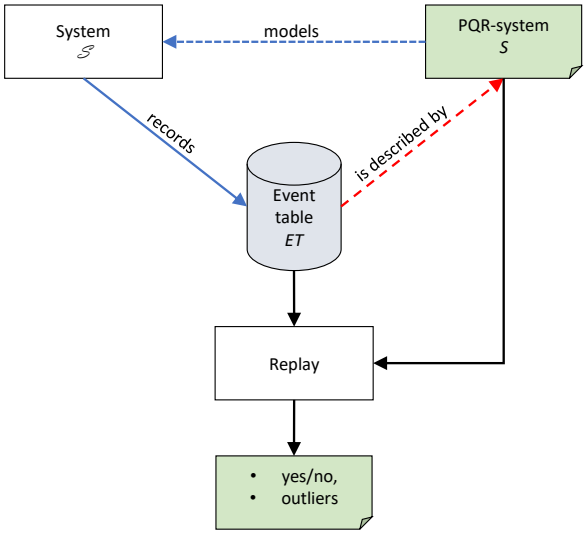


Figure 7.2: Conformance checking using PQR-systems.

2. and *interaction conformance* [133], that we call *synchronization conformance* to emphasize synchronization properties of the PQR-system.

For that, we consider clauses of Definition 6.14 top-down. Thus, it requires that

- all the clauses in Definition 6.13 hold, this is addressed in Section 7.2.2,
- and additionally defines conditions 1 and 2 in its clauses (2), this is addressed as synchronization conformance in Section 7.2.4.

In turn, clauses 1 and 2 of Definition 6.13 hold by the conditions in the problem statement input above, while its clauses 3-5 require that all the traces in event table *ET* have to be replayed over the P-, Q- and R-proclets according to Definition 6.12. This is addressed separately as

- the trajectory conformance problem for P-proclets in Section 7.2.2,
- the trajectory conformance problem for Q- and R-proclets in Section 7.2.3.

Next, we introduce conformance checking of P-proclets.

7.2.2 Trajectory Conformance for P-Proclets

This section formulates the sub-problem of conformance checking for P-proclets, provides the motivation for that as well as problem instances found in real-world systems, and suggests an approach for solving the sub-problem.

7.2.2.1 Problem

For the P-proclet, we check whether a trace of a classical event log can be replayed over the P-proclet net. The problem reads as follows.

Trajectory Conformance for P-Proclets. Let N_0 be the P-proclet of PQR-system S. Let L^{ET}_{pid} be an event log, derived from event table ET for case notion pid according to Definition 2.12. The trajectory conformance problem for P-proclets is the following: can each trace $\sigma \in L^{ET}_{\text{pid}}$ be replayed over net N_0 according to Definition 6.12?

7.2.2.2 Motivation

Before we discuss the approach of conformance checking for P-proclets, let us introduce practical reasons for doing that in the BHS domain. For that, we recall the running example of the simple BHS we discussed throughout the preceding chapters. In Figure 7.3(a), the sensors tracking passing bags are shown additionally. These sensors are installed according to the pattern we often observe in the real BHSs, i.e., they are installed:

- *at* the system entrances and exits (sensors $a1, a2$ and c),
- *before* merge units from the side of a conveyor with a *lower* priority (sensor b).

Note, there is no sensor before merge unit b on the conveyor coming from check-in counter $a2$ because this is the "main road", i.e., this conveyor has a higher priority and never yields bags coming from $a1$, so the bag tracking is not needed on this side for normal system operating. Moreover, we assume that the system records only events of *start* life-cycle transitions, as we observe in the real BHSs. As a result of this sensor configuration for the system in Figure 7.3(a), the traces of bags $pid1$ and $pid2$ (after leaving the system) look as follows:

- $\sigma_{pid2} = \langle a1_s, b1_s, c_s \rangle$,
- $\sigma_{pid1} = \langle a2_s, c_s \rangle$.

Note, events from sensor b for $pid1$ are missing because there was no sensor before merge unit b on its way. Given this system configuration, we explain three frequent use cases for the P-proclet conformance checking application.

Sensor Malfunctioning. A sensor is a photo-electronic device for bag detection, as explained in Section 4.3 in detail. It can fail to detect a bag because it is dirty, misplaced due to vibration, or broken. A sensor can either fail to detect *all* passing bags or fail to detect just *some* of them. In case of sensor b failure, the trace of $pid2$ from our example in Figure 7.3(a) is as follows:

- $\sigma^*_{pid2} = \langle a1_s, c_s \rangle$,

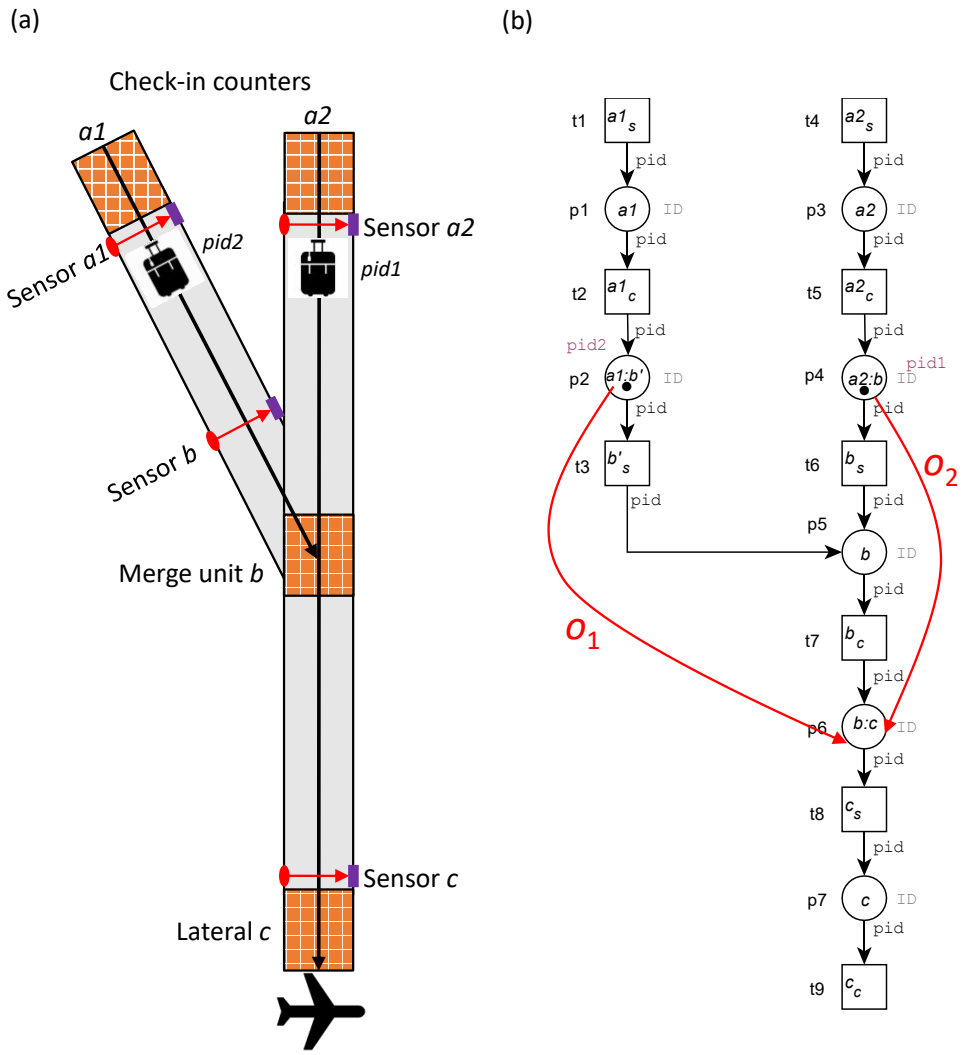


Figure 7.3: A BHS (a) and possible control flow outliers over the corresponding P-proclet (b).

i.e., it looks as if process step b was never executed for *pid*2. Conformance checking allows detecting trace σ^*_{pid2} as an outlier, which is shown as an arrow o_1 over the P-proclet net in Figure 7.3(b). Detection of this type of outlier can trigger the root-cause analysis in order to find and fix this failure in the system.

Manual Interventions. In BHSs, some emergency situations may require the engagement of workers in the baggage handling process. We call that *manual interventions*. For example, in case of merge unit b malfunctioning from the side of check-in counter

$a1$, bag $pid2$ can be *manually* brought to conveyor $b:c$ by a worker. In this case, the event(s) of activity b for $pid2$ are missing in the trace, exactly as in the case of sensor b failure discussed above (trace σ^*_{pid2}). Still, these two situations can be distinguished as follows.

- In the case of manual intervention, the time difference between the events before and after the skip is longer than in the case of sensor malfunctioning and has greater variety (in duration) since it is a manual operation.
- Manual intervention affects a smaller number of bags for a shorter period of time because any problems requiring manual interventions are typically solved urgently. In contrast, sensor malfunction usually lasts longer, until the problem is noticed and sensors are replaced.

So, using the information about the frequency and location of outliers during P-proclet conformance checking, possible root causes of outliers can be identified as well (in addition to outlier detection).

Incomplete Logging In the real world, recorded event logs are often incomplete due to various reasons [1]. In the BHS domain, one large and important source of this incompleteness is the architecture of bag tracking, as we saw in the previous examples. For the system in Figure 7.3(a), it results in losing all the events of *complete* transitions, and in losing the events of step b for all the bags coming from location $a2$. While the recorded trace of $pid1$ is

- $\sigma^*_{pid1} = \langle a2_s, c_s \rangle$,

the complete trace is

- $\sigma^c_{pid1} = \langle a2_s, a2_c, b_s, b_c, c_s, c_c \rangle$.

Event log incompleteness dramatically impedes the system performance analysis quality. Availability of the process model describing the control-flow perspective, i.e., the P-proclet, allows restoring unobserved (unrecorded) events in traces through conformance checking approaches that we consider next.

7.2.2.3 Approach

Now, we show how the problem of replaying a token over a CPN (Definition 6.12) in case of P-proclets can be reduced to the problem of conformance checking for Petri nets with black tokens, i.e., for cases executing in isolation, which has been extensively studied in process mining [24]. Concretely, we propose using an alignment-based approach [43].

In [43], a sound WF-net and event log are the input for conformance checking. In Section 6.3 we discussed that a P-proclet can be seen as a transition-bordered Petri net with black tokens, transformed into a CPN by adding color `ID` to tokens and

places for distinguishing tokens of different instances. As trajectory conformance is checked *per instance in isolation*, we do not have to distinguish case identifiers. So, we remove the colors of tokens and places and obtain a Petri net with black tokens. Next, we can convert the obtained net into a sound WF-net N_0^{WF}, according to clause 6 in Definition 6.2. As a result, we have WF-net N_0^{WF}. By applying alignment-based conformance checking [43] to each trace of L_{pid}^{ET}, we determine whether it can be replayed over WF-net N_0^{WF}. The outcome for each trace can be used for solving the problem instances discussed above.

7.2.3 Trajectory Conformance for Q- and R-Proclets

This section follows the same structure as the previous section, formulating the sub-problem, motivation, problem instances, and approaches for Q- and R-proclet conformance checking.

7.2.3.1 Problem

For Q- and R-proclets, we check whether the trace of an event log, derived for case notion qid or rid respectively from the given event table ET, can be replayed over the net of a Q- or R-proclet. This sub-problem reads as follows.

Trajectory Conformance for Q- and R-proclets. Let N_i be a Q- or R-proclet of the PQR-system S. Let L_{cn}^{ET} be an event log, derived from event table ET for case notion $cn = \text{qid}$ if N_i is a Q-proclet, or for case notion $cn = \text{rid}$ if N_i is an R-proclet, according to Definition 2.12. The trajectory conformance problem for Q- and R-proclets is the following: can each trace $\sigma \in L_{cn}^{ET}$ be replayed over net N_i?

7.2.3.2 Problem Instances

As we discussed in Section 6.5, the intuition behind a trace replaying over a CPN (Definition 6.12) assumes that the interpretation of a trace as a labeled transition occurrence sequence (Definition 6.9) provides a valid sequence of transition labels and bindings, while time distances between directly following occurrences (events) cannot be less than the corresponding minimum waiting or service time in the CPN. So, we are interested in problem instances for which detecting a wrong occurrence sequence and/or timing has a valuable practical use. In the following, we discuss three such problem instances.

More Accurate Trace Alignment of P-Proclet Traces Using Q-Proclets. Traces can be accurately aligned during conformance checking of the P-proclet when a single alignment variant is possible. However, it is possible to have multiple ones. Let us consider the loop with a shortcut in Figure 7.4, where the sensors track bags:

- before unit a,

Event identifier	Activity	Time	pid
$e1$	a_s	1	$pid1$
$e4$	b_c	12	$pid1$

Table 7.1: Bag $pid1$ incomplete trace.

Event	Activity	Time	pid
$e1$	a_s	1	$pid1$
$e2$	a_c	?	$pid1$
$e3$	b_s	?	$pid1$
$e4$	b_c	12	$pid1$

Table 7.2: Possible variant 1 of bag $pid1$ complete trace.

- after unit a on the path to the shortcut,
- after merge unit b.

The corresponding PQR-system is shown in Figure 7.5, where the observed transitions are drawn as filled rectangles. Note that in this figure, the R-proclets are not shown for simplicity. Let us now consider the trace of instance $pid1$ (case notion pid), shown in Table 7.1. In this trace, only events of the observed transitions are recorded. No identifies for case notions qid and rid are recorded, as it is usually the case in real systems. This trace can be alignment with the P-proclet in two different ways:

1. bag $pid1$ followed a longer path $a : b$, see the complete trace in Table 7.2, and
2. bag $pid1$ followed shortcut $a' : b'$, see the complete trace in Table 7.3.

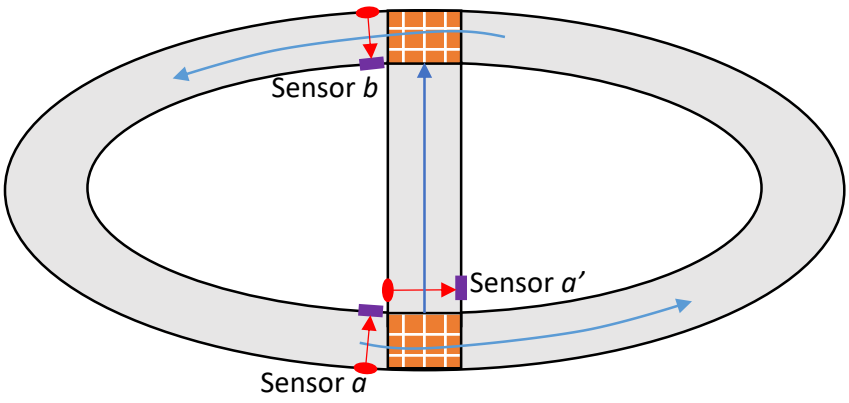


Figure 7.4: Sorting loop with a shortcut.

Event	Activity	Time	pid
$e1*$	a_s	1	$pid1$
$e2*$	a'_c	?	$pid1$
$e3*$	s_s	?	$pid1$
$e4*$	b_c	12	$pid1$

Table 7.3: Possible variant 2 of bag $pid1$ complete trace.

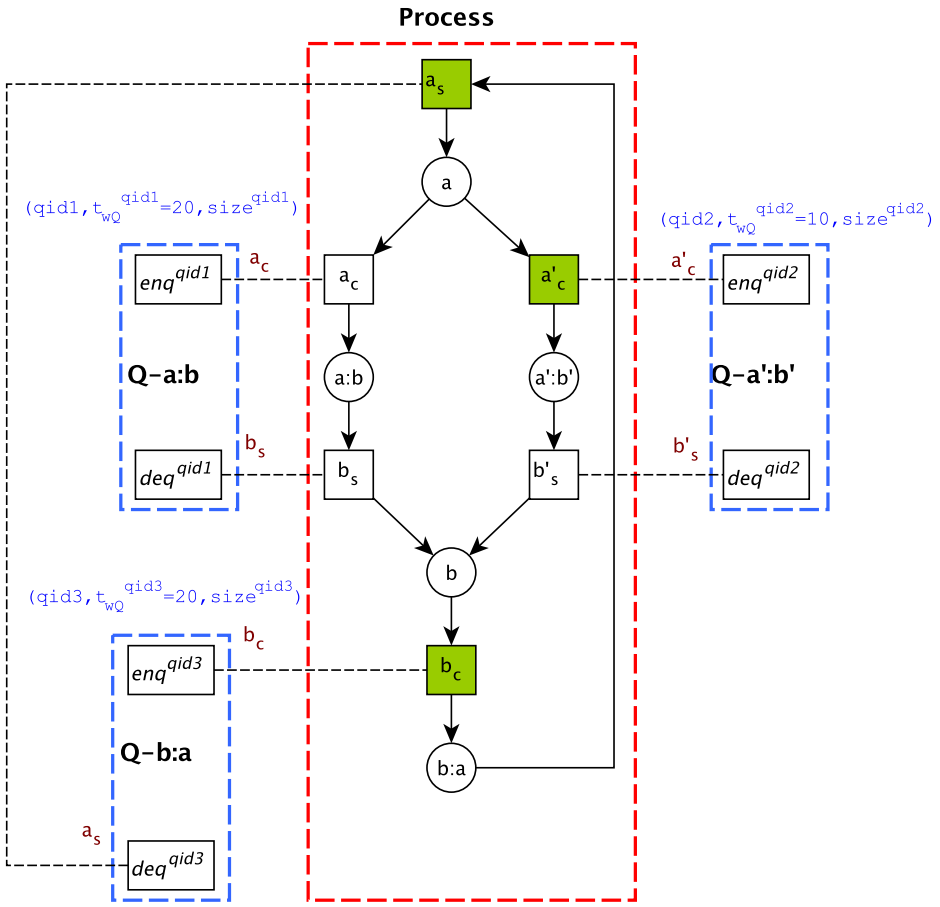


Figure 7.5: PQR-system of the sorting loop in Figure 7.4 (R-proclets are omitted). Only the transitions drawn as filled rectangles are recorded.

If only P-proclet is used for trace alignment, these variants are valid, and there is no reason to prefer one to another. However, conformance checking of the Q-proclets can help to choose the right one. To explain how it works, we consider both trace alignments (in Table 7.2 and Table 7.3).

1. First, we assume $e2$ and $e3$ are due to the a_c and b_s (Table 7.2). The observed time distance between events $e1$ and $e4$ $\#_{\text{time}}(e4) - \#_{\text{time}}(e1) = 11$. The time distance between events $e2$ and $e3$ cannot be greater than $\#_{\text{time}}(e4) - \#_{\text{time}}(e1)$ because $e2$ and $e3$ happened after $e1$ and before $e4$, i.e., $\#_{\text{time}}(e3) - \#_{\text{time}}(e2) \leq \#_{\text{time}}(e4) - \#_{\text{time}}(e1) = 11$. As we assume $e2$ and $e3$ are due to the a_c and b_s, then the bag traveled the conveyor $a : b$ and $e2$ and $e3$ are part of a trace of proclet *Q-a:b*, i.e., they are the part of a trace $\langle \ldots, e2, e3, \ldots \rangle$ for case notion qid and instance *qid*1. In this trace, events $e2$ and $e3$ are recorded for enqueuing and dequeuing of bag *pid*1, so the time distance between $e2$ and $e3$ cannot be less than $t_{\text{wQ}}^{qid1} = 20$, i.e., the condition $\#_{\text{time}}(e3) - \#_{\text{time}}(e2) \geq t_{\text{wQ}}^{qid1} = 20$ must hold for replaying this trace over the Q-proclet. As this is not the case, the trace $\langle \ldots, e2, e3, \ldots \rangle$ for case notion qid and instance *qid*1 cannot be replayed over proclet *Q-a:b*.
2. In contrast, if we assume $e2$ and $e3$ are due to the a'_c and b'_s (Table 7.3), condition $\#_{\text{time}}(e3) - \#_{\text{time}}(e2) \geq t_{\text{wQ}}^{qid2} = 10$ must hold. Since it is the case, this alignment is valid.

In practice, more accurate trace alignment results in more accurate analysis results on the aligned traces (event logs).

Concept Drift Detection for R-Proclets. The previous problem instance is about detecting problems on the data side when the model is assumed to be prescriptive. Now, we consider a problem instance caused by a descriptive model that becomes incorrect due to concept drift. Let us consider a conveyor where the protective space policy normally requires *two* protective space units between neighbor bags (see the right-hand part in Figure 7.6). While this policy provides plenty of room between bags for reliable work of the system equipment, it also "wastes" a lot of conveyor space. If the load becomes heavy, another policy can be invoked for better utilization of the conveyor space, for example, by keeping only *one* protective space unit between neighbor bags (see the left-hand part in Figure 7.6). As a result, the current template parameter t_{wR}, defining the minimum distance between neighboring bags, is no longer correct. Detection of systematic violations of this time characteristic can reveal this concept drift in the resource behavior, so the R-proclets can be tuned accordingly, i.e., the minimum waiting time of the corresponding R-proclets can be set to the correct (observed) value.

Imperfect Log Repair (Q- and R-proclets). So far, we considered problems on the data side caused by logging issues. Now, let us consider problems on the data side caused by the possible application of log repair techniques [50, 137, 138]. When

a log repair approach restores missing events, the timestamps of these events can be restored inaccurately because, for example, there is no sufficient information for accurate restoring available [50]. The resulting inaccuracy can cause wrong results of consequent performance analysis when unnoticed. These timestamp errors can be often detected through conformance checking of Q- and R-proclets, and taken into account by the analysts. Later, we additionally show how this type of conformance checking facilitates our conformance checking and log repair framework.

7.2.3.3 Approach

To address the problem of conformance checking for Q- and R-proclets, an approach capable to do conformance checking of CPNs would perfectly work. However, it is a rather complicated problem, and we could not find an existing approach in the literature. Fortunately, because the Q- and R-proclets use only a very restricted subset of the CPN syntax and semantics, this problem can be reduced to less general approaches. In this section, we focus on a work addressing compliance checking of temporal compliance requirements [53]. It leverages a data-aware conformance checking technique allowing conformance checking with respect to a given *data-aware Petri net* [52]. In the following, we explain how to apply this technique in our setting by the example of a simple CPN —R-proclet —first, and then discuss how this technique can be potentially extended for conformance checking of the more complex Q-proclet.

The approach [53] idea is as follows. A data-aware Petri net describes both the control flow perspective and temporal compliance requirements. The control flow is described as usual through the net of connected transitions and places [1], while the temporal conditions are defined through the *predicates* of *transition guards*. The R-proclet, converted into a data-aware Petri net, is shown in Figure 7.7. The minimum waiting and service time are defined through the guards on transitions $t2$ and $t3$ respectively.

While replaying a trace, the attribute values of each replaying event can be stored in the *variables* of the net, shown as triangles in Figure 7.7. The transition guards evaluate the predicates, composed from these net variables, and the variables are bound to the current event attribute values, e.g., for a currently replaying event e, a variable $e.time$ is bound to the attribute value $\#_{time}(e)$. If any predicate of the transition guards is evaluated as *false*, the replaying trace does not fit into the model.

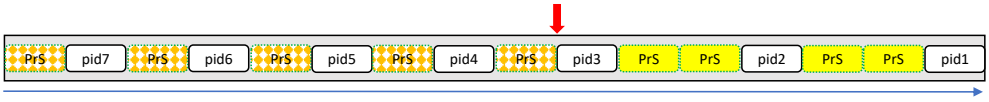


Figure 7.6: Concept drift due to the protective space policy change.

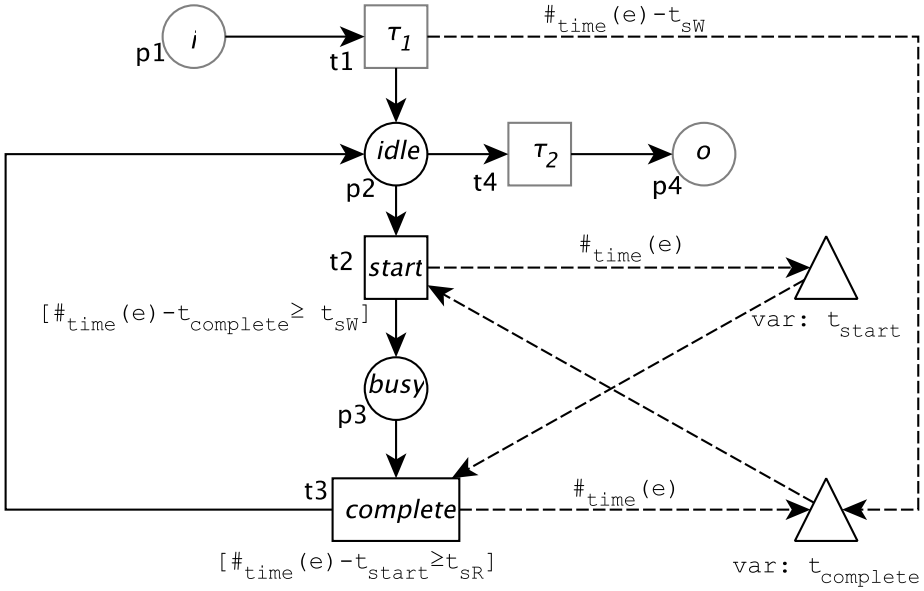


Figure 7.7: R-proclet converted into a data Petri net.

While further diagnostics can be obtained [53], these are not needed for solving our problem. Note, that other event attributes, besides *time*, can be used in a data-aware Petri net as well, e.g., a variable *e.pid* can be used for accessing the process instance identifier $\#_{\text{pid}}(e)$.

This approach can determine whether a trace can be replayed over an R-proclet, according to Definition 6.12, through:

1. conformance checking of the control-flow perspective, i.e., by checking if the interpretation of a trace as a labeled transition occurrence sequence has a correct sequence of labels (event activities) according to the R-proclet,
2. evaluation of transition guards, i.e., by checking if the time intervals between *start-complete* and *complete-start* pairs of occurrences are equal or greater than the minimum service and waiting time of the R-proclet respectively.

To obtain it, we first converted the initial model (Figure 6.9) into a WF-net and then added the variables and guards. For the conversion into a WF-net, the following steps have been done.

Event	Activity	Time	pid	qid	rid
$e3*$	b'_s	11	$pid2$	$qid1$	$rid3$
$e4*$	b_c	12	$pid2$	$qid3$	$rid3$
$e9*$	b_s	21	$pid1$	$qid2$	$rid3$
$e10*$	b_c	22	$pid1$	$qid3$	$rid3$

Table 7.4: Events and attributes of the trace σ_8 of Table 6.2.

1. We removed all the variables and color sets from the initial model because we do not have to distinguish different cases for replaying a single trace in isolation, as in [53].
2. We converted the resulting black token Petri net model into a WF-net by adding a source place i, a sink place o, and two transitions τ_1, τ_2 to connect them with place *idle*.

Afterward, we converted the resulting WF-net into a *data-aware Petri net*, capable to check the compliance to time characteristics of the R-proclet, in the following way. We defined two variables t_{start} and $t_{complete}$ to store the timestamps of events being replayed on transitions *start* and *complete* respectively. We also connect variables t_{start} and $t_{complete}$ with transitions *complete* and *start* respectively for reading their values. Finally, we provided guards on these transitions for checking compliance with the minimum waiting and service time.

To explain those modifications by example, let us assume $t_{\text{sR}} = 1$, $t_{\text{wR}} = 2$. Now, we replay trace $\sigma_8 = \langle e3*, e4*, e9*, e10* \rangle$ of Table 6.2, whose event attributes are shown in Table 7.4, as follows.

1. Initially, we have a model move [43] at transition $t1$. We use it to initialize variable $t_{complete}$ with the timestamp of the very first trace event $\#_{\text{time}}(e3*) = 11$, shifted to $t_{\text{wR}} = 2$ back at time, to make the very first check of the guard of transition $t2$ successful. The corresponding alignment is shown in Table 7.5, where model moves are written as $-$.
2. After the initialization, we continue by replaying $e3*$ on $t2$, whose firing stores the timestamp of $e3*$ in variable t_{start}. The guard of $t2$ holds as $11 - (11 - 2) \geq 2$.
3. Next, we replay $e4*$ on $t2$. Now, its guard expression uses the timestamp of the previous event $e3*$, stored in variable t_{start}. It also holds. Firing of $t2$ stores the timestamp of $e4*$ in variable $t_{complete}$.
4. Applying the same, we can show that trace σ_8 can be replayed on this net. The resulting alignment is presented in Table 7.5.

If at least one of the guards did not hold (e.g., if the timestamp of $e4*$ were 11.5), we would say the trace cannot be replayed. In our example, control-flow conformance is checked as in [43], which we do not discuss explicitly.

Event	Transition
–	$t1$
$e3*$	$t2$
$e4*$	$t3$
$e9*$	$t2$
$e10*$	$t3$
–	$t4$

Table 7.5: Alignment of events trace σ_8 (Table 7.4) with the transitions of the model in Figure 7.7.

So, we showed that temporal compliance checking [53] can be used for checking whether a trace can be replayed over an R-proclet. Now, let us consider the use of this technique for Q-proclet conformance checking. First, we converted the model of the Q-proclet (Figure 6.10) into a WF-net (Figure 7.8) in the same way as we have done for the R-proclet. Then, we have to provide the following constraints:

1. the minimum waiting time for queue elements, distinguished by the value of attribute pid, is greater or equal to t_{wQ}, and
2. the FIFO ordering is observed for queue elements.

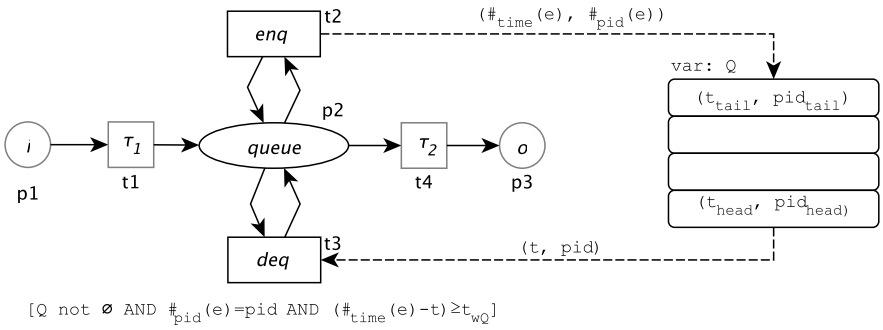


Figure 7.8: Q-proclet converted into an extended data Petri net.

Note, if we consider a trace of a Q-proclet for case notion qid, the directly following events of the transitions *enq* and *deq* do not necessarily relate to the same queue element (with the same value of pid). So, memorizing and using just the previous time of an *enq* or *deq* occurrence would not help in computing the actual waiting time for each queue element. Instead, some data structure is needed to keep the enqueuing time per element (*pid*) for checking on dequeuing of the same element. As

we have to verify the FIFO ordering of elements as well, we can use a list structure to kill two birds with one stone. However, data-aware Petri nets have no support for data structures. Assuming a conformance checking approach that can manage lists of values becomes available, the following procedure can be used for Q-proclet conformance checking.

To preserve the ordering, we define a variable Q of type *list of tuples*, where each tuple is in $\mathbb{T} \times \mathscr{I}$. On the replay of an event corresponding to enqueuing, the values of the event timestamp and attribute pid are appended to the list (as a tuple). On the replay of an event corresponding to dequeuing, the following conditions are checked:

1. the list Q is not empty, i.e., there is an element for dequeuing,
2. the identifier *pid* of the element to be dequeued (according to the ordering in Q) is equal to $\#_{\text{pid}}(e)$ being currently replayed over $t2$,
3. the time interval between enqueuing and dequeuing (for the element with this *pid*) is not less than t_{wQ}.

If these conditions, defined in the guard of transition $t3$, hold for an event being replayed over $t3$, event replay for the trace can be continued, and the tuple is removed from the list head. If all the events of a trace can be replayed, it means this trace can be replayed over the corresponding Q-proclet.

7.2.4 Synchronization Conformance Checking

So far, we discussed trajectory conformance for the P-, Q- and R-proclets, which considers replaying of traces over the corresponding proclet in isolation. However, this type of conformance checking does not address the problem of synchronization, which is the key process behavioral property described by the PQR-system. This section addresses this problem, following the structure of the two previous sections.

7.2.4.1 Problem

The precondition for the problem input is that process, queue, and resource traces have trajectory conformance to the P-, Q-, and R-proclets. The problem reads as follows.

Synchronization Conformance Checking. Let PQR-system S and event table ET be so that all traces of event logs derived for case notions pid, qid and rid (Definition 2.12) can be replayed over the P-, Q- and R-proclets respectively. The synchronization conformance checking problem is the following: is event table ET an execution of S, according to Definition 6.14?

Reducing The Problem. Before providing a problem instance example, let us revisit Definition 6.14. By the trajectory conformance checking of the P-, Q- and R-proclets we ensure clause (1) of Definition 6.14, i.e., that an event table can be replayed

Event	Activity	Time	pid	qid	rid
$e1$	c_s	1	$pid1$	$qid3$	$rid4$
$e2$	c_c	2	$pid2$	–	$rid4$
$e3$	c_s	6	$pid2$	$qid3$	$rid4$
$e4$	c_s	7	$pid1$	–	$rid4$

Table 7.6: Trace of resource $rid4$ of the system in Figure 7.3(a) (see Figure 6.6 for the whole PQR-system).

over the P-, Q- and R-proclets. For that, we can use the techniques of Sections 7.2.2 and 7.2.3. By the trajectory conformance checking of the Q-proclets we also ensure clause (2)2 of Definition 6.14, i.e., the FIFO ordering of queue elements is preserved with respect to the *pid* identifier. The ordering is checked by the guard on transition $t3$ that ensures that the element, being currently dequeued, is in the head of queue data structure Q (see Figure 7.8 of Section 7.2.3).

Finally, clause (2)1 of Definition 6.14, which requires that cases (for case notion pid) do not overtake each other within an R-proclet, is to be checked. For that, we assume that trajectory conformance of all the proclets has already been successfully checked. In the following, we consider the corresponding problem instance and the corresponding approach.

7.2.4.2 Motivation

Let us consider the trace of resource $rid4$ of the PQR-system in Figure 6.6, shown in Table 7.6. Assuming

- $t_{\text{sR}}^{rid4} = 1$, and
- $t_{\text{wR}}^{rid4} = 2$,

we can check that trajectory conformance holds for this trace, as the event order and timing are correct. Nevertheless, the *start* event $e1$ is related to bag $pid1$, while the directly following event $e2$ is related to bag $pid2$, i.e., this trace shows that $rid4$ started handling $pid1$ but then completed handling $pid2$, whose handling actually had not started yet. The same issue we see with events $e3$ and $e4$ of this trace. That is, the correlation constraint of the PQR-system is violated, and point (2)1 in Definition 6.14 does not hold. Detecting such problems allows for revealing issues in data, for example, due to incomplete logging or inaccurate log repair.

7.2.4.3 Approach

To check point (2)1 in Definition 6.14, it is sufficient (additionally to what is already checked for R-proclets in Section 7.2.3) to check whether each event e_i of a *complete*

transition $t3$ has the same value of attribute pid as the directly preceding event e_{i-1} of the *start* transition $t2$, i.e., if $\#_{\text{pid}}(e_{i-1}) = \#_{\text{pid}}(e_i)$. For that, we extend the model we used for temporal compliance checking of the R-proclet (Figure 7.7) as follows:

1. the value of attribute pid is stored for each *start* event in variable *pid* (Figure 7.9), and
2. condition $\#_{\text{pid}}(e_{i-1}) = \#_{\text{pid}}(e_i)$ is checked in the guard of the *complete* transition as $\#_{\text{pid}}(e) = \texttt{pid}$.

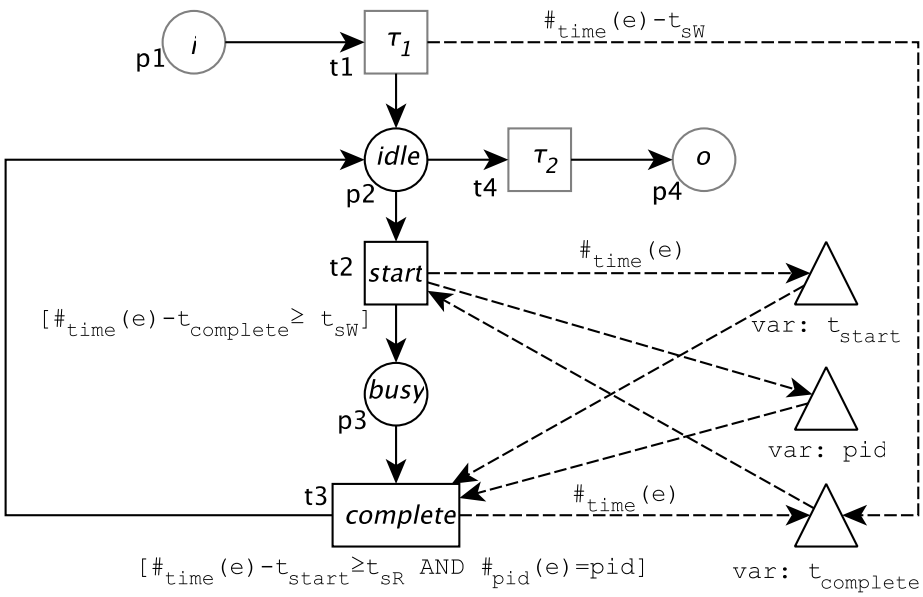


Figure 7.9: R-proclet as a data Petri net, extended for synchronization conformance checking.

Using the model of Figure 7.9 for trajectory conformance of R-proclet, we can check point (2)1 of Definition 6.14 together with trajectory conformance of R-proclets.

7.2.5 Limitations

In this section, we made an initial step toward conformance checking of PQR-systems. However, its approaches have the following limitations.

- Optimality of an alignment for the control-flow perspective of P-proclets and data-aware Petri nets is inherited from the corresponding approach [43] and expressed in the number of model and log moves. This definition of optimality does not consider the other perspectives and can lead to erroneous alignments, as shown in Section 7.2.3. However, incorporating other perspectives like resources causes an increase in the complexity of the problem and makes it difficult to apply to real-life problems [24].
- The data-aware Petri net, constructed for conformance checking of R-proclets and synchronization conformance checking in Section 7.2.4 uses single variables for saving timestamps and identifiers of events. It does not allow for supporting true concurrency because these variables cannot store this information for events with different identifiers generated concurrently.
- The approaches allow for limited diagnostics and aimed to only detection of outliers.

A full-scale *multi-perspective* conformance checking approach with the support of true concurrency and extended diagnostics is the subject of future work.

So far, we discussed conformance checking and model repair. Next, we consider our log repair approach.

7.3 Inferring Unobserved Events

In Section 7.1, we discussed how generalized conformance checking can be applied in our setting. Its conformance checking and model repair tasks are considered in Section 7.2. In this section, we propose our approach for the log repair task, using the PQR-system as a process model.

We provide motivation for log repair, and informal problem formulation, in Section 7.3.1. We consider reasons for information loss in BHSs in Section 7.3.2. We discuss partially ordered view on event tables, and their relation with the sequential view, in Sections 7.3.4 and 7.3.5. We formulate the problem formally in Section 7.3.6 and propose a visualization of performance spectra for event data with uncertainty in Section 7.3.7. We present our approach in Section 7.3.8, and its evaluation in Section 7.3.9.

7.3.1 Motivation

Precise knowledge about the actual process behavior and performance is required for identifying causes of performance issues [66], as well as for predictive process monitoring of important process performance indicators [32]. For MHSs, performance incidents are usually investigated offline, using recorded event data for finding root causes of problems [45], while online event streams are used as input for predictive

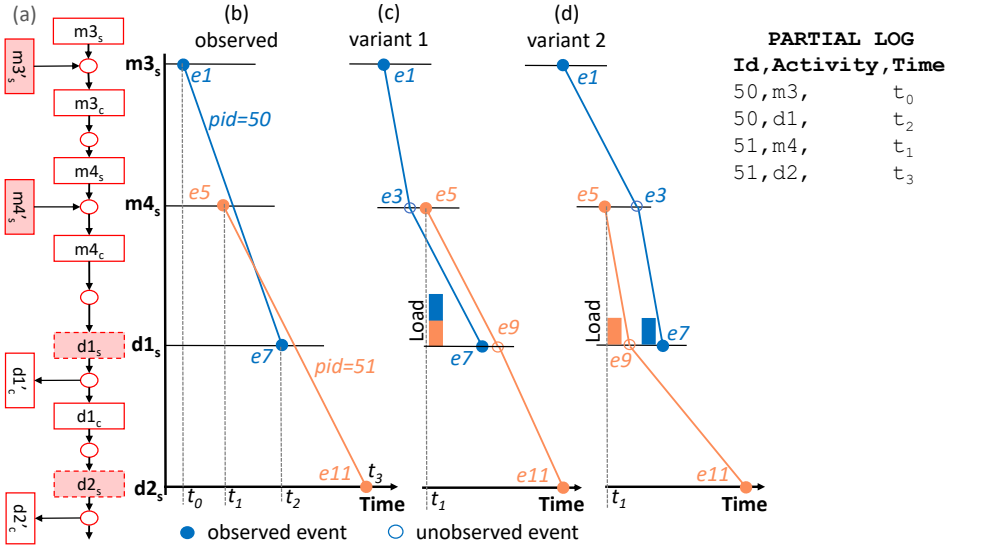


Figure 7.10: BHS model example (a), the observed imprecise behavior for two cases pid = 50 and pid = 51 (b), and two possible alternatives of the actual behavior (c) and (d).

performance models [160]. Both analysis and monitoring heavily rely on the completeness and accuracy of the input data. For example, events may not be recorded and, as a result, we do not know *when* they happened, even though we can derive that they must have happened. Yet, when different cases are competing for shared resources, it is important to reconstruct the ordering of events and provide bounds for non-observed timestamps.

However, in most real-life systems, items are not continuously tracked, and not all events are stored for cost-efficiency, leading to incomplete performance information, which impedes precise analysis. For example, a BHS tracks the location of a bag via hardware sensors placed throughout the system, generating tracking events for the system control, monitoring, analysis, and prediction. Historically, to reduce costs, a tracking sensor is only installed when it is strictly necessary for the correct execution of a particular operation, e.g., only for the precise positioning immediately before shifting a bag from one conveyor to another. Moreover, even if a sensor is installed, an event still can be *systematically* discarded for particular situations (e.g., if a bag was not diverted). As a result, the recorded event data of a BHS are typically incomplete, hampering the analysis based on these incomplete data. Therefore, it is essential to *repair* the event data before the analysis.

For example, Figure 7.10 shows a BHS fragment where not all events are always recorded. The process model is given, and for two cases the recorded incomplete sets of events are depicted, using the performance spectrum (see Chapter 3). In Figure 7.10(b), bag pid = 50 entering the system via $m3$ at time t_0 (event e_1) and leaving the system via $d1$ at time t_2 (e_7), and item pid = 51 entering the system via $m4$ at time t_1 (e_5) and leaving the system via $d2$ at time t_3 (e_{11}). As only these four events are recorded, the event data do not provide any information about in which order both cases traversed the *segment* $m4 \rightarrow d1$. Naively interpolating the movement of both items, as shown in Figure 7.10(b), suggests that item pid = 51 overtakes item pid = 50. This contradicts that all items are moved from $m4$ to $d1$ via a conveyor belt, i.e., a FIFO queue: item 51 cannot have overtaken item 50.

In contrast, Figure 7.10(c) and Figure 7.10(d) show two possible variants of the behavior, which are consistent with our knowledge of the system. We know that a conveyor belt (FIFO queue) is a shared resource between $m4$ and $d1$. Both variants differ in the order in which items 50 and 51 enter and leave the shared resource, the speed with which the resource operated, and the load and free capacity the resource had during this time.

In general, the longer the duration of naively interpolated segment occurrences, the larger the potential error. Errors in load, for example, make the performance outlier analysis [45] or short-term performance prediction [47] rather difficult. Errors in the order impede the root-cause analysis of performance outliers, e.g., determining the cases that caused or were affected by the outlier behavior.

Problem. In this section, we address a novel type of problem as illustrated in Figure 7.10 and explained above. The behavior and performance of the system cannot be determined by the properties of each case in isolation but depends on the *behavior of other cases* and the *behavior of the shared resources* involved in the cases. Crucially, each case is handled by multiple resources, and each resource handles multiple cases, resulting in *many-to-many* relations between them.

The concrete problem we address is to reconstruct the *unobserved* behavior and performance information of *each case* and each *shared resource* in the system that is *consistent* with both observed and reconstructed unobserved behavior and performance of all other cases and shared resources.

Input. More specifically, we consider the following information as given:

1. a PQR-system S (see Definition 6.7), i.e., a process model that describes possible paths for handling each individual case, resources and queues involved in each step, and their parameters, such as the ordering, minimum service and waiting times,
2. a monotone event table $ET_1 = (E_1, \{\text{pid}\}, \#)$ (Definition 2.10) such that:
   (a) case notion pid is global, i.e., $\forall e \in E_1, \#_{\text{pid}}(e) \neq \bot$,

(b) some activities of the P-proclet of S are unobservable,
(c) no events are explicitly related to instances of the Q- and R-proclets, i.e., identifiers for the case notions qid and rid are not recorded: $\forall e \in E_1, \#_{\text{qid}}(e) = \#_{\text{rid}}(e) = \perp$.

Problem. Given the input above, we want to provide an event table $ET_2 = (E_2, CN_{\text{PQR}}, \#^3)$ that describes:

1. for each instance of the P-proclet, i.e., for a case of case notion pid, the exact sequence of process steps,
2. for each unobserved event $e \in E_{\text{u}} = E_2 \setminus E_1$, a time-window of the earliest and latest occurrence of the event so that either all earliest or all latest timestamps altogether describe a consistent execution of the entire process over all shared resources and queues, described in S,
3. for each event, related to a Q-proclet and/or R-proclet, the corresponding case identifier for case notions qid and rid respectively.

The resulting event table ET_2 can be converted into a complete event table (Definition 2.9) ET_3 that is also correct, i.e., it can be replayed over S (see Definition 6.14) by choosing exact timestamp values from the defined timestamp intervals for events in E_{u}.

7.3.2 Information Loss

In this section, we introduce a running example that has a typical logging architecture of the BHSs we analyzed. We show how and why information is lost, resulting in incomplete event logs (tables), by the example of a system run.

Running Example. In Section 6.7, we introduced a BHS simulation model that we use throughout Chapters 7-9 for providing examples and evaluation on synthetic data. Although it is much simple than any real BHS, it is still too large to explain the ideas behind our approach for inferring unobserved events concisely. In this section, we use its simplified modification.

Figure 7.11(a) shows the BHS. It has four parallel check-in counters $c1 - c4$ that merge into one "main" linear conveyor through merge units $m2 - m4$. Incoming conveyors before $c2 - c4$ are accumulating (see Chapter 4), i.e., they have several short independent belts at the end. They are used for accumulating bags when the main conveyor is unavailable so that their other belts do not need to stop in this case. Diverting units $d1$ and $d2$ can divert bags from the main conveyor toward scanners $s1$ and $s2$.

Figure 7.11(b) shows an MFD of the system in Figure 7.11(a). The corresponding PQR-system (Definition 6.7) is shown in Figure 7.12.

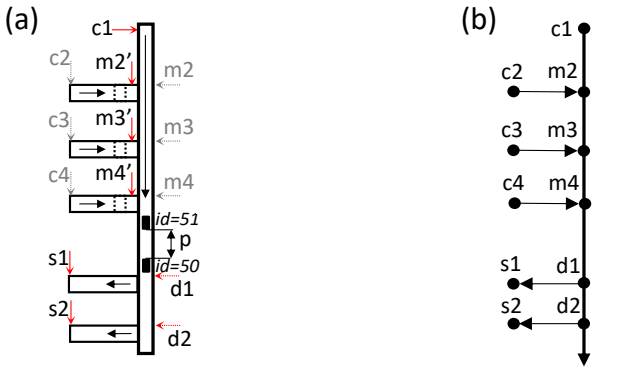


Figure 7.11: In a BHS fragment (a), where the red arrows correspond to the installed sensors, and grey ones correspond to not installed, and the corresponding material flow diagram (b).

For this system, we assume the event logging architecture that we often observed in real BHSs. That is, the sensors tracking bag movement are installed only if it is required for internal system needs, and not for logging all process steps. The installed and "missing" sensors are shown in Figure 7.11(a) by red and grey arrows respectively. The dashed red arrows show sensors whose events are *not* always recorded (only if a bag is diverted toward $s1, s2$). The installed (red) sensors correspond to the P-proclet transitions shown as filled boxes in Figure 7.12. To summarize, a process step is recorded when

- a bag is merged onto another conveyor (sensors $m2' - m4'$ in Figure 7.11(a)),
- a bag is diverted toward scanners $s1, s2$ (sensors $d1, d2$) (if a bag is not diverted, events of $d1, d2$ are unobserved), and
- a bag reaches a scanner location (sensors $s1, s2$).

Events of process steps executed in the remaining situations are unobserved (not recorded). As Figure 7.12 shows, most transitions of the P-proclet in Figure 7.12 generate unobserved events. Next, we show by example what are the consequences of the logging architecture for recorded event logs.

System Run and Event Log. Now, we consider how the BHS handles two bags, what system run it generates, and what information is missing in the recorded event table. To demonstrate the latter, we proved two resulting event tables:

- an event table ET, shown in Table 7.7, contains both observed and unobserved events, and identifiers of case notions in CN_{PQR},

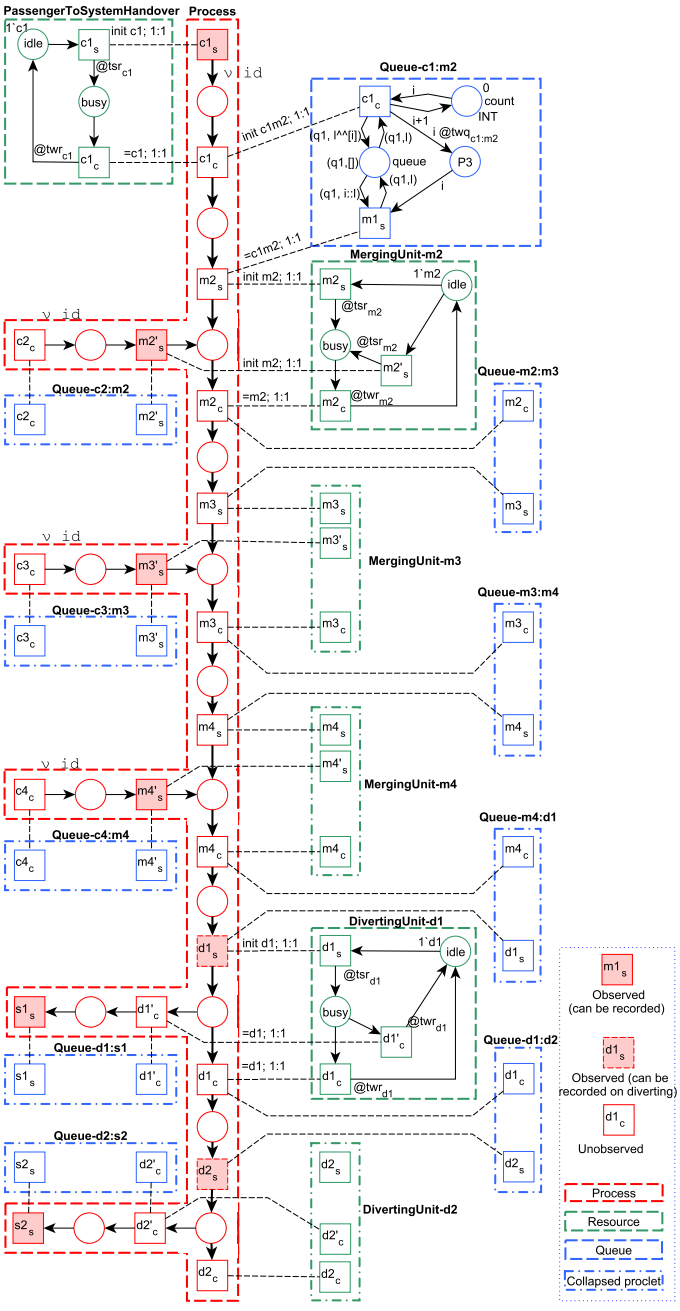


Figure 7.12: PQR-system of the BHS in Figure 7.11.

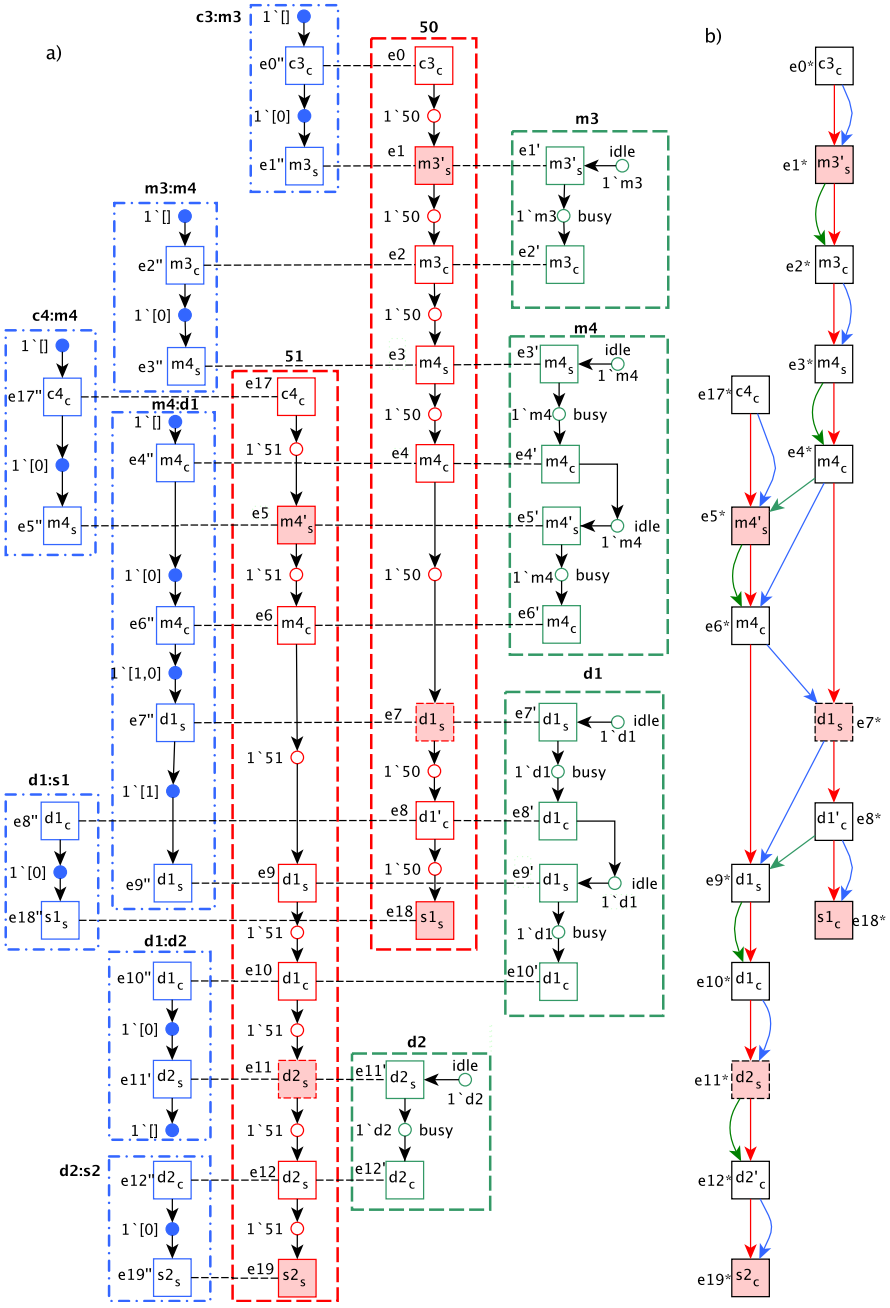


Figure 7.13: System run (a), and its labeled partial order (b).

event ID	pid	act	time	is recorded	qid	rid
$e0$	50	$c3_c$	t_0 (9:00:15)	no	$c3:m3$	$\perp$
$e1$	50	$m3'_s$	t_1 (9:00:30)	yes	$c3:m3$	$m3$
$e2$	50	$m3_c$	t_2 (9:00:40)	no	$m3:m4$	$m3$
$e3$	50	$m4_s$	t_3 (9:00:45)	no	$m3:m4$	$m4$
$e4$	50	$m4_c$	t_4 (9:00:50)	no	$m4:d1$	$m4$
$e7$	50	$d1_s$	t_7 (9:01:05)	yes	$m4:d1$	$d1$
$e8$	50	$d1'_c$	t_8 (9:01:10)	no	$d1:s1$	$d1$
$e18$	50	$s1_s$	t_{18} (9:01:15)	yes	$d1:s1$	$\perp$
$e17$	51	$c4_c$	t_{17} (9:00:35)	no	$c4:m4$	$\perp$
$e5$	51	$m4'_s$	t_5 (9:00:55)	yes	$c4:m4$	$m4$
$e6$	51	$m4_c$	t_6 (9:01:00)	no	$m4:d1$	$m4$
$e9$	51	$d1_s$	t_9 (9:01:15)	no	$m4:d1$	$d1$
$e10$	51	$d1_c$	t_{10} (9:01:20)	no	$d1:d2$	$d1$
$e11$	51	$d2_s$	t_{11} (9:01:25)	yes	$d1:d2$	$d2$
$e12$	51	$d2'_c$	t_{12} (9:01:30)	no	$d2:s2$	$d2$
$e19$	51	$s2_s$	t_{19} (9:01:35)	yes	$d2:s2$	$\perp$

Table 7.7: Complete event table.

event ID	pid	act	time
$e1$	50	$m3'_s$	t_1
$e7$	50	$d1_s$	t_7
$e18$	50	$s1_s$	t_{18}
$e5$	51	$m4'_s$	t_5
$e11$	51	$d2_s$	t_{11}
$e19$	51	$s2_s$	t_{19}

Table 7.8: Recorded incomplete event table.

- an event table ET', shown in Table 7.8, contains only observed events, whose identifiers for case notions qid and rid are not recorded, as it usually is for real BHSs.

Bag handling happens as follows.

1. First, bag 50 is inserted in the system via input transition $c3_c$ at time t_1. As a result, event $e0^*$ is generated (see Figure 7.13(b)). This event is a *synchronization* of $e0$ and $e0''$ (see the system run in Figure 7.13(a)) when $c3_c$ occurs for bag 50 in the P-proclet queue $c3:m3$ respectively (see Figure 7.13(a)). The attribute values of $e0*$ are shown in ET in Table 7.7. However, $e0$ is not recorded to ET' (see Table 7.8), according to the logging architecture.
2. At least the minimum waiting time $t_{\text{WQ}}^{c3:m3}$ must pass before bag 50 reaches the end of $c3:m3$, and next process step $m3$ can be started.

3. In process step $m3$ ($e1$), bag 50 is merged onto the main conveyor. For that, bag 50 leaves $c3:m3$ ($e1''$), and resource $m3$ starts its merge ($e1'$). These tree events result in synchronized event $e1^*$ in Figure 7.13(b). Event $e1$ is recorded to ET', however resource identifier $m3$ and queue identifier $c3:m3$ are not recorded, as in real BHSs.
4. When merge starts, resource *m3* switches from *idle* to *busy*. At least minimum service time t_{sR}^{m3} must pass before it completes merge (process step $m3_c$, event $e2^*$). Complete process steps are never recorded to ET'.
5. Concurrently, bag 51 is inserted via $c4_c$ ($e17^*$), moves via queue $c4:m4$ toward merge unit *m4* to enter queue $m4:d1$.
6. Both bags start competing for merge unit *m4*.
7. *m4* executes $m4_s$ and $m4_c$ for bag 51 ($e5^*$ and $e6^*$) *after* they are completed for bag 50 ($e3^*$ and $e4^*$). Thus, 51 enters the queue ($e6^*$) after bag 50 ($e5^*$) but before bag 50 leaves it ($e7^*$).
8. As a result, unit *d1* first serves (and diverts) bag 50 ($e7^*$ and $e8^*$) toward *s1* ($e18^*$), and serving bag 51 afterward ($e9^*$ and $e10^*$).
9. Finally, bag 51 reaches scanner *s1* ($e19^*$).

On completion, only six out of 16 generated events in ET are recorded to ET' (compare Table 7.7 and Table 7.8).

In this section, we showed by example how logging implemented in real-world BHSs makes a significant part of event information unobserved in recorded event data. In the next sections, we consider how event tables can be considered via the views of different types, which we exploit for inferring unobserved events later in this chapter.

7.3.3 Sequential View on Event Tables

view!sequential

Event tables that we considered so far lack explicit structure, required for reasoning about the behavior they capture. In this and the next section, we introduce two views on them that are capable to express the captured behavior:

1. as a set of sequential logs,
2. as partial order.

First, we define a sequential event log as follows.

Definition 7.1 (Sequential event log). *$L = (cn, E, \#)$ be an event log with case notion attribute cn (see Definition 2.11) and the set of cases $cn(L)$ (see Definition 2.13). If multiple events in E with the same case id in $cn(L)$ have identical timestamps, i.e., $\exists e, e' \in E, \#_{cn}(e) = \#_{cn}(e')$ and $\#_{\text{time}}(e) = \#_{\text{time}}(e')$, case id has multiple traces (see Definition 2.13).*

We write $\sigma(L, cn = id)$ for the set of traces of case id. A sequential event log *of L that is a set $\sigma(L, cn)$ which contains for each $id \in cn(L)$ exactly one trace $\sigma \in \sigma(L, cn = id)$.*

If an event log L is incomplete (see Definition 2.11), the notion of trace and sequential event log are non-deterministic because events without timestamps can be placed at arbitrary positions of traces, thereby allowing multiple traces for a case. However, for a time-monotone event log, each case *id* has a *unique* trace $\{\sigma_{cn}^{id}\} = \sigma(L, cn = id)$ and the log $\sigma(L)$ is is uniquely defined. We then write $\sigma_{cn}^{id} = \sigma(L, cn = id)$.

The recorded event table in Table 7.8 is a complete, time-monotone event log for case notion (entity type) pid defining cases $\{50, 51\}$ and traces

- $\sigma(L, \text{pid}, 50) = \sigma_{\text{pid}}^{50} = \langle e1, e7, e18\rangle$, and
- $\sigma(L, \text{pid}, 51) = \sigma_{\text{pid}}^{51} = \langle e5, e11, e19\rangle$.

This is a "classical" sequential interpretation of observed events in the complete event table shown in Table 7.7, it describes how bags 50 and 51 travel from check-in counters $c3, c4$ to scanners $s1, s2$.

However, what is a sequential interpretation of an event table with *multiple* case notions, such as the event table shown in Table 7.7? To represent it, we define a *sequential view* on an event table as follows (note that the functions $cn(L)$, $corr(L, cn, id)$, $\sigma(L, cn, id)$, $\sigma(L, cn)$ of Definition 2.13 are well-defined over event tables).

Definition 7.2 (Sequential view on event table)**.** *Let $ET = (CN, E, \#_n)$ be an event table (Definition 2.8). A sequential view on ET is a family $\langle \sigma(ET, cn)\rangle_{cn \in CN}$ of sequential event logs (one per each entity type in CN).*

Similarly to sequential logs, if ET is time-monotone (see Definition 2.10), each sequential log $\sigma(ET, cn)$ is unique, and the sequential view on ET is unique.

Let us provide an example of a sequential view on event table ET shown in Table 7.7. It is time-monotone, as can be seen from the absolute values of the timestamps, so such a view is unique.

ET has three case notions pid, qid and rid, so it has three sequential logs $\sigma(ET, \text{pid})$, $\sigma(ET, \text{qid})$ and $\sigma(ET, \text{rid})$.

The traces of $\sigma(ET, \text{pid})$ are

- $\sigma(ET, \text{pid}, 50) = \langle e0, e1, e2, e3, e4, e7, e8, e18\rangle$, and
- $\sigma(ET, \text{pid}, 51) = \langle e17, e5, e6, e9, e10, e11, e12, e19\rangle$.

Similarly to the example above, they describe how bags 50,51 travel in the system.

The traces of $\sigma(ET, \text{rid})$ are

- $\sigma(ET, \text{rid}, m3) = \langle e1, e2\rangle$,
- $\sigma(ET, \text{rid}, m4) = \langle e3, e4, e5, e6\rangle$,
- $\sigma(ET, \text{rid}, d1) = \langle e7, e8, e9, e10\rangle$, and
- $\sigma(ET, \text{rid}, d2) = \langle e11, e12\rangle$.

These traces describe the order in which each machine, corresponding to a case (case notion rid), handled bags 50 and 51, thus

- $e1, e2, e3, e4, e7, e8 \in corr(ET, \text{pid}, 50)$, and
- $e5, e6, e9, e10, e11, e12 \in corr(ET, \text{pid}, 51)$.

That is, some traces $\sigma(ET, \text{rid}, m4)$ and $\sigma(ET, \text{rid}, d1)$ contain events related to different bags. It can be seen as these traces go "accross" bag traces $\sigma(ET, \text{pid}, 50)$ and $\sigma(ET, \text{pid}, 51)$.

Finally, the traces of $\sigma(ET, \text{qid})$ are

- $\sigma(ET, \text{qid}, c3 : m3) = \langle e0, e1 \rangle$,
- $\sigma(ET, \text{qid}, m3 : m4) = \langle e2, e3 \rangle$,
- $\sigma(ET, \text{qid}, m4 : d1) = \langle e4, e6, e7, e9 \rangle$,
- $\sigma(ET, \text{qid}, d1 : s1) = \langle e8, e18 \rangle$,
- $\sigma(ET, \text{qid}, c4 : m4) = \langle e17, e5 \rangle$,
- $\sigma(ET, \text{qid}, d1 : d2) = \langle e10, e11 \rangle$, and
- $\sigma(ET, \text{qid}, d2 : s2) = \langle e12, e19 \rangle$.

These traces describe the order in which bags (cases of pid) 50 and 51 entered and left each queue. In $\sigma(ET, \text{qid}, m4 : d1)$, $e4, e7 \in corr(ET, pid, 50)$, and $e6, e9 \in corr(ET, pid, 51)$.

In a sequential view, the same event can be a part of different event logs, as we show above. The rid and qid traces describe how pid traces synchronize on shared resources (rid) and queues (qid). However, the sequential view does not show it explicitly. Next, we propose a partial-order view on event tables.

7.3.4 Partially Ordered View on Event Tables

view!partially ordered

In this section, we encode the timestamp-based order of events of the same traces not within traces of multiple sequential event logs, but as a partial order, that is a Strict Partial Order (SPO) due to the monotonicity of the $\#_{\text{time}}(.)$ values. We start by ordering events $e_1 < e_2$ only if they are related to the *same entity*, and then extend this ordering across multiple different entities through the transitive closure. In the remainder of this chapter, we refer to case notions in CN as *entity types* because we consider event tables related to PQR-systems.

The partial order view definition reads as follows.

Definition 7.3 (Partial-order view, system-level run)**.** *Let $ET = (CN, E, \#)$ (Definition 2.10) be a monotone event table. Let $cn \in CN$ and $id \in cn(ET)$. Event $e_1 \in E$* precedes *event $e_2 \in E$ in entity id of type cn, written $e_1 <^{id}_{cn} e_2$ iff*

1. *$\bot \neq \#_{\text{time}}(e_1) < \#_{\text{time}}(e_2) \neq \bot$, i.e., the timestamp of e_1 is before the timestamp of e_2, and*

2. $\#_{cn}(e_1) = \#_{cn}(e_2) = id$, *i.e., both events are related to the same entity id.*

e_1 directly precedes e_2 *in entity id of type cn, written* $e_1 \lessdot^{id}_{cn} e_2$, *iff there exists no* $e' \in E$ *with* $e_1 <^{id}_{cn} e' <^{id}_{cn} e_2$. *This ordering lifts to entity types and entire ET:*

- e_1 directly precedes e_2 *in entity type cn, written* $e_1 \lessdot_{cn} e_2$, *iff* $e_1 <^{id}_{cn} e_2$ *for some* $id \in cn(ET)$, *and*
- e_1 directly precedes e_2, *written* $e_1 \lessdot e_2$, *iff* $e_1 \lessdot_{cn} e_2$ *for some* $cn \in CN$.
- *The transitive closures* $(\lessdot_{cn})^+ =<_{cn}$ *and* $(\lessdot)^+ =<$ *define* (indirectly) precedes *per entity type and for all events in ET, respectively.*

The partial-order view on ET or system-level run *of ET (induced by* $\#_{\text{time}}$*) is* $\pi = (E, <, CN, \#)$.

Note, if an event table is incomplete, events with undefined timestamps are unordered to all other events.

Figure 7.13(b) visualizes the directly precedes relations $\lessdot_{\text{pid}}$, $\lessdot_{rid}$, $\lessdot_{qid}$ induced by $\#_{\text{time}}$ for the event table shown in Figure 7.11(c). It shows that cases pid = 50 and pid = 51 are independent under the control-flow perspective $<_{\text{pid}}$ (e.g., $e4 \not<_{\text{pid}} e5 \not<_{\text{pid}} e7$), but mutually depend on each other under $<_{rid}$ and $<_{qid}$ (e.g., $e4 \lessdot_{rid} e5 \lessdot_{rid} e6$, and $e6 \lessdot_{qid} e7$).

Let us show that $<$ is an SPO.

Lemma 7.4. *Let* $ET = (CN, E, \#)$ *(Definition 2.10) be a monotone event table. Let* $\pi = (E, <, CN, \#)$ *be a system-level run of ET. Then* $(E, <)$ *is an SPO.*

Proof. We have to show that $<$ is transitive and irreflexive. $<= (\lessdot)^+$ is transitive by construction in Definition 7.3. Regarding irreflexivity: $e_1 \lessdot e_2$ holds only if $\#_{\text{time}}(e_1) < \#_{\text{time}}(e_2)$. As ET is monotone, either $\#_{\text{time}}(e_1) < \#_{\text{time}}(e_2)$ or $\#_{\text{time}}(e_2) < \#_{\text{time}}(e_1)$ holds (Definition 2.10) but not both, hence $<$ is irreflexive. □

7.3.5 Relation between Sequential and Partially-Ordered View

Now, we establish a more explicit relation between the traces in the sequential view on ET, and the system-level run π. Later, we use it for defining and solving the problem of this section.

Given a system-level run $\pi = (E, <, CN, \#)$, we write π_{cn} for the projection of π onto entity type $cn \in CN$ there $\pi_{cn} = (E_{cn}, <_{cn}, \{cn\}, \#)$ contains only events $E\big|_{cn} = \{e \in E, | \#_{cn}(e) \neq \perp\}$ related to cn. Relation $<_{cn}$ is already well-defined with respect to $E\big|_{cn}$. We call π_{cn} the entity-type level run of ET for entity type cn.

Correspondingly, given an identifier $id \in cn(ET)$, the projection $\pi^{id}_{cn} = (E^{id}_{cn}, <^{id}_{cn}, \{cn\}, \#)$ contains only the events $E\big|^{id}_{cn} = corr(ET, cn = id)$ of id. We call π^{id}_{cn} the entity-level run of ET of entity id of type cn.

For example, from the system-level run in Figure 7.13(b), we can obtain the entity-level runs

- π^{50}_{pid} and π^{51}_{pid} from the process perspective,
- $\pi^{m3}_{rid}, \pi^{m4}_{rid}, \pi^{d1}_{rid}, \pi^{d2}_{rid}$ from the resource perspective, and
- $\pi^{c3:m3}_{qid}, \pi^{m3:m4}_{qid}, \pi^{c4:m4}_{qid}, \pi^{m4:d1}_{qid}, \pi^{d1:d2}_{qid}, \pi^{d1:s1}_{qid}, \pi^{d2:s2}_{qid}$ from the queue perspective.

Each entity-level run π^{id}_{cn} corresponds to a sequential trace σ^{id}_{cn} in the sequential view of ET because either view derives the direct precedence/succession of events from the same principles.

Lemma 7.5. *Let $ET = (CN, E, \#)$ (Definition 2.10) be a monotone event table. Let $\pi = (E, <, CN, \#)$ be a system-level run of ET.*

For all $e_1, e_2 \in E$ holds: $e_1 \lessdot e_2$ iff there exists $cn \in CN$ and $id \in cn(ET)$ so that $\langle \ldots, e_1, e_2, \ldots \rangle = \sigma(L, cn = id)$ is a trace in the sequential view $\langle \sigma(ET, cn) \rangle_{cn \in CN}$ of ET.

Proof. If $e_1 \lessdot e_2$ then by Definition 7.3, $e_1 \lessdot^{id}_{cn} e_2$ for some $cn \in CN$ and $id \in cn(ET)$. Thus, $\#_{cn}(e_1) = \#_{cn}(e_2)$ and $\#_{\text{time}}(e_1) < \#_{\text{time}}(e_2)$ (by Definition 7.3 and ET being monotone). By Definition 2.13, $e_1, e_2 \in corr(ET, cn = id)$ (correlated into the same case id for cn). Further, because there is no $e' \in corr(CN, cn = id)$ with $\#_{\text{time}}(e_1) < \#_{\text{time}}(e') < \#_{\text{time}}(e_2)$ (definition of $\lessdot$ in Definition 7.3), e_1 and e_2 are ordered next to each other in the sequential trace $\langle \ldots, e_1, e_2, \ldots \rangle = \sigma(ET, cn = id)$. The converse holds by the same arguments. □

Corollary 7.5.1. Let $ET = (CN, E, \#)$ (Definition 2.10) be a monotone event table. Let $\pi = (E, <, CN, \#)$ be the system-level run of ET. For any π^{id}_{cn} for $cn \in CN$, $id \in cn(ET)$ holds $e_1 \lessdot^{id}_{cn} e_2$ iff $\langle \ldots, e_1, e_2, \ldots \rangle = \sigma(ET, cn = id)$ and $e_i \lessdot^{id}_{cn} e_j$ iff $\langle \ldots, e_i, \ldots, e_j, \ldots \rangle = \sigma(ET, cn = id)$.

The above relation we use in the next sections from being able to change perspectives at will and study (and operate on) behavior as a classical sequence (and use sequence reasoning for a single entity) as well as a partial order (and use partial order reasoning across different entities). For instance, the directly precedes relations $\lessdot_{\text{pid}}, \lessdot_{\text{qid}}, \lessdot_{\text{rid}}$, visualized in Figure 7.13(b), directly define the sequences of events we find in $\sigma(ET, \text{pid}), \sigma(ET, \text{qid})$, and $\sigma(ET, \text{rid})$.

7.3.6 Problem Statement

In this section, we formulate the problem addressing **RQ-5**. For that, we first define what is a *correct and complete* event table with respect to a given PQR-system S.

Definition 7.6 (Correct and complete event table)**.** *Let S be a PQR-system (see Definition 6.7) that defines proclets for a process, queues, and resources with case notions (entity types)* pid, qid *and* rid *respectively. Let CN_{PQR} be the set of the PQR-system case*

notions $\{\text{pid}, \text{qid}, \text{rid}\} \subseteq \mathbb{CN}$. *Let* $ET = (CN, E, \#), CN_{\text{PQR}} \subseteq CN$ *(Definition 2.8) be an event table. Event table* ET *is a* correct and complete event table *iff it can be replayed over the entire system* S *according to Definition 6.14.*

However, only a subset of events in E without defined identifiers qid and rid has been often recorded in an event table in reality, as we discuss in Section 7.2.2. It makes such an event table *partial*.

Definition 7.7 (Partial event table, observed event, corresponds)**.** *An event table* $ET' = (\{\text{pid}\}, E', \#')$*, where events in* E *have attribute names in some set* $AN' \subseteq AN$*, is a* partial *(and correct) event table of PQR-system* S *if there exist a correct and complete event log* $ET = (CN, E, \#)$ *of* S *such that:*

1. $E' \subseteq E$*, and* $\#' = \#\big|_{\mathcal{E}' \times AN'}$*,*
2. *no queue or resource identifiers are defined for events in* E'*, i.e.,* $\forall cn \in CN_{\text{PQR}} \setminus \{\text{pid}\}, e \in E', \#_{cn}(e) = \perp$*,*
3. *for each* $e \in E'$ $\#_{\text{time}}(e) \neq \perp$*, and*
4. *for each complete trace* $\sigma(ET, \text{pid} = id) = \langle e_1, \ldots, e_n \rangle$*, the partial trace* $\sigma(ET', \text{pid} = id) = \langle f_1, \ldots, f_k \rangle$ *contains at least the first and last event of the complete trace, i.e.,* $e_1 = f_1$ *and* $e_n = f_k$*.*

We call each event in E' *an* observed event *and say that complete event table* ET corresponds to *the partial event table* ET'*.*

For example, in Figure 7.11(c) a complete and correct event table corresponds to a recorded partial event table whose (observed) events are highlighted.

In general, events of a partial event log are less ordered because case notions qid and rid are missing. For example, the observed events of the incomplete event table in Figure 7.11(c), shown as filled boxes in Figure 7.13(b), are less ordered than in the complete event table. For example, events $e1$ and $e5$, and events $e5$ and $e7$ are unordered, while they are ordered in the complete event table.

This allows hypothesizing that this missing ordering can be inferred for reconstructed unobserved events if their qid and rid identifiers are known (reconstructed). We formulated it next.

Formal Problem Statement. Given a PQR-system S (see Definition 6.7), and a partial log ET_1 of S (Definition 7.7), we want to reconstruct a complete and correct event table ET_2 of S (Definition 7.6) that corresponds to ET_1.

However, restoring *exact* timestamp of unobserved events is usually infeasible and also not required for most cases. Thus, our problem formulation does not require reconstructing the exact timestamps. Our CPN replay semantics (see Chapter 2) allow firing transitions after their first moment of enabling. However, they have to fire "early" or "late" enough to not conflict with earlier or later observed events, i.e., in a way that

event	act	time	tmin	tmax
e_1	a	1	$\perp$	$\perp$
e_2	b	$\perp$	2	6
e_3	c	$\perp$	5	8
e_4	d	$\perp$	7	10
e_5	e	11	$\perp$	$\perp$

Table 7.9: Trace with unobserved events $e_2 - e_4$.

1. the minimum service time t_{sR} and waiting time t_{wQ} of the R- and Q-proclets of S,
2. and the FIFO ordering of the queues

hold. Thus, we have to reconstruct *time-windows* providing minimum and maximal timestamps for each unobserved event, resulting in the following sub-problems:

- Infer unobserved events E_{u} for all process cases in ET_1 and their relations to queues and resources, i.e., infer their missing identifiers.
- Infer for each unobserved event $e \in E_{\text{u}}$ a time-window of the earliest and latest occurrence of the event $\#_{\text{tmin}}(e), \#_{\text{tmax}}(e)$ so that setting $\#_{\text{time}}(e) = \#_{\text{tmin}}(e)$ or $\#_{\text{time}}(e) = \#_{\text{tmax}}(e)$ for $e \in E_{\text{u}}$ results in a complete and correct event table of S.

7.3.7 Performance Spectra with Uncertainty

The problem formulation above uses time intervals for defining an interval of possible timestamps for unobserved (and inferred) events, i.e., when an exact timestamp is unknown. These intervals are represented through event attributes tmin and tmax as interval $[\#_{\text{tmin}}(e), \#_{\text{tmax}}(e)]$. However, when considering multiple events of multiple traces, examples consisting purely of formulas are difficult to comprehend. We deal with these difficulties by using *performance spectra with uncertainty*, which we informally introduce in this section.

Let us consider trace σ, shown in Table 7.9. In this trace:

- each observed event has an exact timestamp $\#_{\text{time}}(e)$, while the attributes tmin and tmax are undefined (i.e., assigned to $\perp$),
- each unobserved event has a timestamp interval defined through the value of attributes tmin and tmax, while the exact timestamp value is unknown and assigned to $\perp$.

For example, for the observed event e_1 in Table 7.9, the value of attribute time is defined, i.e., $\#_{\text{time}}(e_1) = 1$, while attribute values tmin and tmax are undefined, i.e., $\#_{\text{tmin}}(e_1) = \#_{\text{tmax}}(e_1) = \perp$. For an unobserved event e_2 its timestamp interval is $[\#_{\text{tmin}}(e_2), \#_{\text{tmax}}(e_2)]$, i.e., its timestamp can have a value in $[2,6]$.

In Chapter 3, we introduced the performance spectrum (see Definition 3.7) for events with defined timestamps. For each segment (see Definition 3.1, the performance spectrum is a multiset of triples (t_a, t_b, c), i.e., a multiset of classified time intervals $[t_a, t_b]$. To extend the performance spectrum for supporting unobserved events with timestamps defined as intervals, we introduce a *performance spectrum with uncertainty*, where the performance spectrum of each segment is a multiset of triples $([t_a^{\text{tmin}}, t_a^{\text{tmax}}], [t_b^{\text{tmin}}, t_b^{\text{tmax}}], c)$; in each triple, $[t_a^{\text{tmin}}, t_a^{\text{tmax}}]$ is the interval $[\#_{\text{tmin}}(e), \#_{\text{tmax}}(e)]$ of the corresponding event e. If an event e has the value of attribute time defined, the corresponding interval is set to $[t_a^{\text{time}}, t_a^{\text{time}}]$, i.e., it represents the exact value $\#_{\text{time}}(e)$.

Figure 7.14 shows how we visualize the performance spectrum with uncertainty. For that, as usual, we draw segments and the time axis. Then, we draw points corresponding to the interval borders of each segment occurrence. For example, in the occurrence of segment (a, b) (events e_1 and e_2), event e_1 is observed while event e_2 is not, so this occurrence is visualized as a single point A at time $\#_{\text{time}}(e_1)$ and two points B_1 and B_2 for event e_2 at time $\#_{\text{tmin}}(e_2)$ and $\#_{\text{tmax}}(e_2)$ respectively, as shown in Figure 7.14(a). In Figure 7.14(a), the other occurrences are visualized in the same way. Finally, for each segment occurrence, we connect by a line

- points representing the minimal timestamp interval (e.g., line AB_1 in Figure 7.14(b)), and
- points representing the maximal timestamp interval (e.g., line AB_2 in Figure 7.14(b)).

As a result, shapes that we call *regions* materialize if the segments in the spectrum are sorted according to the control flow. For example, the region R_1 in Figure 7.14(b) represents the performance spectrum with uncertainty for the trace in Table 7.9.

In a performance spectrum with uncertainty, each region covers all possible locations of the lines visualizing the corresponding segment occurrences. The wider a region, the higher uncertainty about the exact values of the corresponding unobserved event timestamps, and vice versa. For example, a trace in Table 7.10 has shorter timestamp intervals of events e_2 and e_3 than the trace in Table 7.9. As a result, the region R_2 of its performance spectrum, shown in Figure 7.14(c), is more narrow than region R_1 in Figure 7.14(b).

In the next sections, we extensively use the performance spectrum with uncertainty to explain our method.

7.3.8 Inferring Timestamps Along Entity Traces

In Section 7.3.6, we formulated a problem of restoring a correct and complete event table ET_2 (see Definition 7.6) for PQR-system S from a partial event table ET_1 (see Definition 7.7). In this section, we solve this problem. For that, we extensively use the fact that event table ET_1 can be seen either

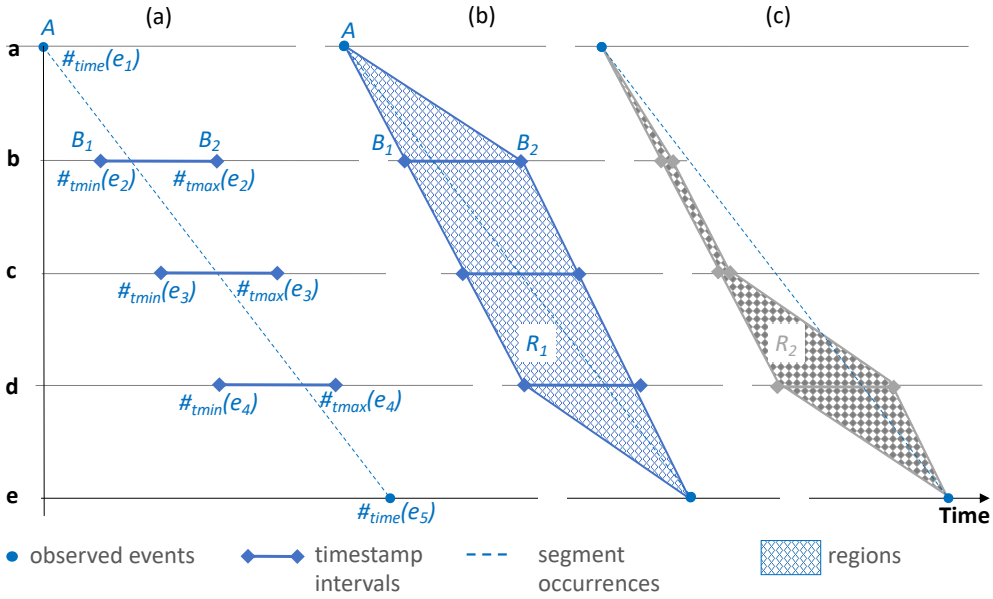


Figure 7.14: In the performance spectrum with uncertainty, the possible intervals for timestamps are shown as lines in (a), as a region in (b), and as a reduced region in (c).

event	act	time	tmin	tmax
e_1	a	1	$\perp$	$\perp$
e_2	b	2	2	3
e_3	c	5	5	6
e_4	d	7	7	10
e_5	e	11	$\perp$	$\perp$

Table 7.10: Trace with lower uncertainty of timestamps than one in Table 7.9.

- as a sequential view (see Definition 7.2) consisting of multiple sequential logs (see Definition 2.13) of entity types in CN_{PQR},
- or as a system level run $\pi(ET_1) = (E_1, <_1, \{\text{pid}\}, \#^2)$ (see Definition 7.3) with SPO $(E_1, <_1)$.

In Section 7.3.8.1, we show how to obtain an under-specified intermediate system-level run $\pi_2(ET_2) = (E_2, <_2, CN_{\text{PQR}}, \#^2)$, where E_2 also contains unobserved events $E_u = E_2 \setminus E_1$. These events still do not have timestamps, but $<_2$ already contains all ordering constraints that must hold in S.

event ID	pid	act	time	is recorded
$f1$	53	*c1*	8:00:00	no
$f3$	53	*m2*	8:00:15	no
$f5$	53	*m3*	8:00:30	no
$f6$	53	*m4*	8:00:45	no
$f9$	53	*d1*	8:01:00	yes
$f0$	54	*c3*	8:01:20	yes
$f12$	54	*m3*	8:00:35	yes
$f14$	54	*m4*	8:00:50	no
$f16$	54	*d1*	8:01:05	yes

Table 7.11: Partial event table containing events for bags 53 and 54, used as a running example in Section 7.3.8.

In Section 7.3.8.2, we refine π_2 int $\pi(ET_3) = (E_2, <_3, CN_{\text{PQR}}, \#^3)$, where $<_3$ is no longer explicitly constructed but completely inferred from timestamps that fit S. We determine minimal and maximal timestamps $\#^3_{\text{tmin}}$ and $\#^3_{\text{tmax}}$ for each unobserved event $e \in E_{\text{u}}$ (through a linear program) so that if we set $\#^3_{\text{time}} = \#^3_{\text{tmin}}$ or $\#^3_{\text{time}} = \#^3_{\text{tmax}}$, the induced partial order $<_3$ refines $<_2$, i.e., $<_2 \subseteq <_3$. By construction of $\#^3_{\text{tmin}}$ and $\#^3_{\text{tmax}}$, ET_3 is a complete and correct event table of S and has ET_1 as a partial event table.

In this section, we use another example for two bags 53 and 54, processed in the system of Figure 7.11. The corresponding event table is shown in Table 7.11, and the traces of logs of its sequential view are shown in Figure 7.15.

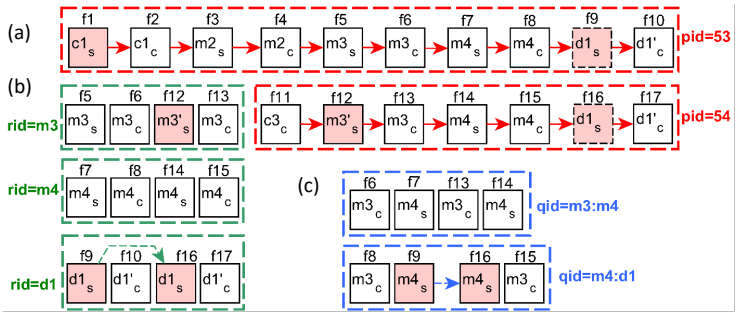


Figure 7.15: Partially complete traces of the Process (a), Resource (b), and Queue (c) proclets, restored by oracles O_1, O_2. Only observed events are ordered, e.g., $f9 <^{d1}_{\text{rid}} f16$, while the other events are isolated.

7.3.8.1 Infer Potential Complete Runs from a Partial Run

We first infer from the partial event table ET_1 an under-specified intermediate system-level run π_2 containing all unobserved events and an explicitly constructed SPO $<_2$ so

that each entity-level run $\pi^{id}_{2,\text{pid}}$ is complete (i.e.,, can be replayed on the process proclet in S). In a second step, we relate each unobserved event $e \in E_\text{u}$ to a corresponding resource and/or queue identifier that orders observed events with respect to $<_\text{rid}$ and $<_\text{qid}$. All unobserved events in E_u lack a timestamp and hence are left unordered with respect to $<_\text{rid}$ and $<_\text{qid}$ in ET_2. We later refine $<_2$ in Section 7.3.8.2.

We specify how to solve each of these two steps in terms of two *oracles* O_1 and O_2, and describe concrete implementations for either.

Restoring Process Traces. Oracle O_1 has to return a set of sequential traces $L_2 = \{\sigma^{id}_\text{pid} \mid id \in \text{pid}(L_1)\} = O_1(L_1, S)$ by completing each partial trace $\sigma(L_1, \text{pid} = id)$ of any process case $id \in \text{pid}(L_1)$ into a complete trace σ^{id}_pid that can be replayed on the P-proclet of S (see Definition 6.13). Let $E_2 = \{e \in \sigma^{id}_\text{pid} \mid \sigma^{id}_\text{pid} \in L_2\}$. The restored *unobserved* events $E_\text{u} = E_2 \setminus E_1$ only have attributes act and pid, and events are *totally ordered* along pid in each trace σ^{id}_pid. O_1 can be implemented using well-known trace alignment [43] by aligning each sequential trace $\sigma(L_1, \text{pid} = id)$ on the labeled Petri net (P, T, F, ℓ) of the P-proclet of S. For example, applying O_1 on the partial log of Table 7.11 results in the complete process traces of Figure 7.15(a).

At this point, the events $e \in E_\text{u}$ have no timestamps and the ordering of events is only available in the explicit sequences $\sigma^{id}_\text{pid} = \langle e_1, \ldots, e_n \rangle$. Until we have determined $\#_\text{time}(e_i)$, the SPO $<_2$ has to be constructed explicitly from the ordering of events in the traces σ^{id}_pid, i.e., we define $<_2$ as $e_i < e_j$ iff there exists a trace $\langle \ldots, e_i, \ldots, e_j, \ldots \rangle = \sigma^{id}_\text{pid} \in L_2$ (see Corollary 7.5.1).

Moreover, as each trace σ^{id}_pid can be replayed over the process proclet, each event is either a *start* event (i.e., replays a start transition) or a *complete* event (replays a complete transition, see Definition 6.13).

Inferring Dependencies Due To Shared Resources and Queues. Oracle O_2 has to enrich events in E_2 with information about queues and resources so that for each $e_2 \in E_2$ if resource r is involved in the step $\#_\text{act}(e)$, then $\#_\text{rid}(e) = r$ and if queue q was involved, then $\#_\text{qid}(e) = q$.

Moreover, in order to formulate the linear program to derive timestamps in a uniform way, each event e has to be annotated with the performance information of the involved resource and/or queue. That is, if e is a start event and $\#_\text{rid}(e) = r \neq \bot$, then $\#_\text{tsr}(e)$ and $\#_\text{twr}(e)$ hold the minimum service and waiting time of r, and if $\#_\text{qid}(e) = q \neq \bot$, then $\#_\text{twq}(e)$ hold the minimum waiting time of q.

For the concrete PQR-system considered in this section, we set $\#_\text{rid}(e) = r$ based on the model S if r is the identifier of the resource proclet that synchronizes with a transition that generated e via a synchronous channel of S (there is at most one). Attributes $\#_\text{tsr}(e)$ and $\#_\text{twr}(e)$ can be set from the model as they are parameters of the resource proclet. To ease the LP formulation, if e is unrelated to a resource, we set $\#_\text{rid}(e) = r^*$ to fresh identifier and $\#_\text{tsr}(e) = \#_\text{twr}(e) = 0$. Values $\#_\text{rid}(e) = r$ and $\#_\text{twq}(e)$ are set correspondingly. By annotating the events in E_2 as stated above, we obtain

$\pi_2 = (E_2, <_2, CN_{\text{PQR}}, \#^2)$. Moreover, we can update the SPO $<_2$ by inferring $\lessdot_{\text{rid}}$ and $\lessdot_{\text{qid}}$ from $\#_{\text{time}}(e)$ for all events where $\#_{\text{rid}}(e) \neq \perp$ and $\#_{\text{qid}}(e) \neq \perp$ (see Definition 7.3).

The system-level run π_2, contains complete entity-level runs for pid (except for missing timestamps). The entity-level runs of queues (qid) and resources (rid) already contain all events to be complete with respect to S but only the observed events are ordered (due to their time stamps). For example, Figure 7.15(b) shows the entity-level run $\pi^{md:d1}_{\text{qid}}$ containing events f_8, f_9, f_{16}, f_{15} with only $f_9 <_{\text{qid}} f_{16}$. Next, we define constraints based on the information in this intermediate run π to infer timestamps for all unobserved events.

7.3.8.2 Restoring Timestamps of Unobserved Events by Linear Programming

The SPO $\pi_2 = (E_2, <_2, CN_{\text{PQR}}, \#^2)$ obtained in Section 7.3.8.1 from partial log ET_1 includes all unobserved events $E_{\text{u}} = E_2 \setminus E_1$ of the correct and complete event table but lacks timestamps for each event in E_{u}, i.e., $e \in E_{\text{u}}, \#_{\text{time}}(e) = \perp$. Each observed $e \in E_1$ has a timestamp $\#_{\text{time}}(e)$ and we also added minimum service time $\#_{\text{tsr}}(e)$, minimum waiting time $\#_{\text{twr}}(e)$ of the resource $\#_{\text{rid}}(e)$ involved in e, and minimum waiting time $\#_{\text{twq}}(e)$ of the queue involved in e. We now define a constraint satisfaction problem that specifies the earliest $\#_{\text{tmin}}(e)$ and latest $\#_{\text{tmax}}(e)$ timestamps for each $e \in E_{\text{u}}$ so that all earliest (latest) timestamps yield a consistent ordering of all events in E with respect to $<_{\text{pid}}$ (events follow the process), $<_{\text{rid}}$ (events follow the resource life-cycle), and $<_{\text{qid}}$ (events satisfy the queueing behavior). The problem formulation propagates the known $\#_{\text{time}}(e)$ values along with the different case notions $<_{\text{pid}}$, $<_{\text{rid}}$, $<_{\text{qid}}$, using $t_{\text{sR}}, t_{\text{wR}}$, and t_{wQ}. For that, we introduce variables x_e^{tmin}, $x_e^{\text{tmax}} \geq 0$ for representing event attributes tmin, tmax of each event $e \in E_{\text{u}}$. For all observed events $e \in E_1$, we set $x_e^{\text{tmin}} = x_e^{\text{tmax}} = \#_{\text{time}}(e)$ as here the correct timestamp is known. We now define two groups of constraints to constrain the x_e^{tmin} and x_e^{tmax} values for the unobserved events further.

In the following, to have simpler constraints, we assume that all observed events are *start events* (which is in line with logging in the systems we focus on). These constraints can easily be reformulated to assume only complete events were observed (as in most non-BHS event logs), or a mix (requiring further case distinctions).

Propagate Information along Process Traces. The first group propagates constraints for attribute $\#_{\text{time}}(e)$ along $<_{\text{pid}}$, i.e., for each process-level run (viz. process trace) π^{id}_{pid} of pid in π. By the steps in Section 9.3.2, events in π^{id}_{pid} are totally ordered and we derived from the trace $\sigma^{id}_{\text{pid}} = \langle e_1 \dots e_m \rangle$. Each process step has a *start* and *complete* event in σ^{id}_{pid}, i.e., $m = 2 \cdot y, y \in \mathbb{N}$, odd events are *start* events, and even events are *complete* events. For each process step $1 \leq i \leq y$, the time between start event e_{2i-1} and complete event e_{2i} is at least the service time of the resource involved (which we stored as $\#_{\text{tsr}}(e_{2i-1})$ in Section 9.3.2). Thus, the following constraints must hold for

the earliest and latest time of e_{2i-1} and e_{2i}:

$$x^{\text{tmin}}_{e_{2i}} = x^{\text{tmin}}_{e_{2i-1}} + \#_{\text{tsr}}(e_{2i-1}), \tag{7.1}$$

$$x^{\text{tmax}}_{e_{2i}} = x^{\text{tmax}}_{e_{2i-1}} + \#_{\text{tsr}}(e_{2i-1}). \tag{7.2}$$

For the remainder, it suffices to formulate constraints only for *start* events. We make sure that tmin and tmax define a proper interval for each *start* event:

$$x^{\text{tmin}}_{e_{2i-1}} \leq x^{\text{tmax}}_{e_{2i-1}}. \tag{7.3}$$

We write $e^s_i = e_{2i-1}$ for the start event of the i-th process step in σ^{id}_{pid} and write $\theta^{id}_{\text{pid}} = \langle e^s_1, \ldots, e^s_m \rangle$ for the sub-trace of *start* events of σ^{id}_{pid}. Any event $e^s_i \in \theta^{id}_{\text{pid}}$ that was observed in ET_1, i.e., $e^s_i \in E_1$, has $\#^2_{\text{time}}(e^s_i) \neq \perp$ defined. By Definition 7.7, σ^{id}_{pid} as well as θ^{id}_{pid} always start and end with observed events, i.e., $e^s_1, e^s_y \in E_1, \#^2_{\text{time}}(e^s_1) \neq \perp$, and $\#^2_{\text{time}}(e^s_y) \neq \perp$. An unobserved event e^s_i has no timestamp $\#^2_{\text{time}}(e^s_i) = \perp$ yet, but $\#^2_{\text{time}}(e^s_i)$ is bounded by $\#^2_{\text{time}}(e^s_1)$ (minimally) and $\#^2_{\text{time}}(e^s_y)$ (maximally). Furthermore, any two succeeding start events in $\theta^{id}_{\text{pid}} = \langle \ldots, e^s_{i-1}, e^s_i, \ldots \rangle$ are separated by the service time $\#^2_{\text{tsr}}(e^s_{i-1})$ of step e^s_{i-1} and the waiting time $\#^2_{\text{twq}}(e_i)$ of the queue from e_{i-1} to e_i. Similarly to Equation 7.1 and Equation 7.2, we formulate this constraint for both x^{tmin}_e and x^{tmax}_e variables:

$$x^{\text{tmin}}_{e^s_k} \geq x^{\text{tmin}}_{e^s_{k-1}} + (\#^2_{\text{tsr}}(e^s_{k-1}) + \#^2_{\text{twq}}(e^s_k)), \tag{7.4}$$

$$x^{\text{tmax}}_{e^s_k} \leq x^{\text{tmax}}_{e^s_{k+1}} - (\#^2_{\text{tsr}}(e^s_k) + \#^2_{\text{twq}}(e^s_{k+1})). \tag{7.5}$$

Figure 7.16 uses the performance spectrum with uncertainty to illustrate the effect of applying our approach step by step to the partially complete traces of Figure 7.15, obtained in the steps of Section 9.3.2. The straight lines in Figure 7.16(a) from f_1 to f_9 (for pid = 53) and from f_{12} to f_{16} (for pid = 54) illustrate that ET_2 (after applying O_1) contains all intermediate steps that both process cases passed through but not their timestamps. Further (after applying O_2), we know for each process step the resources (i.e., *c1*, *m2*, *m3*, *m4*, *d1*) and the queues (*c1*: *m2*, *m2*: *m3*, etc.), and their minimum service and waiting times $t_{\text{sR}}, t_{\text{wR}}, t_{\text{wQ}}$. The sum $t_{\text{sR}} + t_{\text{wQ}}$ is visualized as bars on the time axis in Figure 7.16(a), the duration of t_{wR} is shown in Figure 7.16(b).

We now explain the effect of applying Equation 7.4 on pid = 53 for f_3, f_5 and f_7. We have $\theta^{53}_{\text{pid}} = \langle f_1, f_3, f_5, f_7, f_9 \rangle$ with f_1 and f_9 observed, thus $x^{\text{tmin}}_{f_i} = x^{\text{tmax}}_{f_i} = \#^2_{\text{time}}(f_i)$ for $i \in \{1, 9\}$. By Equation 7.4, we obtain the lower-bound for the time for f_3 by $x^{\text{tmin}}_{f_3} \geq x^{\text{tmin}}_{f_1} + \#^2_{\text{tsr}}(f_1) + \#^2_{\text{twq}}(f_3)$ with $\#^2_{\text{tsr}}(f_1)$ and $\#^2_{\text{twq}}(f_3)$ the service time of resource

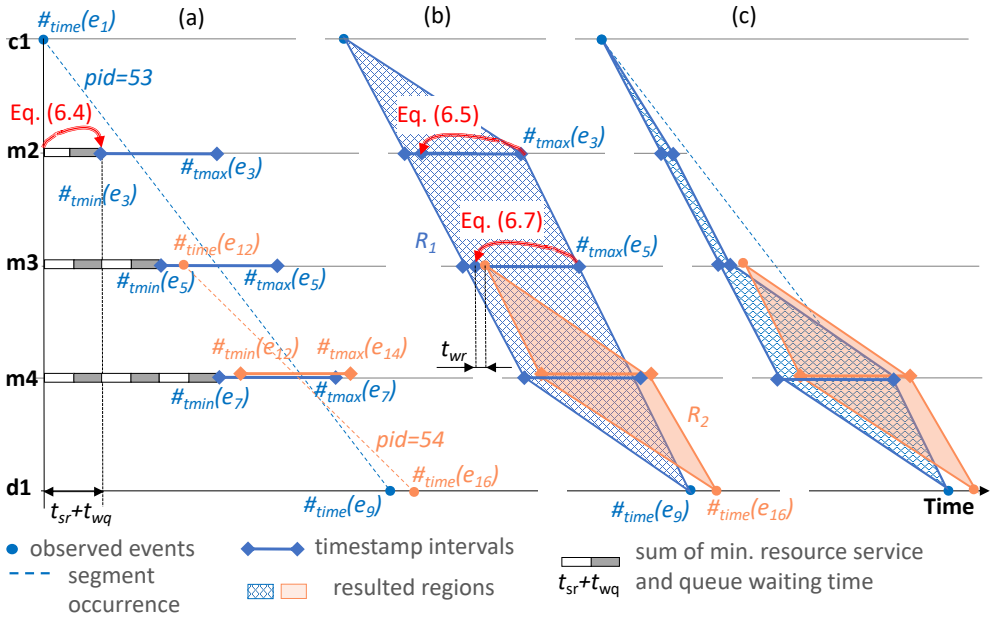


Figure 7.16: Equations 7.1-7.5 define time intervals for unobserved events (a), defining regions for the possible traces (b). Equations 7.6-7.7 propagate orders of cases observed on one resource to other resources (b), resulting in tighter regions (c).

c1 and the waiting time of queue $c1:m2$. Similarly, Equation 7.4 gives the lower bound for f_5 from the lower bound of f_3, etc. Conversely, the upper bounds $x^{\text{tmax}}_{f_i}$ are derived from f_9 "downwards" by Equation 7.5. This way, we obtain for each $f_i \in \theta^{53}_{\text{pid}}$ an initial interval for the time of f_i between the bounds $x^{\text{tmin}}_{f_i} \leq x^{\text{tmax}}_{f_i}$ as shown by the intervals in Figure 7.16(a). As $x^{\text{tmin}}_{f_1} = x^{\text{tmax}}_{f_1} = \#^2_{\text{time}}(f_1)$ and $x^{\text{tmin}}_{f_1} = x^{\text{tmax}}_{f_1} = \#^2_{\text{time}}(f_9)$, the lower and upper bounds for the unobserved events in θ^{53}_{pid} form a polygon as shown in Figure 7.16(b). Case 53 must have passed over the process steps and resources as a path inside this polygon, i.e., the polygon contains all admissible solutions for the timestamps of the unobserved events of θ^{53}_{pid}. We call this polygon the *region* of case 53. The region for case 54 overlays with the region for case 53.

Propagate Information along Resource Traces. We now introduce a second group of constraints by which we infer more tight bounds for $x^{\text{tmin}}_{e_i}$ and $x^{\text{tmax}}_{e_i}$ based on the overlap with other regions. While the first group of constraints traversed token trajectories along pid (i.e., process traces), the second group of constraints traverses token trajectories for resources along rid.

Each resource trace π^r_{rid} in π contains all events E^r_{rid} resource r was involved in - *across* multiple different process traces. The SPO $<^r_{\text{rid}}$ orders *observed* events of

this resource trace due to their known timestamps. For example, in Figure 7.16(b) $f_9 <^{m1}_{\text{rid}} f_{16}$ with f_9 from pid = 53 and f_{16} from pid = 54.

The order of the two events $e^s_{p1} <^r_{\text{rid}} e^s_{p2}$ for the *same* step $\#^2_{\text{act}}(e^s_{p1}) = \#^2_{\text{act}}(e^s_{p2}) = t_1$ in different cases $\#^2_{\text{pid}}(e^s_{p1}) = p1 \neq \#^2_{\text{pid}}(e^s_{p2}) = p2$ propagates "upwards" and "downwards" the process traces π^{p1}_{pid} and π^{p2}_{pid} as follows. Let events $f^s_{p1} \in E^{p1}_{\text{pid}}$ and $f_{p2} \in E^{p2}_{\text{pid}}$ be events in process traces π^{p1}_{pid} and π^{p2}_{pid} of the same step $\#^2_{\text{act}}(f^s_{p1}) = \#^2_{\text{act}}(f^s_{p2}) = t_n$. We say t_1 and t_n are *in FIFO relation* iff there is a unique path $\langle t_1 \dots t_n \rangle$ between t_1 and t_n in the process proclet (i.e., no loops, splits, parallelism) so that between any two consecutive transitions t_k, t_{k+1} only synchronize with single-server resources or FIFO queues. If t_1 and t_n are in FIFO relation, then also $f^s_{p1} <^{r2}_{\text{rid}} f^s_{p2}$ on the resource *r2* involved in t_n (as the case *p1* cannot overtake the case *p2* along this path). Thus, $x^{\text{tmin}}_{f^s_{p1}} \leq x^{\text{tmin}}_{f^s_{p2}}$ must hold. More specifically, $x^{\text{tmin}}_{f^s_{p1}} + \#^2_{\text{tsr}}(f^s_{p1}) + \#^2_{\text{twr}}(f^s_{p1}) \leq x^{\text{tmin}}_{f^s_{p2}}$ must hold as the service time and waiting time of the resource involved in f^s_{p1} must elapse.

For any pair $e^s_{p1}, e^s_{p2} \in E^r_{\text{rid}}$ with $e^s_{p1} <^r_{\text{rid}} e^s_{p2}$ and any other trace θ^{r2}_{rid} for resource *r2* and any pair $f^s_{p1}, f^s_{p2} \in E^{r2}_{\text{rid}}$ such that $\#^2_{\text{pid}}(e^s_{p1}) = \#^2_{\text{pid}}(f^s_{p1}), \#^2_{\text{pid}}(e^s_{p2}) = \#^2_{\text{pid}}(f^s_{p2})$ and transition $\#^2_{\text{act}} e^s_{p1}$ that is in FIFO relation with $\#^2_{\text{act}}(f^s_{p1})$, we generate the following constraint for tmin between different process cases *p1* and *p2*:

$$x^{\text{tmin}}_{f^s_{p1}} \leq x^{\text{tmin}}_{f^s_{p2}} - (\#^2_{\text{tsr}}(f^s_{p1}) + \#^2_{\text{twr}}(f^s_{p1})), \tag{7.6}$$

and the following constraint for tmax:

$$x^{\text{tmax}}_{f^s_{p1}} \leq x^{\text{tmax}}_{f^s_{p2}} - (\#^2_{\text{tsr}}(f^s_{p1}) + \#^2_{\text{twr}}(f^s_{p1})). \tag{7.7}$$

In the example of Figure 7.16(b), we observe $f_9 <^{d1}_{\text{rid}} f_{16}$ (both of transition $d1_s$) along resource $d1$ at the bottom of Figure 7.16(b). By Figure 7.12, $d1_s$ and $m3_s$ are in FIFO-relation. Applying Equation 7.7 yields $x^{\text{tmax}}_{f_5} \leq \#^2_{\text{time}}(f_{12}) - (\#^2_{\text{tsr}}(f_5) + \#^2_{\text{twr}}(f_5))$, i.e., f_5 occurs at latest before f_{12} minus the service and waiting time of $m3$. This operation significantly reduces the initial region R_1. By Equation 7.5, the tighter upper bound for f_5 also propagates along the trace pid = 53 to f_3, i.e., $x^{\text{tmax}}_{f_3} \leq x^{\text{tmax}}_{f_5} - (\#^2_{\text{tsr}}(f_3) + \#^2_{\text{twq}}(f_5))$, resulting in a tighter region as shown in Figure 7.16(c). If another trace $\langle m3_s, d1_s \rangle$ were present *before* trace 53, then this would cause reducing the tmin attributes of the events of trace 53 by Equations 7.4 and 7.6 in a similar way. In general, the more cases interact through shared resources, the more accurate timestamp intervals can be restored by Equation 7.1-Equation 7.7 as we show in Section 9.4.

To construct the linear program, we generate Equation 7.1 to Equation 7.5 by iterating over each process trace in ET_2. Further, by iterating over each resource trace and each pair of events $e_{p1} <^r_{\text{rid}} e_{p2}$, we generate Equation 7.6 and Equation 7.7 for each other pair of events $f_{p1} <^{r2}_{\text{rid}} f_{p2}$ that is in FIFO relation. The objective function to maximize is the sum of all intervals $\sum_{e \in E_2} (x^{\text{tmax}}_e - x^{\text{tmin}}_e)$, to maximize the coverage of possible timestamp values by those intervals.

Solving this linear program assigns to each event $e \in E_2$ upper and lower bounds $\#_{\text{tmin}}(e)$ and $\#_{\text{tmax}}(e)$ for timestamp $\#_{\text{time}}(e)$. For all $e \in E_1, \#_{\text{tmin}}(e) = \#_{\text{tmax}}(e) = \#_{\text{time}}(e)$ (by Equation 7.1 and Equation 7.2 the solutions for the *start* events propagate to *complete* events with time difference t_{sR}). Be assigning $\#_{\text{time}}(e) = \#_{\text{tmin}}(e)$ (or $\#_{\text{time}}(e) = \#_{\text{tmax}}(e)$), we obtain $ET_3 = (E_2, CN_{\text{PQR}}, \#^3)$ where SPO $<_3$ of the system-level run $\pi(ET_3)$ refines SPO $<_2$ constructed explicitly in Section 9.3.2.

By oracle O_1, $\sigma(ET_3, \text{pid})$ can be replayed on the P-proclet.

By Equation 7.1 and Equation 7.6, for any two events $e \lessdot_{\text{rid}} e'$ the time difference is $\#_{\text{time}}(e') - \#_{\text{time}}(e) > t_{\text{wR}}$ or $\#_{\text{time}}(e') - \#_{\text{time}}(e) > t_{\text{sR}}$ of the corresponding R-proclet R_i (depending on whether e replays by the *start* or *complete* transition of R_i). Thus, $\sigma(ET_3, r)$ can be replayed on the corresponding R-proclet for any resource $r \in ET_3$.

By Equation 7.1 and Equation 7.4, the timestamps of $e \lessdot_{\text{pid}} e'$ where e replays *enq* and e' replays *deq* of a Q-proclet Q_i have at least time difference t_{wQ} of Q_i (i.e., the time constraint of Q_i is satisfied). If for two process cases $p1$ and $p2$ we observe $e_{p1} <_{\text{rid}} e_{p2}$ at the same step $\#_{\text{act}}(e_{p1}) = \#_{\text{act}}(e_{p2})$ with $\#_{\text{pid}}(e_{p1}) = p1 \neq p2 = \#_{\text{pid}}(e_{p1})$ at some step, we also observe $f_{p1} <_{\text{rid}} f_{p2}$ at another step $\#_{\text{act}}(f_{p1}) = \#_{\text{act}}(f_{p2})$ with $\#_{\text{pid}}(f_{p1}) = p1 \neq p2 = \#_{\text{pid}}(f_{p1})$ at later events $e_1 <_{\text{pid}} f_1$ and $e_2 <_{\text{pid}} f_2$ (by Equation 7.6 and Equation 7.7). As in a PQR-system, for each queue, the enqueue transition synchronizes with a different resource than the dequeue transition, the relation $e_{p1} <_{\text{qid}} e_{p2}$ and $f_{p1} <_{\text{qid}} f_{p2}$ also holds if e_{p1}, e_{p2} are enqueue events and f_{p1}, f_{p2} are dequeue events of the same queue Q_i. Thus the FIFO constraint of Q_i is satisfied. Thus, $\sigma(ET_3, q)$ can be replayed on the correspond Q-proclet for any queue q in ET_3.

Altogether, ET_3 is a complete and correct event table that can be replayed on the PQR-system S (by Definitions 6.7 and 6.14).

Computational Complexity. Last but not least, the computational complexity of this method is of crucial importance in real-world settings. It depends on the computational complexity of

1. inferring all the constraints (Equations 7.1-7.7),
2. solving the linear program.

The worst-case complexity of the former we estimate as quadratic, assuming that each event has constraints with respect to all the other events for each process step and resource. Note, we do not take into account the number of constraints, process steps, and resources because it is usually much less than the number of events in an event log. To estimate the computational complexity of the latter, we assume the use of Dantzig's simplex algorithm (or simplex method) [171], whose numerous specializations and generalizations have dominated operations research for decades [146]. Although its worst-case computational complexity is exponential, it is often polynomial in practice. As a result, the complexity of solving a linear program defines the complexity of our method. Potentially, it is possible to split an input event log into

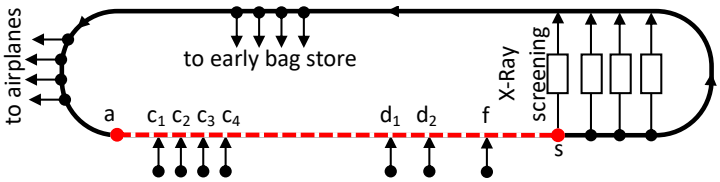


Figure 7.17: In the BHS bags come from check-in counters c_{1-4} and another terminals d_{1-2}, f, go through mandatory screening and continue to other locations.

partitions and apply the method for each partition individually to make its scalability easier.

7.3.9 Evaluation

To evaluate our approach, we formulated the following questions.

- (Q1) Can timestamps be estimated in real-life settings and used to estimate performance reliably?
- (Q2) How accurately can the load (items per minute) be estimated for different system parts, using restored timestamps?
- (Q3) What is the impact of sudden deviations from the minimum service/waiting times, e.g., the unavailability of resources or stop/restart of a BHS conveyor, on the accuracy of restored timestamps and the computed load?

For answering them, we extended the interactive ProM plug-in "Performance Spectrum Miner" with an implementation of our approach that solves the constraints using heuristics[1]. As input, we considered the process of a part of real-life BHS shown in Figure 7.17 and used Synthetic Logs (SL) (simulated from a model to obtain ground-truth timestamps) and Real-life Logs (RL) from a major European airport. Regarding Q3, we generated SL with regular performance and with *blockages* of belts (i.e., a temporary stand-still); the RL contained both performance characteristics. All logs were partial as described in Section 7.3.2. We selected the acyclic fragment highlighted in Figure 7.17 for restoring timestamps of steps c_{1-4}, d_{1-2}, f, s.

We evaluated our technique against the ground truth known for SL as follows. For each event we measured the error of the estimated timestamp intervals $[t_{\min}, t_{\max}]$ against the actual time t as $\max(|t_{\max} - t|, |t_{\min} - t|)$, normalized over the sum of the minimum service and waiting times of all the involved steps (to make errors comparable). We report the Mean Absolute Error (MAE) and Root Mean Square Error (RMSE) of these errors. Applying our technique to SL with the regular behavior, we observed very narrow time intervals for the estimated timestamps, shown in Figure 7.18(a),

[1]The simulation model, simulation logs, ProM plugin, and high-resolution figures are available on `https://github.com/processmining-in-logistics/psm/tree/rel`.

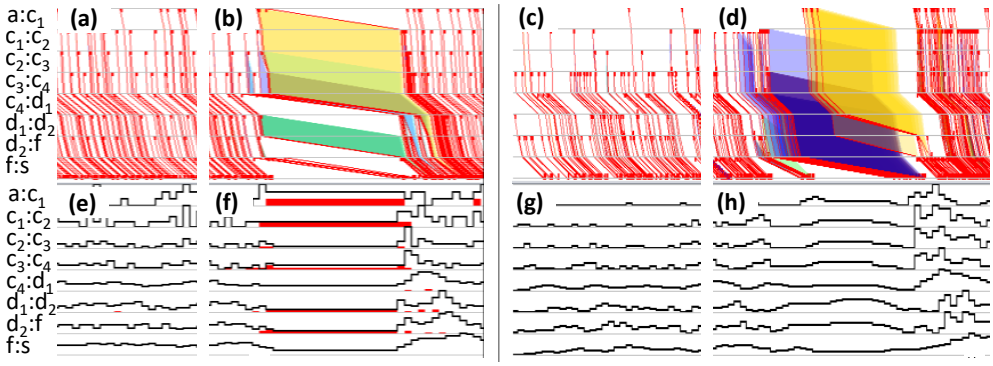


Figure 7.18: Restored performance spectrum for synthetic (a,b) and real-life (c,d) logs. The estimated load (computed on estimated timestamps) for synthetic (e,f) and real-life (g,h) logs. For the synthetic logs, the load error is measured and shown in red (e,f).

and an MAE of $< 5\%$. The MAE of the estimated load (computed on estimated timestamps), shown in Figure 7.18(e), was $< 2\%$. For SL with the blockage behavior, the intervals grew proportionally with the duration of blockages (Figure 7.18(b)), leading to a proportional growth of the MAE for the timestamps. However, the MAE of the estimated load (Figure 7.18(f)) was at most 4%. The load MAE for different processing steps for both scenarios is shown in Table 7.12. Notably, both observed and reconstructed loads showed load peaks each time the conveyor belt starts moving again.

When evaluating on the real-life event log, we measured errors of timestamp estimation as the length of the estimated intervals (normalized over the sum of the minimum service and waiting times of all the involved steps). Performance spectra, built using the restored RL logs, are shown in Figure 7.18(c,d), and the load, computed using these logs, is shown in Figure 7.18(g,h). The observed MAE was $< 5\%$ in the regular behavior and increased proportionally as observed on SL. The load error could not be measured, but similarly to the synthetic data, it showed peaks after assumed conveyor stops.

The obtained results on SL show that the timestamps can be always estimated, and the actual timestamps are always within the timestamp intervals (Q1). When the system resources and queues operate close to the known performance parameters $t_{\mathrm{sR}}, t_{\mathrm{wR}}, t_{\mathrm{wQ}}$, our approach restores accurate timestamps resulting in reliable load estimates in SL (Q2). During deviations in resource performance, the errors increase proportionally with performance deviation while the estimated load remains reliable (error $< 4\%$ in SL) and show known characteristics from real-life systems on SL and RL (Q3).

Scenario	MAE, $c_4 : d_1$	RMSE, $c_4 : d_1$	MAE, $d_1 : d_2$	RMSE, $d_1 : d_2$	MAE, $f : s$	RMSE, $f : s$
no block.	0.16	1.01	0.22	1.66	0.17	0.89
blockages	1.67	4.8	3.19	7.17	0.15	0.75

Table 7.12: Estimated load (computed on estimated timestamps) Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) are shown in % of max. load.

We showed how incomplete event logs can be repaired using the PQR-system as the source of linear programming constraints for restoring missing timestamps. Next, conclude this chapter.

7.4 Chapter Summary

In this chapter, we studied how to relate event data to the PQR-system. We chose generalized conformance checking instead of "classical" conformance checking because, in the real-world setting, we can fully trust neither event data nor PQR-system. We adopted the generalized conformance checking pipeline of [44] to our settings. We showed how the decomposition of the PQR-system into its proclet allows for the use of existing techniques for conformance checking and model repair. We identified gaps in these techniques and suggested possible approaches for bridging them. Finally, we proposed a novel approach for inferring unobserved events with timestamp information for systems with shared resources and queues. It is required for the analysis of BHSs, whose data are usually significantly incomplete.

However, the proposed PQR-system-based conformance checking has limitations. It is based on its replay semantics and can determine whether an execution can be replayed over the system. However, little to no diagnostic information is available for some conformance checking scenarios. Thus, for Q- and R-proclet trajectory conformance, if a trace is not conformable to a Q- or R-proclet, our approach neither can align the trace to the model nor can suggest how to repair the model. Another limitation of Q- and R-proclet trajectory conformance is assuming the support for `list` structures in data-aware Petri nets. Additionally, our PQR-system conformance checking has not been evaluated yet.

Our log repair approach has the following limitations.

- Although the proclet formalism allows for arbitrary, dynamic synchronizations between process steps, resources, and queues, we limited ourselves in this work to a static known resource/queue identifier per process step. The limitation is not severe for some use cases such as analyzing BHS, but generalizing oracle O_2 to a dynamic setting is an open problem.

- The LP constraints to restore timestamps assume an acyclic process proclet without concurrency. Further, the LP constraints assume 1:1 interactions (at most one resource and/or queue per process step). Both assumptions do not hold in business processes in general; formulating the constraints for a more general setting is an open problem.
- Our approach ensures consistency of either all earliest or all latest timestamps with the given model, it does not suggest how to select timestamps between the latest and earliest such that the consistency holds.
- When the system performance significantly changes, e.g., due to sudden unavailability of resources, the error of restored timestamps is growing proportionally with the duration of deviations.
- For evaluation, we used heuristic algorithms instead of solving a linear program. As a result, we did not evaluate the computational complexity using BHS data and did not explore the scalability of our method.

Chapter 8

Multi-Dimensional Performance Analysis

In this chapter, we discuss how performance spectra, proposed in Chapter 3, can be related to the formal process model of systems with shared resources and queues—the PQR-system, suggested in Chapter 6. For that, we first introduce the performance spectra of the queue and resource dimensions, described by the PQR-system. Additionally, we discuss how synchronization channels of a PQR-system allow for obtaining a view on the spectra of the queue and resources related to a segment of interest of the "regular" performance spectrum (describing the process control flow). We show how such a view helps explain the root causes of various performance issues. Further, we propose a method for process performance root cause analysis in systems with shared resources and queues, using the PQR-system and the performance spectra of the process (control flow), queue, and resource dimensions. We evaluated our method using both synthetic data, generated by the BHS simulation model (see Section 6.7), and datasets recorded by real Vanderlande-built BHSs.

8.1 Introduction

Performance analysis is an important element in process management relying on precise knowledge about the actual process behavior and performance [66]. It allows us to get insights into actual process performance, detect performance deviations, and explain their root causes. The latter is often referred to as *Root Cause Analysis* (RCA). RCA is crucial for process management since it helps to come up with process improvements as a result of post-mortem analysis and mitigate these deviations in real-time settings. A classical example of process performance RCA is bottleneck RCA. Approaches for process performance analysis depend on both the process type

and possible root cause. Thus, in processes with isolated cases, the analysis of cases can be done in isolation. For example, a technical support ticket can be delayed because the initial information was incomplete.

However, in this thesis, we focus on systems with shared resources and queues, where cases are not processed in isolation but affect each other. As a result, performance analysis of systems with shared resources and queues is a hard problem because it requires considering the dynamics of all cases in the system or process. In this section, we discuss state-of-the-art approaches for solving it, formulate the RQ, and outline our methods for performance analysis, which we propose in this chapter.

8.1.1 State-of-the-Art Approaches

In the following, we recall the limitations of the performance spectrum, proposed in Chapter 3, and also discuss the limitations of the method for detecting system-level behavior [77, 78]. We consider both as state-of-the-art techniques for performance analysis of systems with shared resources and queues. Their limitations are used as input for the RQ formulation later in the next section.

Performance Spectrum-Based Analysis. The performance spectrum is capable of describing the performance of both individual cases (e.g., bags) and all the cases (bags) together, thereby enabling performance analysis of systems with shared resources and queues. However, our exploratory analysis of systems with shared resources and queues in Chapter 4 showed that their performance is significantly affected by case interaction on shared conveyors and machines, so we identify the concepts of queues and resources for modeling this aspect of systems with shared resources and queues. This understanding allows identifying the following major limitation of the performance spectrum-based analysis of Chapter 3:

- the use of performance spectra describing the performance only *along the case level* (control flow) is not sufficient for explaining which behavior of the queues and/or resources caused performance problems, i.e., domain knowledge (in the analyst's mind) is additionally required.

For example, the performance spectrum in Figure 8.1 shows instances $bl_1 - bl_5$, $hl_0 - hl_5$ of performance patterns but provides no clue about *why* they occurred.

Besides that, many other factors impede performance spectrum-based analysis. For example, domain knowledge is required for:

- interpreting process steps comprising performance spectrum segments,
- sorting and following segments according to the process flow,
- interpreting duration of segment occurrences,
- choosing particular segments (among hundreds) for analysis.

As the evaluation in Section 3.5 showed, the analyst is usually delayed by:

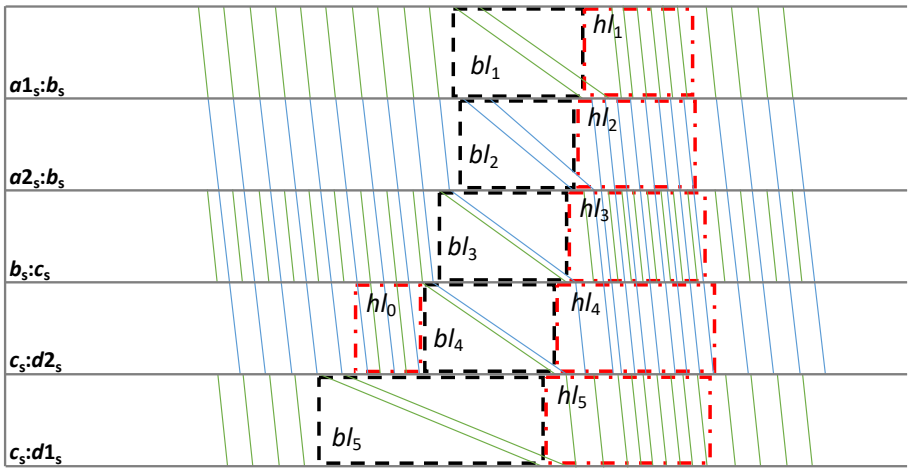


Figure 8.1: Performance spectrum with performance pattern instances $bl_1 - bl_5$, $hl_0 - hl_5$ does not explain why they occurred.

- the constant need for referring to the system MFD for relating the performance spectrum segments to the system conveyors and machines,
- involving domain experts for explaining and interpreting succinct information provided in the MFD,
- inability to see the conveyor travel time (queue waiting time) and bag handling time by a machine (i.e., the resource service time) in the total duration of segments,
- the frequent need to reorder segments in the order of a particular path in the system/process.

As a result, this analysis involves many manual operations with the tools and paper documents, and multiple communication rounds with domain experts, still without seeing behaviors and root causes in the queue and resource dimensions.

However, there is another closely related technique that addresses the performance analysis of systems with shared resources and queues, created within the same project as the methods of this thesis. Let us discuss its limitations as well.

Method for Detecting System-Level Behavior. The method for detecting system-level behavior [77] also addresses the problem of performance analysis of processes with non-isolated cases. For that, the authors introduced *segment events* on top of "classical" events, which directly correspond to segment occurrences (see Definition 3.2 and [45]). Next, they defined *system-level* events that are instances of performance patterns [45], proposed in Section 3.3, that we, in contrast, do not define formally in this thesis. On top of this event data model, they suggested a method

for automatic detection of two types (patterns) of system-level events, also building on [172].

Further, they argued that system-level events that are close to each other in space and time, form a *cascade*, i.e., multiple system events connected by correlation relationship. A detected cascade can potentially describe an undesirable performance situation. The authors also pointed out that early detection of a cascade development, leading to an undesirable performance situation, can potentially help predict performance problems at the end of a cascade development.

In this paper, the authors made a significant contribution to the problem by formalizing performance patterns, suggesting performance pattern automatic detection, and introducing cascades as "traces" describing the behavior on the level of a process or system. This work has been extended in [78], whose authors proposed a framework allowing for the detection of system-level events of many types, in order to make [77] applicable in non-MHS domains.

However, the following limitations, we believe, affect applying this method [77] and framework [78] in practice.

The method detects but does not explain bottlenecks. In practice, the detection of a performance problem is not sufficient for its mitigation or coming up with process improvements, the identification of the problem's root causes is of crucial importance. However, the authors do not suggest or discuss any approaches for explaining the root causes of cascades. For example, the question "*Why* did the blockage instances, which triggered the whole cascade, happen?", cannot be answered. We believe it cannot be answered without providing some domain knowledge (e.g., in a form of a process model) to input of [77, 78].

Longer adoption due to relying on automatic pattern detection. In general, automatic pattern detection is preferable to manual. However, it is difficult to design an algorithm that works reliably for different segments and systems due to the following reasons:

- for each segment in the system, thresholds to detect pattern instances vary [172],
- a pattern detection configuration, working for one system, may not be transferable to other systems,
- when a small number of segment events (i.e., segment occurrences) describes the performance during some period, it may be not enough information to reliably detect a system-level event (pattern instance) if a statistical approach (as in [77, 172]) is used.

We also believe that the precise information about resources' and queues' temporal parameters has to be considered for implementing a reliable pattern instance detector. As a result, *adoption* of such an approach may be difficult because it would require many iterations to adjust the detector configurations per each system segment. At the

same time, each error, leading to too large or too small cascades, would undermine domain experts' trust and decrease the chances of a successful adoption.

Shallow approach for assigning time distance directions between system-level events. To determine time distance directions, the start time of system-level events are compared in [77]. However, this approach requires detecting the start time of each performance pattern instance precisely, otherwise, the direction can be determined incorrectly. In practice, determining the start times of pattern instances is hard because segment events (occurrences) provide no information about performance between the corresponding process steps (i.e., in the middle of the segment occurrence line in the performance spectrum). As a result, a pattern instance start time can be seen more like a time interval when an instance could start rather than an exact timestamp. In both cases (as in [77] or using time intervals), a time-based approach for determining directions between system-level events is problematic. As a result, directions between system events in a cascade might not be used to determine the system events that triggered the cascade, whose identification is required for their root cause analysis.

The method accuracy is very sensitive to event data incompleteness. The method exploits spatial distances to detect correlated system-level events. It is computed based on an assumption that correlated segment events follow each other, i.e., segment (a,b) is followed by a segment started by activity b as well, e.g., (b,d), and segment (e,b) is followed by (b,f). However, when events of b are unobserved, resulting segments (a,d) and (e,f) are not close. Since event data incompleteness is a typical issue for many real-world MHSs, it can significantly affect detection results' accuracy.

To summarize, both approaches considered above have an important common limitation: they *do not explain the root causes* of detected performance problems. Moreover, the latter may be difficult to adopt in the industry. Next, we consider the RQs addressing these limitations.

8.1.2 Research Questions

In this chapter, we focus on performance analysis of systems with shared resources and queues because we identified the related **AQs** in Section 4.2. **AQ3**-**AQ5** and **AQ7** address RCA of various kinds of performance deviations, while **AQ8** (predictive performance monitoring) can potentially benefit from answering **AQ3**-**AQ5** and **AQ7**.

To formulate RQs for addressing those **AQs**, we first briefly recap what foundation we already laid in the previous chapters (see Figure 1.5 in Section 1.3 for details). First, we introduced the performance spectrum in Chapter 3. Next, we identified queues and resources as key building blocks for reasoning about systems with shared resources and queues in Chapter 4. Then, using these building blocks, we proposed the PQR-system, a process model of systems with shared resources and queues, in Chapter 6 to enable process model-aware methods and inject some domain knowl-

edge for them. Using the PQR-system, we adopted generalized conformance checking, including our novel method for inferring unobserved events, in Chapter 7. As a result, now we can build on (besides the performance spectrum):

- the PQR-system, i.e., process models of systems with shared resources and queues that describe the process, queue, and resource dimensions, and
- the availability of *complete* and *correct* event tables (see Chapter 7) that do not have missing events, and can be replayed over the PQR-system.

Assuming it helps overcome the identified limitations of the performance spectrum-based analysis proposed in Chapter 3, we formulate the following general RQ.

- **RQ-6**. *How to relate a PQR-system, describing the process (control-flow), queue and resource dimensions of a process/system, and the corresponding performance spectra computed from complete and correct event tables?*

This general RQ we further divide into two sub-questions. The first one addresses the problem of describing the performance/behavior of resources and queues using performance spectra as follows.

- **RQ-6.1**. *What are performance spectra of the queue and resource dimensions modeled by Q- and R-proclets of a PQR-system, and how synchronization among the PQR-system proclets can be described using performance spectra?*

The second one addresses the problem of RCA within the process, queue, and resource dimensions, which is not addressed by the state-of-the-art methods of Chapter 3 [45] and in [77, 78], as follows.

- **RQ-6.2**. *Given a complete and correct event table, and PQR-system, how to do root cause performance analysis of a system with shared resources and queues?*

Next, we outline our methods for **RQ-6.1** and **RQ-6.2**.

8.1.3 Method Outline

In this section, we outline how **RQ-6.1** and **RQ-6.2** have been approached in this chapter.

Relating Performance Spectra and PQR-systems. We start by introducing a simplified version of performance spectra, that does not describe *complete* transitions of process steps in Section 8.2.2. We define a resource event log and consider the corresponding resource performance spectrum in Section 8.2.3. We define a queue event log and discuss the corresponding queue performance spectrum in Section 8.2.4. Finally, we introduce the notion of a multi-dimensional context of a performance spectrum segment of the process dimension, in Section 8.2.5. This context includes the segments of the related queue and resources. As a result, a multi-dimensional context

view of the performance of the related queue and resources can be obtained. Further, this view is used for enabling RCA (for **RQ-6.2**).

Method for Multi-Dimensional Performance Analysis. We start with a motivating example that describes an undesirable performance scenario in a BHS in Section 8.3.1. We provide an overview of the method, which helps to navigate in the description of the method's steps, in Section 8.3.2. Then, we describe the following method steps, given a PQR-system and performance spectra of the process, queue, and resource dimensions as input.

- The performance pattern detection in Step 1 is described in Section 8.3.3.
- The dependencies among the detected pattern instances are discovered in Step 2. For that, the information captured in P-proclet of the given PQR-system is used to identify how these instances triggered each other, to organize them in graphs called propagation chains. This step is described in Section 8.3.5.
- In Steps 3 and 4, the propagation chains, discovered in the previous step, are merged using reasoning based on the properties of the PQR-system, and observations of performance pattern instances in performance spectra of real-world MHSs. These steps are described in Sections 8.3.6 and 8.3.7 respectively.
- In Steps 5 and 6 (Sections 8.3.8 and 8.3.9), the root causes of instances that triggered propagation chains obtained in Step 4, are identified for each propagation chain. For that, a multi-dimensional context view is analyzed for such instances. Performance pattern instances are detected in the spectra of these views. Based on the combination of the detected instances, the root cause of the instance is identified. As a result, the root causes explain why the whole propagation chain was triggered. Additionally, a propagation chain describes the whole scenario triggered by those root causes. This is the final result of the analysis.

The remainder of this chapter is organized as follows. We introduce the performance spectra of the queue and resource dimensions in Section 8.2 and describe our method for process performance analysis in Section 8.3. We discuss its evaluation in Section 8.4 and its limitations in Section 8.5.

8.2 Relating Performance Spectra to PQR-Systems

In this section, we address **RQ-6.1**. For that, we introduce a running example in Section 8.2.1 and explain what the performance spectrum describes for the process, resource, and queue dimensions, and which data are required to compute each one of them, in Sections 8.2.2,8.2.3, and 8.2.4. Finally, we discuss how performance spectra of different process dimensions can be combined using the information about

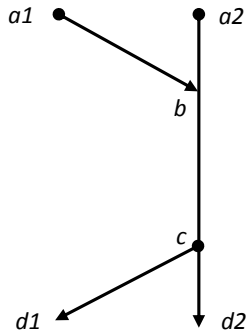


Figure 8.2: BHS material flow diagram.

transition synchronization (i.e., the synchronous channels of the PQR-system) in Section 8.2.5.

8.2.1 Running Example

For a running example, we slightly extended the BHS we consider in Chapters 6 (see Figure 6.1(a)) such that it explains typical phenomena, occurring in real MHSs, better. We transformed exit c into a diverting unit and connected it with new conveyors $c : d1$ and $c : d2$. The resulting MFD is shown in Figure 8.2, and the corresponding PQR-system is shown in Figure 8.3.

Let us explain the scenario of our running example, whose execution is provided in the form of an event table in Table 8.1. In the beginning, two bags with identifiers $pid1$ and $pid2$ were checked in at counters $a2$ and $a1$ respectively at the same time t_1. The system started moving them toward merging unit b. Unit b is potentially reachable from both check-in counters in the same minimal time δ. However, resource $rid3$ at b can handle only one bag at a time. As a result, while bag $pid1$ indeed reached b at time $t_3 = t_1 + \delta$, $pid2$ had to wait till time t_{11} on conveyor $a1 : b$ since this conveyor had a lower priority at b. As a result, its travel time to b was longer, i.e., $t_{11} - t_1 > \delta = t_3 - t_1$. After merging at b, both bags continued to their final destinations $d2$ and $d1$ via c without delays. Note, the minimal travel times from c to $d1$ and $d2$ are equal in this BHS.

The event table (Table 8.1) shows events for both bags for each transition of the PQR-system they "passed". Whenever multiple transitions of different proclets synchronized, the corresponding event has the attributes for the corresponding case notions defined. For example, when resource $rid2$ (check-in counter) at process step $a2$ (check-in counter location $a2$) handed bag $pid1$ over to queue $qid2$ (conveyor $a2 : b$), the corresponding event $e2$ got these case notion identifiers as its attributes.

For this toy example, we formulate the following analysis question.

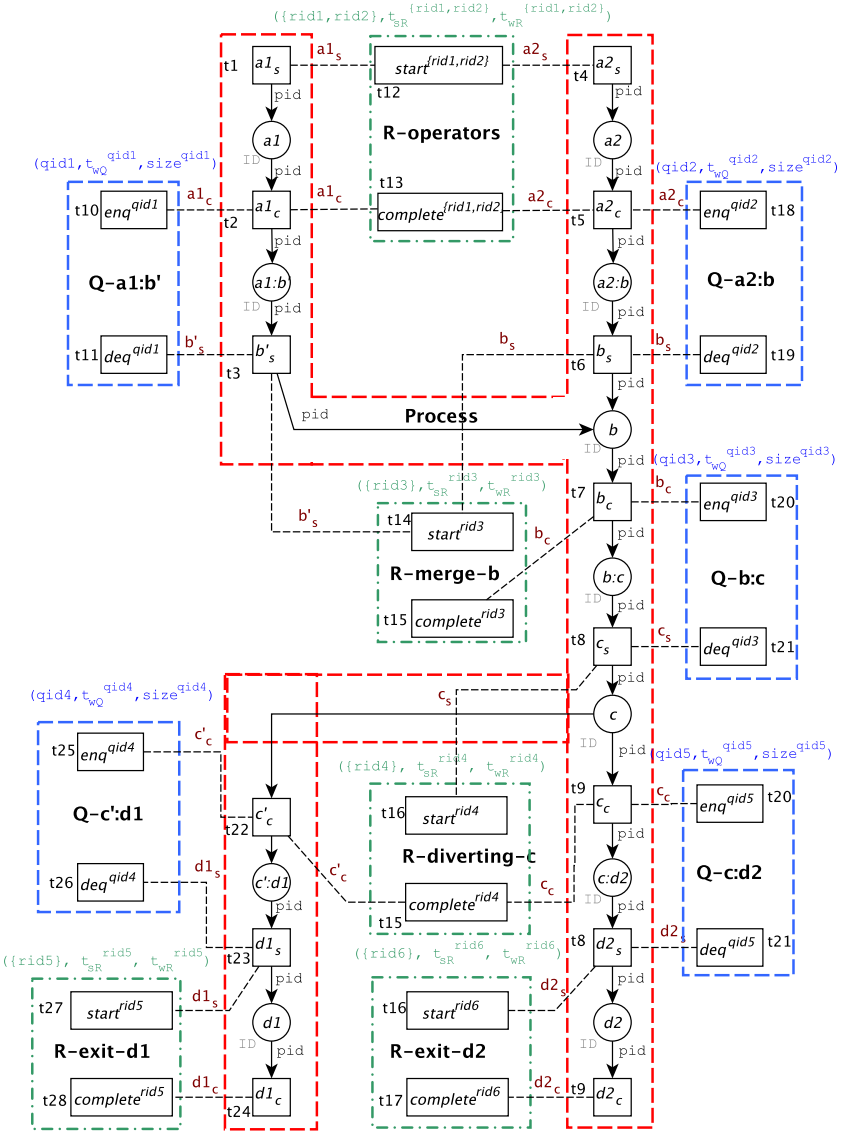


Figure 8.3: PQR-system of the BHS of Figure 8.2.

- **ExampleAQ**. Why did bag *pid*2 arrive at its final destination later than *pid*1 despite the minimal travel time from $a1$ to $d1$ and from $a2$ to $d2$ being equal?

Event	Activity	Time	pid	qid	rid
$e1$	$a2_s$	t_1	$pid1$	-	$rid2$
$e2$	$a2_c$	t_2	$pid1$	$qid2$	$rid2$
$e3$	b_s	t_3	$pid1$	$qid2$	$rid3$
$e4$	b_c	t_4	$pid1$	$qid3$	$rid3$
$e5$	c_s	t_5	$pid1$	$qid3$	$rid4$
$e6$	c_c	t_6	$pid1$	$qid5$	$rid4$
$e7$	$d2_s$	t_7	$pid1$	$qid5$	$rid6$
$e8$	$d2_c$	t_8	$pid1$	-	$rid6$
$e9$	$a1_s$	t_1	$pid2$	-	$rid1$
$e10$	$a1_c$	t_2	$pid2$	$qid1$	$rid1$
$e11$	b'_s	t_9	$pid2$	$qid1$	$rid3$
$e12$	b_c	t_{10}	$pid2$	$qid3$	$rid3$
$e13$	c_s	t_{11}	$pid2$	$qid3$	$rid4$
$e14$	c_c	t_{12}	$pid2$	$qid4$	$rid4$
$e15$	$d1_s$	t_{13}	$pid2$	$qid4$	$rid5$
$e16$	$d1_c$	t_{14}	$pid2$	-	$rid5$

Table 8.1: Execution of the PQR-system of Figure 8.3.

In the following, we show how performance spectra can be computed for the process, queue, and resource dimensions and used for answering the **ExampleAQ**.

8.2.2 Relating P-Proclets to Process Performance Spectra

The P-proclet (see Definition6.2) describes the process dimension, e.g., how process steps can be executed within a baggage handling process. When an event table ET (Definition 2.8) is an execution of the process modeled by this PQR-system, an event log L^{ET}_{pid} for process case notion pid can be derived from ET according to Definition 2.12. Then, for segments SEG (Definition 3.1) and performance classifier $\mathbb{C}$ (Definition 3.3), defining a layer lr (see Definition 3.6), a performance spectrum $\mathbb{PS}^{\text{seg}}_{L^{ET}_{\text{pid}}}(lr)$ of the process dimension can be computed according to Definition 3.7, as we showed in Chapter 3.

Thus, event log L^{ET}_{pid}, computed from table in Table 8.1, contains two traces:

- $\sigma_{pid1} = \langle e1, \ldots, e8 \rangle$, and
- $\sigma_{pid2} = \langle e9, \ldots, e16 \rangle$.

The process performance spectrum, computed from this event log for segment sequence

- $\langle (a2_s, a2_c)$,
- $(a2_c, b_s)$,

- (b_s, b_c),
- (b_c, c_s),
- (c_s, c_c),
- $(c_c, d2_s)$,
- $(d2_s, d2_c)$,
- $(a1_s, a1_c)$,
- $(a1_c, b'_s)$,
- $('b_s, b_c)$,
- (c_s, c'_c),
- $(c'_c, d1_s)$,
- $(d1_s, d1_c)\rangle$,

is shown in Figure 8.4(a). For simplicity, we do not consider any performance classifier in this example.

In the P-proclet (Figure 8.3), each pair of transitions producing a token onto a place, and a transition consuming a token from this place, corresponds to a segment in the process performance spectrum. Each segment occurrence (Definition 3.2) shows transitioning ("moving") between the corresponding steps or steps' lifecycle transitions. For example, transitions $t4$ (label $a2_s$) and $t5$ ($a2_c$) around place $a2$ correspond to segment $(a2_s, a2_c)$. Its occurrences show transitioning from lifecycle *start* to *complete* of process step $a2$.

Vice versa, each segment in the process performance spectrum corresponds to a pair of producing and consuming transitions in the P-proclet. For example, occurrence o_1 of segment $(a2_s, b_s)$ corresponds to transitioning from transition $t5$ (step $a2$ completion) to $t6$ (step b start) via place $a2:b$ in between.

As the P-proclet places and performance spectrum segments correspond to each other, these segments can be ordered according to the possible paths in the P-proclet. For example, segment sequence

- $\langle(a2_s, a2_c)$,
- $(a2_c, b_s)$,
- (b_s, b_c),
- (b_c, c_s),
- (c_s, c_c),
- $(c_c, d2_s)$,
- $(d2_s, d2_c)\rangle$

is ordered according to the path from $a2$ to $d2$, while the remaining segments partially describe the path from $d1$ to $a1$. They describe it only partially because segment (b_c, c_s) is common for both and sorted to show the former.

Although the process performance spectrum, computed from event log L^{ET}_{pid}, describes the process performance, it has the following drawbacks.

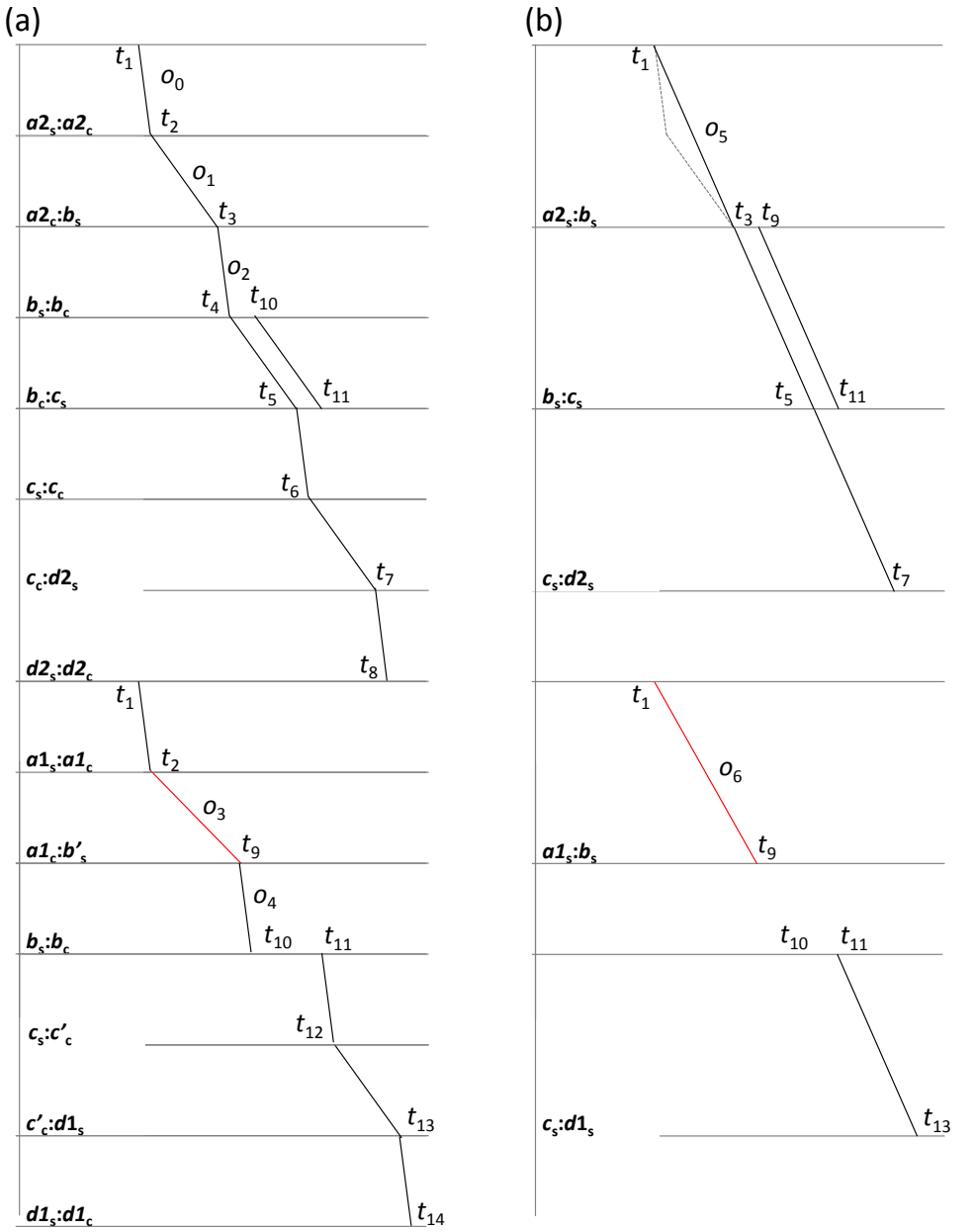


Figure 8.4: Performance spectrum (a), and the spectrum computed from a start-only event log (b).

Event	Activity	Time	pid	qid	rid
$e1$	$a2_s$	t_1	$pid1$	-	$rid2$
$e3$	b_s	t_3	$pid1$	$qid2$	$rid3$
$e5$	c_s	t_5	$pid1$	$qid3$	$rid4$
$e7$	$d2_s$	t_7	$pid1$	$qid5$	$rid6$
$e9$	$a1_s$	t_1	$pid2$	-	$rid1$
$e11$	b_s	t_9	$pid2$	$qid1$	$rid3$
$e13$	c_s	t_{11}	$pid2$	$qid3$	$rid4$
$e15$	$d1_s$	t_{13}	$pid2$	$qid4$	$rid5$

Table 8.2: Start-only process event log obtained from Table 8.1 (in the form of an event table).

- The number of segments is often large because each process step is presented with its *start* and *complete* lifecycle transitions. For example, process step b is presented in five segments of *SEG*.
- Additionally, tags in transition and segment labels increase the segment number, for example, transitions with labels b_s, b_c and b'_c result in two segments (b_s, b_c) and (b_s, b'_c).

That makes it difficult for the analyst to digest this information. Moreover, comparative analysis of segment occurrences when they are in different segments, like (b_s, b_c) and (b_s, b'_c), is time-consuming.

To overcome these drawbacks, we propose a "simplified" process performance spectrum. It is derived from an event table where all events corresponding *complete* transitions are dropped, and activity labels of the remaining events are transformed by removing all the tags. The definition of such a log reads as follows.

Definition 8.1 (Start-only process event log)**.** *Let $ET = (CN, E, \#)$ be an event table (Definition 2.8). Let elem be a function that maps a tuple to its n^{th} element. Let set $E^P \subseteq E$ be the set of all* start *events in E, i.e., $E^P = \{e \mid e \in E, elem_2(\#_{\text{act}}(e)) = start\}$. Let $\#^P$ be an attribute function that maps all event attributes exactly as # but always provides empty tag value ϵ for the tags of activity labels, i.e., $\#^P(e, an) : E^P \times AN \nrightarrow Val =$*

$$\begin{cases} (elem_1(\#_{\text{act}}(e)), elem_2(\#_{\text{act}}(e)), \epsilon) & an = \text{act}, \\ \#_{an}(e) & otherwise. \end{cases}$$

Let $ET^P = (CN, E^P, \#^P)$ be an event table. An event log $L_{\text{pid}}^{ET^P}$ derived for pid *from ET^P according to Definition 2.12 is a* start-only process event log*.*

The start-only process event log, obtained from the event table in Table 8.1, is shown in Table 8.2 in the form of an event table.

The corresponding performance spectrum, built from this log, is shown in Figure 8.4(b). In this spectrum, each segment corresponds to transitioning from the

start of one process step via this step's completion to the start of the next process step, thereby aggregating the *complete* lifecycle transition of the first process step. For example, occurrences o_0 and o_1 in Figure 8.4(a) are aggregated as an occurrence o_5 in Figure 8.4(b). As a result,

- the number of segments is less than in the spectrum in Figure 8.4(a) – five and 13 respectively,
- the number of segment occurrences is less as well– six and 14 respectively.

So, the analyst can read and understand this spectrum much easier. In the remainder of this thesis, we refer to this type of performance spectrum, computed from a start-only event log, as the *process performance spectrum* and write *PS-P* for referring to it.

Next, we consider the performance spectrum of R-proclets.

8.2.3 Relating R-Proclets to Resource Performance Spectra

Each R-proclet in a PQR-system describes the workflow of the corresponding resource, which is modeled through transitions *start* and *complete*, and states *idle* and *busy*, as shown in Figure 8.5(a). A modeled resource (potentially endlessly) switches from state *idle* to state *busy* and back. An event log for resource case notion rid can be obtained from an event table according to Definition 2.12 in the same way as for case notion pid and P-proclets. Respectively, a performance spectrum can be computed for such a log.

Let us consider traces of event log L^{ET}_{rid} for process case notion rid, derived from the event table in Table 8.1:

- $\sigma_{rid1} = \langle e9, e10 \rangle$,
- $\sigma_{rid2} = \langle e1, e2 \rangle$,
- $\sigma_{rid3} = \langle e3, e4, e11, e12 \rangle$,
- $\sigma_{rid4} = \langle e5, e6, e13, e14 \rangle$,
- $\sigma_{rid5} = \langle e15, e16 \rangle$,
- $\sigma_{rid6} = \langle e7, e8 \rangle$.

These traces describe the firing of the resource lifecycle transitions. For example, trace σ_{rid3} shows switching from *start* to *complete*, then back to *start*, and finally to *complete* again. However, if we consider the corresponding activity labels of these events, we get the following sequence

- $\sigma_{rid3} = \langle b_s, b_c, b'_s, b_c \rangle$,

and segments

- (b_s, b_c),
- (b_c, b'_s),

- (b'_s, b_c)

in the corresponding performance spectrum. These segments describe transitioning from *start* to *complete* and back in an unclear way. To prevent it, we introduce a resource event log with activity labels showing the resource lifecycle transition and identifier, thereby allowing better readability of the corresponding resource performance spectrum. Its definition reads as follows.

Definition 8.2 (Resource event log)**.** *Let $ET = (CN, E, \#)$ be an event table (Definition 2.8). Let $\#^R$ be an attribute function that maps each event $e \in E$ and an attributes name $an \in AN$ (see Definition 2.7) to attribute values in Val as follows: $\#^R(e, an) : E \times AN \nrightarrow Val =$*

$$\begin{cases} (elem_2(\#_{\text{act}}(e)), \#_{\text{rid}}(e)) & an = \text{act}, \\ \#_{an}(e) & otherwise. \end{cases}$$

Let $ET^R = (CN, E, \#^R)$ be an event table. An event log $L^{ET^R}_{\text{rid}}$ derived for rid *from ET^R according to Definition 2.12 is a* resource event log*.*

We write activity labels of queue event log events as the R-proclet transition label *start* or *complete* with a superscript showing the resource identifier *rid*. For example, we write label $(start, rid3)$ as $start^{rid3}$.

The resource event log, obtained from the event table in Table 8.1, is shown in Table 8.3 in the form of an event table. In this log, the activity label sequence of σ_{rid3} is

- $\langle start^{rid3}, complete^{rid3}, start^{rid3}, complete^{rid3} \rangle$.

Now, it describes the resource lifecycle clearly.

When we compute the corresponding performance spectrum, it describes the resource performance through the two following segments:

- $(start, complete)$,
- $(complete, start)$.

Each occurrence of the former describes the resource's transitioning from the moment it enters state *busy* till it leaves this state, and corresponds to place *busy*. Each occurrence of the latter describes the resource's transitioning from the moment it enters state *idle* till it leaves this state, and corresponds to place *idle*. The duration of occurrences of $(start, complete)$ corresponds to the resource service time, and the duration of occurrences of $(complete, start)$ corresponds to the resource waiting time. In the following, we call it a *resource performance spectrum* and write *PS-R* for referring to it.

Let us consider a PS-R, computed from a resource event log, containing the only trace σ_{rid3}, for segments $SEG^R = \langle (start^{rid3}, complete^{rid3}) \text{ and } (start^{rid3}, complete^{rid3}) \rangle$. It

Event	Activity	Time	pid	qid	rid
$e9$	$start^{rid1}$	t_1	$pid2$	-	$rid1$
$e10$	$complete^{rid1}$	t_2	$pid2$	$qid1$	$rid1$
$e1$	$start^{rid2}$	t_1	$pid1$	-	$rid2$
$e2$	$complete^{rid2}$	t_2	$pid1$	$qid2$	$rid2$
$e3$	$start^{rid3}$	t_3	$pid1$	$qid2$	$rid3$
$e4$	$complete^{rid3}$	t_4	$pid1$	$qid3$	$rid3$
$e11$	$start^{rid3}$	t_9	$pid2$	$qid1$	$rid3$
$e12$	$complete^{rid3}$	t_{10}	$pid2$	$qid3$	$rid3$
$e5$	$start^{rid4}$	t_5	$pid1$	$qid3$	$rid4$
$e6$	$complete^{rid4}$	t_6	$pid1$	$qid5$	$rid4$
$e13$	$start^{rid4}$	t_{11}	$pid2$	$qid3$	$rid4$
$e14$	$complete^{rid4}$	t_{12}	$pid2$	$qid4$	$rid4$
$e15$	$start^{rid5}$	t_{13}	$pid2$	$qid4$	$rid5$
$e16$	$complete^{rid5}$	t_{14}	$pid2$	-	$rid5$
$e7$	$start^{rid6}$	t_7	$pid1$	$qid5$	$rid6$
$e8$	$complete^{rid6}$	t_8	$pid1$	-	$rid6$

Table 8.3: Resource event log obtained from Table 8.1 (in the form of an event table).

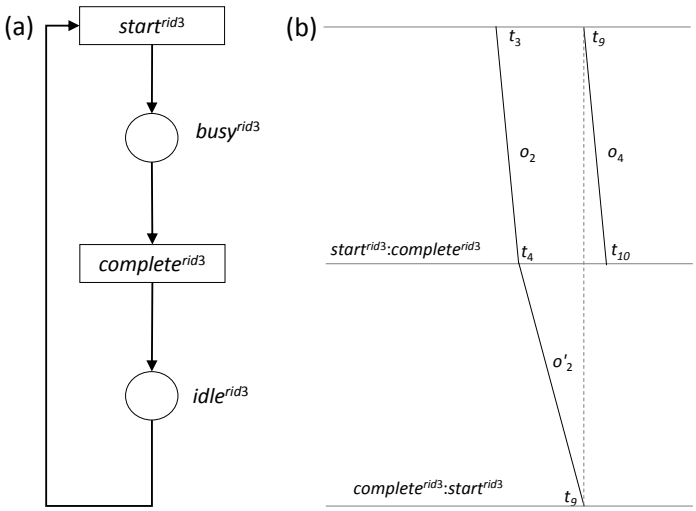


Figure 8.5: R-proclet fragment (a), and resource performance spectrum (b).

is shown in Figure 8.5(b). Again, for simplicity, we do not consider any performance classifier in this example.

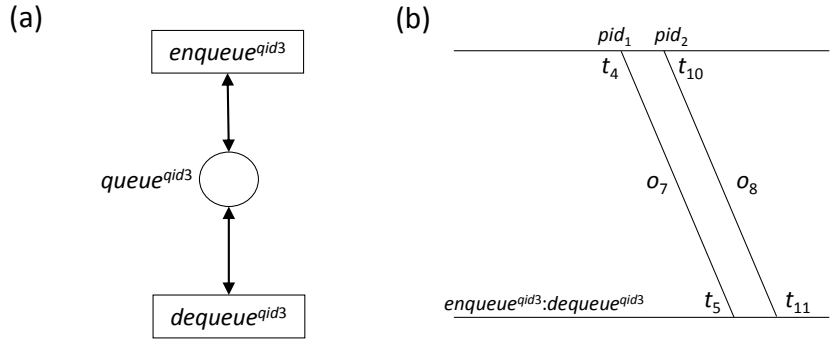


Figure 8.6: Q-proclet fragment (a), and queue performance spectrum (b).

This trace starts at time t_3 when resource *rid*3 entered state *busy* to handle a bag. Occurrence o_2 shows how *rid*3 stayed in this state till time t_4 when it switched to state *idle*. It remained *idle* till time t_9 (o_2') when it switched back to *busy* for handling another bag. Finally, it switched from *busy* to *idle* again at time t_{10}. However, an occurrence corresponding to this period of time when it was *busy* is not presented in this PS-R because it was not fully observed in the event log (i.e.,, its second event is missing). We do not see any performance anomalies in this spectrum, so we still cannot answer the **ExampleAQ**.

In the following, we consider the performance spectra of queues.

8.2.4 Relating Q-Proclets to Queue Performance Spectra

Each Q-proclet in a PQR-system describes the workflow of the corresponding queue, which is modeled through transitions *enqueue* and *dequeue*, as shown in Figure 8.6(a). The elements are modeled as tokens with identifiers of the process (P-proclet). For the queue analysis, we are interested in the actual waiting time of cases in queues. To analyze that using the performance spectrum, we need a segment showing how long each case waits in the queue. To get an event log describing this behavior, we introduce a *queue event log* that has Q-proclet transition labels as activity labels, and pid as a case notion for describing the performance of individual process cases. To distinguish queues, we also add queue identifiers to activity labels. Additionally, we exclude all events irrelevant to queues, i.e., events with undefined queue identifiers *qid*. The queue event log definition read as follows.

Definition 8.3 (Queue event log)**.** *Let $ET = (CN, E, \#)$ be an event table (Definition 2.8). Let set $E^Q \subseteq E$ be the set of all* start *events in E that have defined attribute qid, i.e., $E^Q = \{e \mid e \in E, \#_{\text{qid}}(e) \neq \perp\}$. Let $\#^Q$ be an attribute function that maps each event e in E^Q and an attributes name an in AN (see Definition 2.7) to attribute values in Val as*

Event	Activity	Time	pid	qid	rid
$e10$	enq^{qid1}	t_2	$pid2$	$qid1$	$rid1$
$e11$	deq^{qid1}	t_9	$pid2$	$qid1$	$rid3$
$e2$	enq^{qid2}	t_2	$pid1$	$qid2$	$rid2$
$e3$	deq^{qid2}	t_3	$pid1$	$qid2$	$rid3$
$e4$	enq^{qid3}	t_4	$pid1$	$qid3$	$rid3$
$e12$	enq^{qid3}	t_{10}	$pid2$	$qid3$	$rid3$
$e5$	deq^{qid3}	t_5	$pid1$	$qid3$	$rid4$
$e13$	deq^{qid3}	t_{11}	$pid2$	$qid3$	$rid4$
$e14$	enq^{qid4}	t_{12}	$pid2$	$qid4$	$rid4$
$e15$	deq^{qid4}	t_{13}	$pid2$	$qid4$	$rid5$
$e6$	enq^{qid5}	t_6	$pid1$	$qid5$	$rid4$
$e7$	deq^{qid5}	t_7	$pid1$	$qid5$	$rid6$

Table 8.4: Queue event log obtained from Table 8.1 (in the form of an event table).

follows: $\#^Q(e, an) : E^Q \times AN \not\to Val =$

$$\begin{cases} (enq, \#_{\text{qid}}(e)) & an = \text{act } \textit{and } elem_2(\#_{\text{act}}(e)) = start, \\ (deq, \#_{\text{qid}}(e)) & an = \text{act } \textit{and } elem_2(\#_{\text{act}}(e)) = complete, \\ \#_{an}(e) & otherwise. \end{cases}$$

Let $ET^Q = (CN, E^Q, \#^Q)$ *be an event table. An event log* $L^{ET^Q}_{\text{pid}}$ *derived for* pid *from* ET^Q *according to Definition 2.12 is a* queue event log.

We write activity labels of queue event log events as the Q-proclet transition label *enq* or *deq* with a superscript showing the queue identifier *qid*. For example, we write label $(enq, qid3)$ as enq^{qid3}.

The queue event log, obtained from the event table in Table 8.1, is shown in Table 8.4 in the form of an event table. It does not contain events $e1, e8, e9, e16$ because they have an undefined value of *qid*.

To compute a performance spectrum from a queue event log, we choose segments whose labels are related to the same queue, for the queue event log in Table 8.4 these segments are $SEG^Q =$:

- $\langle (enq^{qid1}, deq^{qid1}),$
- $(enq^{qid2}, deq^{qid2}),$
- $(enq^{qid3}, deq^{qid3}),$
- $(enq^{qid4}, deq^{qid4}),$
- $(enq^{qid5}, deq^{qid5}) \rangle$.

We call a performance spectrum computed from a queue event log for segments related to the same queue a *queue performance spectrum* and write *PS-Q* to refer to it.

The PS-Q for segment (enq^{qid3}, deq^{qid3}), built from the queue event log in Table 8.4, is shown in Figure 8.6(b). In this figure, occurrence o_7 shows that bag *pid*1 was enqueued *qid*3 at time t_4. Later, another bag *pid*2 was enqueued at time t_{10} as well. Finally, these bags were dequeued at time t_5 and t_{11} respectively in the same order as they were enqueued, i.e., they observed the FIFO discipline. They *waited* in this queue for time $t_5 - t_4$ and $t_{11} - t_{10}$ respectively, and they shared this queue from t_{10} till t_5.

For answering the **ExampleAQ**, we combine PS-R and PS-Q according to synchronization described in the PQR-system in the next section.

8.2.5 Combining Resource and Queue Performance Spectra along Synchronization Channels

As we discussed previously, the performance spectra of processes, resources, and queues describe the system performance in the corresponding dimensions. For their analysis, it is crucial to be able to identify segments of the PS-P, PS-Q, and PS-R, related to each other, and analyze them *together*, preferably collocated on a computer screen or paper. To identify and group such segments, we exploit PQR-system synchronization channels and unique identifiers of entities.

Identifying Related Entities. We introduced the PS-P for facilitating the analysis, i.e., for easier detection of zones of interest. So, we choose a PS-P segment (x_s, y_s) as a starting point for grouping the spectra of the other dimensions "around" it. For this segment, we identify all Q- and R-proclets (their queue/resource identifiers) whose transitions synchronize with the P-proclet transitions corresponding to process steps x_s, y_s. Finally, these identifiers are used to identify PS-Q and PS-R segments. For example, if a queue *qid* is identified as a related one, the corresponding PS-Q segment is (enq^{qid}, deq^{qid}). If a resource *rid* is identified as a related one, the corresponding PS-R segments are $(start^{rid}, complete^{rid})$ and $(complete^{rid}, start^{rid})$.

For example, let us consider segment $(a1_s, b'_s)$. The corresponding transitions $t1$ and $t3$ (see Figure 8.7(a)), generating occurrences of $(a1_s, b'_s)$, synchronize with entities of the other (non-process) dimensions as follows:

- $t1$ synchronizes with *R-operators* transition $t12$ (resources *rid*1 and *rid*2),
- $t3$ synchronizes with R-proclet *R-merge-b* transition $t14$ (resource *rid*), and Q-proclet *Q-a1:b'* transition $t11$ (queue *qid*1).

As a result, we can identify one related queue *qid*1, two resources *rid*1, *rid*2 related to the first process step *a* of the given PS-P segment $(a1_s, b'_s)$, and one resource *rid*3 related to the second process step *b*.

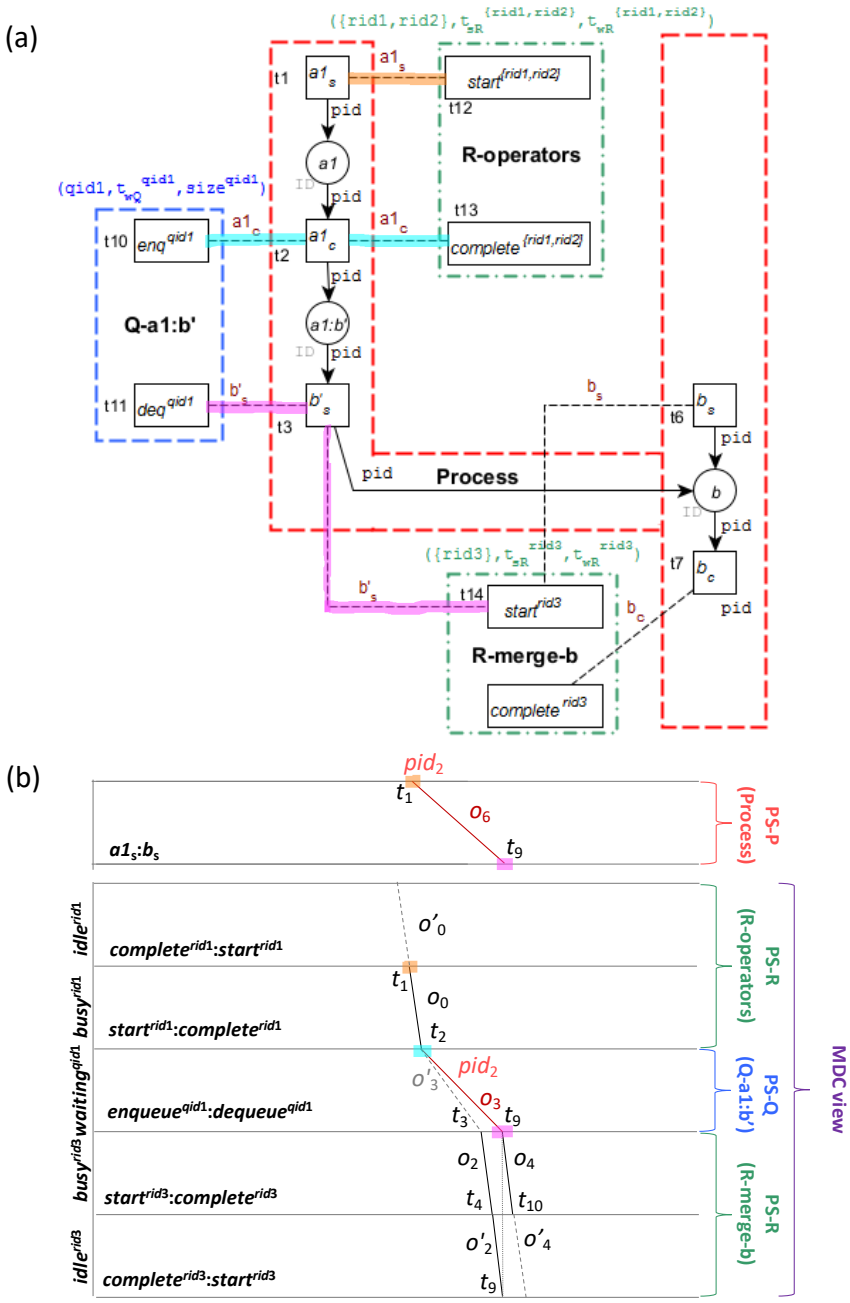


Figure 8.7: PQR-system fragment (a), and PS-P segment with its MDC view (b).

We call such entities *PS-P segment multidimensional context*, its definition reads as follows.

Definition 8.4 (Multidimensional context of a PS-P segment.)**.** *Let S be a PQR-system (see Definition 6.7), let $a, b \in Act$ be activity labels, and $seg^{a,b} = ((a, \text{start}, \epsilon), (b, \text{start}, \epsilon))$ be a PS-P segment. Let $T_{\text{P}}^{a} = \{t \mid t \in T_0, elem_1(\ell_0(t)) = a, elem_2(\ell_0(t)) = start\}$ be the set of the P-proclet transitions that executes the* start *of process steps a, and $T_{\text{P}}^{b} = \{t \mid t \in T_0, elem_1(\ell_0(t)) = b, elem_2(\ell_0(t)) = start\}$ be the set of the P-proclet transitions that executes the* start *of process steps b. Let*

- $\mathscr{I}_{\text{R}}^{a} = \bigcup_{N_i \in \mathscr{N}^R} \bigcup_{t_r \in \{t \mid t \in T_i \text{ such that } \exists t_1 \in T_{\text{P}}^{a}, (t_1,t) \in C\}} elem_2(\ell_i(t_r))$*, and*
- $\mathscr{I}_{\text{R}}^{b} = \bigcup_{N_i \in \mathscr{N}^R} \bigcup_{t_r \in \{t \mid t \in T_i \text{ such that } \exists t_1 \in T_{\text{P}}^{b}, (t_1,t) \in C\}} elem_2(\ell_i(t_r))$

be the non-empty sets of all identifiers of resources whose transitions can synchronize with transitions in T_{P}^{a} and T_{P}^{b} respectively. We call $\mathscr{I}_{\text{R}}^{a}$ and $\mathscr{I}_{\text{R}}^{b}$ top *and* bottom context *respectively. Let $qid^{a,b} \in \mathscr{I}$ be the identifier of a queue related to segment $seg^{a,b}$ such that*

- $\exists N_j \in \mathscr{N}^Q, t_e, t_d \in T_j, \ell_j(t_e) = (enq, \{qid_{a,b}\}, \epsilon), \ell_j(t_d) = (deq, \{qid^{a,b}\}, \epsilon)$*,* $\exists t_3 \in T_0, t_4 \in T_{\text{P}}^{b}$*, and $p_{34} \in P_0$, such that (t_3, p_{34}) and $(p_{34}, t_4) \in F_0$, $(t3, t_e)$ and $(t_4, t_d) \in C$.*

The triple $(\mathscr{I}_{\text{R}}^{a}, \mathscr{I}_{\text{R}}^{b}, qid^{a,b})$ is the Multi-Dimensional Context *(MDC) of* PS-P segment *$seg^{a,b}$.*

In MDC $(\mathscr{I}_{\text{R}}^{a}, \mathscr{I}_{\text{R}}^{b}, qid^{a,b})$, the top and bottom context describe one or multiple resources that synchronize with *start* transitions of process step a and b respectively. Resources of top context enqueue cases in the only queue $qid^{a,b}$ to hand them over to resources of bottom context. We write $MDC(S, seg^{a,b})$ for PQR-system S and PS-P segment $seg^{a,b}$. For the example above, $MDC(S, (a1, b)) = (\{rid1, rid2\}, \{rid3\}, qid1)$.

Grouping Performance Spectra of Related Entities. In MDC (Definition 8.4), the resources of the top and bottom context synchronize with one MDC queue. First, a resource from the top context handles a case and enqueues it to the queue. Then, a resource from the bottom context dequeues this case and handles it subsequently. In general, all combinations of resources from the top and bottom contexts are possible. However, it is easier for the analyst to analyze them separately, grouping the corresponding performance spectrum segment in the execution direction. Because of that, we introduce an MDC view that describes such a combination.

Definition 8.5 (MDC segment series.)**.** *Let $(\mathscr{I}_{\text{R}}^{a}, \mathscr{I}_{\text{R}}^{b}, qid^{a,b})$ be an MDC (Definition 8.4). Let $rid_t \in \mathscr{I}_{\text{R}}^{a}$ and $rid_b \in \mathscr{I}_{\text{R}}^{b}$. The segment series $SEG =$*

- $\langle (complete^{rid_t} start^{rid_t}), (start^{rid_t}, complete^{rid_t}),$
- $(enq^{qid^{a,b}}, deq^{qid^{a,b}}),$
- $(start^{rid_b}, complete^{rid_b}), (complete^{rid_b} start^{rid_b}) \rangle$

is an MDC segment series*.*

An MDC segment series always contains *five* segments. Note, resource segments are ordered in a way that the segments of the top context resource and the segments of the bottom context resource have the opposite sorting order. It allows connecting their lines across these five segments. Given an MDC MDC_i, we write $MDCSeg(MDC_i)$ for its MDC segment series. We call the segments of the PS-R and PS-Q, chosen and arranged according to an MDC segment series, an *MDC view*.

MDC View Example. Let us provide an example of an MDC view for PS-P segment $(a1_s, b_s)$. We choose *rid*1 as a top context resource for an MDC segment series for our example because *rid*1 handled the delayed case *pid*2. The corresponding segment series SEG_1 =

- $\langle(complete^{rid1} start^{rid1}), (start^{rid1}, complete^{rid1}),$
- $(enq^{qid1}, deq^{qid1}),$
- $(start^{rid3}, complete^{rid3}), (complete^{rid3} start^{rid3})\rangle$.

Figure 8.7(b) shows PS-P of $(a1_s, b_s)$, and its MDC view for SEG_1 at the bottom.

In this PS-P, we already detected a slower occurrence o_6 (case *pid*2). Now, in the MDC view, we can see how *pid*2 was handled by resources *rid*1, *rid*3 and queue *qid*1.

1. Initially, resource *rid*1 was *idle* (occurrence o'_0) till time t_1. Note, o'_0 is shown as a dashed line because it is unobserved in the given event table. We provide it for consistency.
2. At t_1, *rid*1 started handling *pid*2 (o_0). The corresponding P-proclet transition $t1$ and R-proclet transition $t12$ synchronize, so occurrences o_6 and o_0 started at the same time t_1 (highlighted in orange in the PQR-system and performance spectrum). Note, o'_0 visually continues as o_0 because the ordering defined in SEG_1 defines $(complete^{rid3}, start^{rid3})$ right after $(complete^{rid1}, start^{rid1})$.
3. At t_2, *rid*2 completed handling and enqueues *pid*2 in queue *qid*1. The corresponding R-proclet transition $t2$ and Q-proclet transition $t10$ synchronize. As a result, the end of o_0 and the beginning of o_3 have the same timestamp (highlighted in cyan), and o_0 visually continues as o_3.
4. At t_9, *rid*3 dequeued *pid*2 from *qid*1 and started handling *pid*2. The corresponding R-proclet transition $t14$ and Q-proclet transition $t11$ synchronize (the end of o_3 and the beginning of o_4 are highlighted in magenta). Again, o_3 and o_4 visually form a continuous line.
5. Finally, *rid*3 became *idle* at t_{10}. The corresponding (unobserved) occurrence o'_4 finishes the line that o'_0, o_0, o_3 and o_4 form.
6. Additionally, occurrences o_2 and o'_2 show how *rid*3 handled another case *pid*1, and line o'_3 show the minimum waiting time of *qid*1.

Let us now explain why o_3 is longer than o'_3. At time t_3, *pid*2 could potentially merge at b, but the corresponding resource *rid*3 was busy handling *pid*1 (o_2). As a result, *pid*2 had to wait in *qid*3 extra time $t_9 - t_3$.

To summarize, an MDC view shows the performance of the resources and queue related to a given PS-P segment. It allows for explaining the performance of cases described in the PS-P. In the next section, we show how various undesirable performance scenarios that impact many cases can be detected in a PS-P, and explained using MDC views.

8.3 Method for Multi-Dimensional Performance Analysis

In this section, we propose the method for answering **RQ-6.2**. The section is organized as follows. We describe a running example in Section 8.3.1. We provide the method overview in Section 8.3.2. We describe six steps of the method in Sections 8.3.3, 8.3.5, 8.3.6, 8.3.7, 8.3.8 and 8.3.9, additionally explaining phenomena of performance pattern instances propagation in Section 8.3.4. Finally, we complete the analysis for the running example in Section 8.3.10.

8.3.1 Running Example

In Section 8.2.1, we introduced a BHS (see Figure 8.2), the corresponding PQR-system (see Figure 8.3), and a simple scenario to explain PS-P, PS-R, and PS-Q. Now, we introduce the following new scenario for the same system, which helps explain the method better.

1. The system started accepting baggage for flights $F1$ and $F2$ at check-in counters $a1$ and $a2$, to move it to exits $d1$ and $d2$ for $F1$ and $F2$ respectively. After exiting the system, bags accumulate near the exits $d1$ and $d2$ to be loaded onto dollies and transported to the aircraft.
2. The system was operating normally till one bag got stuck on conveyor $c : d1$.
3. The system started using the remaining path $c : d2$ to deliver all the baggage to $d2$. Workers started sorting bags, exiting via $d2$, to the dolly corresponding to their flight number.
4. Eventually, space near exit $d2$ became fully occupied with bags (due to the increased load on this exit and the slow speed of manual sorting).
5. The workers at $d2$ could not handle the incoming baggage anymore due to the lack of space and take $rid6$ at $d2$ out of service, thereby effectively stopping conveyor $c : d2$.
6. Gradually, all still operating conveyors stopped as well because they could not hand over their bags to the subsequent conveyors.

7. Over some time, the problematic bag was manually removed from conveyor $c : d1$ and the workers managed to provide free space at $d2$. Resource $d2$ was returned back into service.
8. All the conveyors started working again, but some bags, affected by the delay, did not make it to flight $F1$ that departed earlier than $F2$.

Afterward, the system operator was notified that several bags were checked in at location $a1$ on time, but still could not make it to flight $F1$, i.e., they were delivered at their exit location too late. The operator computed the PS-P (see Figures 8.8(a)) and obtained the MDC views (see Figures 8.8(b)) for the corresponding period. Note, for simplicity, only the MDC view for segment $(c_s, d1_s)$ is shown in Figures 8.8(b).

There are anomalies in these spectra:

- all PS-P segments have some longer occurrences in the context of shorter occurrences,
- all PS-P segments have a higher frequency of occurrences after the longer occurrences, and
- the MDC view segments have some longer occurrences and also periods of absence of any occurrences for the same time period.

The operator's analysis question for this scenario is as follows.

- How to understand why these bags could not make it to the flight, using these spectra, and the PQR-system?

Next, we provide an overview of our method, explaining it through this running example.

8.3.2 Method Overview

In this section, we provide a high-level overview of the method input, output, intermediate results, and steps. The method input is:

1. PQR-system S, and
2. PS-P, PS-Q, and PS-R, i.e., performance spectra built for the process (control-flow), queue, and resource dimensions of S.

The method consists of the six steps shown in Figure 8.9. Let us describe each step in more detail.

Step 1. In the first step, described in Section 8.3.3, all instances of high load and blockage performance patterns are detected. Each instance is described according to a pattern instance definition we introduce in Section 8.3.3, and inserted into a set PI of all detected instances. This set is the result of Step 1.

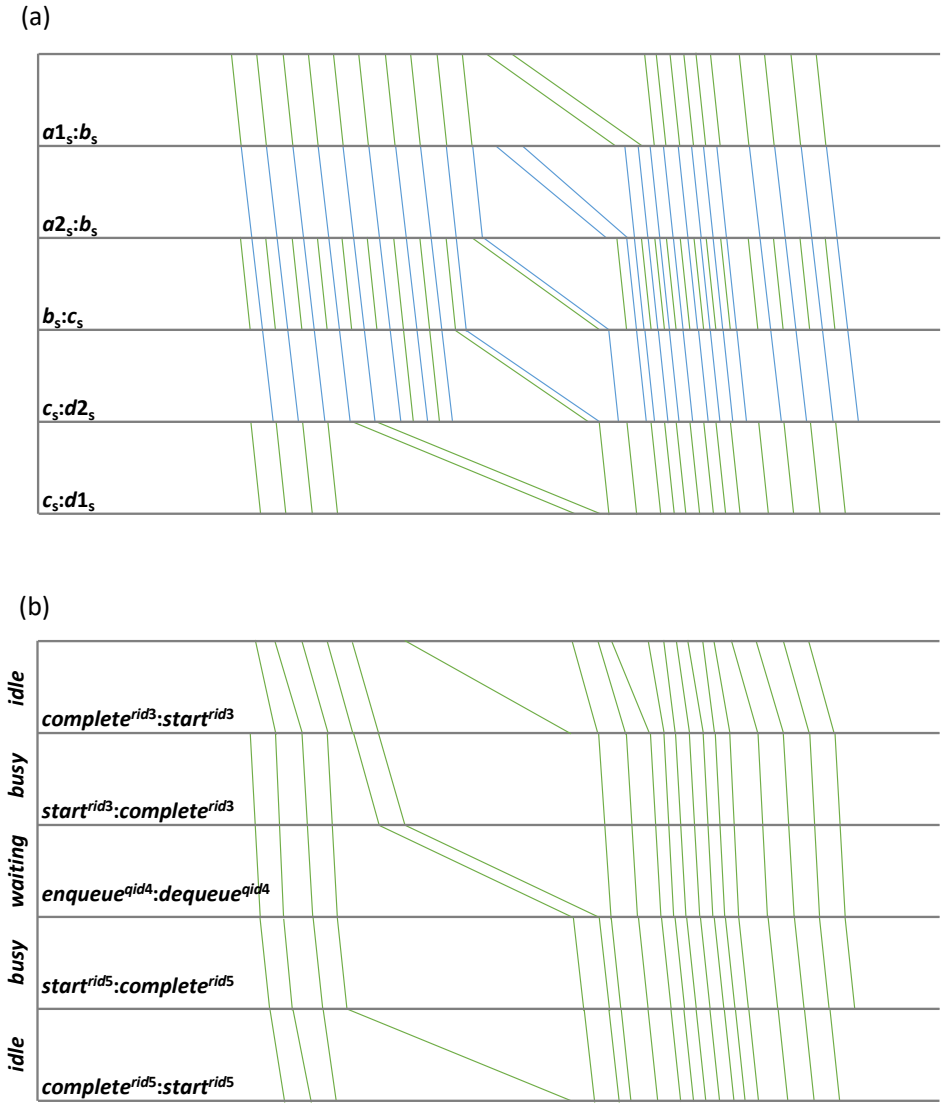


Figure 8.8: PS-P (a), and the MDC view for segment $(c_s, d1_s)$ (b).

Step 2. In the second step, described in Section 8.3.5, dependencies among instances in *PI* are discovered. That is, if one instance z_i in segment seg_i was caused by another instance z_j in the same or other segment seg_j (i.e., it is possible that $seg_i = seg_j$), we say that z_j *propagated* from seg_j to seg_i and *triggered* z_i. We say that two such

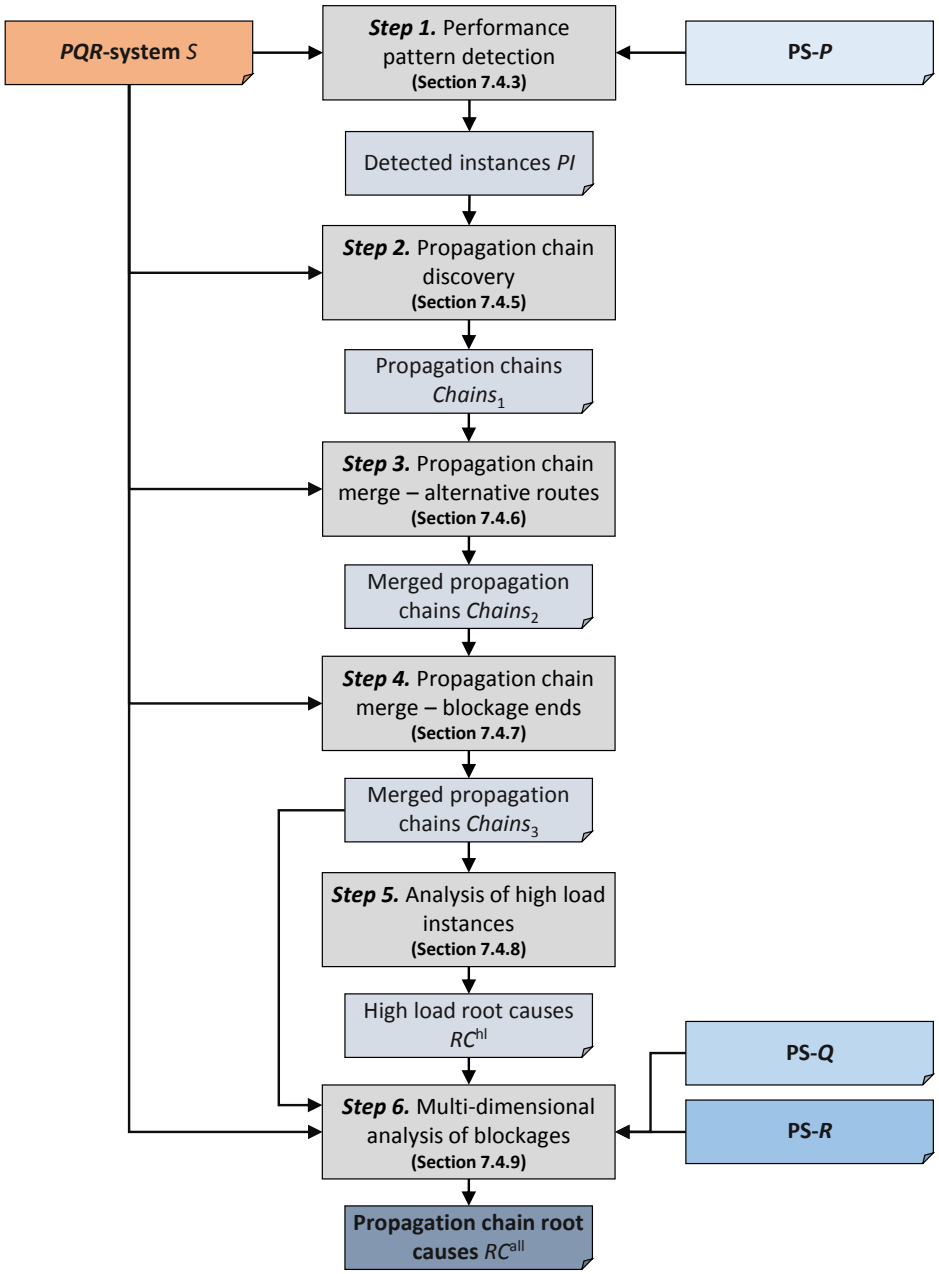


Figure 8.9: Six method steps.

instances form a *propagation link*. Multiple connected propagation links form a *propagation chain* that we define in Section 8.3.4. Propagation chains allow a richer description of the system behavior than individual instances in *PI*. So, they are discovered in this step, using both PS-P and PQR-system, and described according to the definition. Each discovered propagation chain description is inserted into a set $Chains_1$ of initially discovered propagation chains. Note, each instance in *PI* belongs to exactly one propagation chain in $Chains_1$. If an instance was not triggered by any instances in *PI*, and did not trigger any other instances in *PI*, it forms a propagation chain consisting of just this single instance. Set $Chains_1$ is the result of Step 2.

In Step 2, only propagation chains consisting of instances of the same type are discovered. That is, each instance in $Chains_1$ either contains only high load or blockage instances. However, it is possible that blockage instances propagate and trigger high load instances. Steps 3 and 4 consider these situations.

Step 3. In this step, all propagation links corresponding to blockage propagation that triggered high load instances due to *unavailability of alternative paths* are discovered. These links are used further to merge chains in $Chains_1$ into larger ones that describe the behavior more completely than the initial ones.

In P-proclets, multiple paths from one process step to another may exist. If some of them are blocked (i.e., have blockage instances), the load often divides among the remaining unblocked paths. It can cause high load on these paths and, as a result, high load instances in the PS-P. In terms of propagation, we say that a blockage instance propagates and triggers the high load instance(s). If a blockage instance bl_i triggers a high load instance hl_j, they comprise a propagation link $link_{ij} = (bl_i, hl_j)$. If these instances belong to different chains in $Chains_1$, these chains become connected by this link and can be merged into one. In Step 3, the analyst discovers such links and merges the corresponding chains. Our approach for discovering them is described in Section 8.3.6. The result is a set $Chains_2$ containing all merged chains and the chains in $Chains_1$ that were not merged.

Step 4. Similarly to Step 3, chains in $Chains_2$ are merged due to blockage propagation that causes high load instances due to the following reason. During a time period when a blockage instance bl_i is observed in some segment seg_i, the incoming segments of seg_i can accumulate cases (see Chapter 4). As a result, when bl_i ends, all these accumulated cases are handed over to seg_i in a batch. It can cause a load peak, i.e., a high load instance hl_i in seg_i that starts immediately after the end of bl_i. These instances form a propagation link (bl_i, hl_i) that allows the merging of the chains they belong to. Our approach for discovering them is described in Section 8.3.7. The result is a set $Chains_3$ of propagation chains.

After Step 4, set $Chains_3$ contains propagation chains that describe undesirable performance scenarios. However, a chain in $Chains_3$ does not explain *why* the corresponding scenario was initially triggered, i.e., its root causes. The approaches of

final Steps 5 and 6 address this problem for high load and blockage instances respectively. For that, in each chain in $Chains_3$ they determine *initial* instances that were not triggered by any other instances and do their root cause analysis as follows.

Step 5. In Step 5, root causes of initial high load instances are considered. Because Steps 3 and 4 already considered two reasons why high load can be triggered, this step is trivial. We identify the root cause of all initial high load instances in $Chains_3$ as *caused by arrival process*. This step is described in Section 8.3.8. The result is a set RC^{hl} of tuples, where each tuple has a propagation chain in $Chains_3$ as the first element and a set of initial high load instances as the second element.

Step 6. In Step 6, the root causes of initial blockage instances are identified using the analysis of the queue and resource dimensions (see Section 8.3.9). For that, an MDC view (see Section 8.2.5) is obtained for each such instance, and blockage instances are detected in the corresponding PS-Q and PS-R. Information about these newly detected blockage instances is used to determine which resource or queue triggered the initial blockage instance in the PS-P. We identify three root causes that are considered in detail in Section 8.3.9.3. Additionally, we identify situations when blockage instances in the PS-Q and/or PS-R are detected incorrectly. The result of this step is a set RC^{all} of tuples, that contains full information about each propagation chain in $Chains_3$, including high load instances root causes. Thus, each tuple contains:

1. propagation chain $chain_i \in Chains_3$ as the first element,
2. the corresponding set of initial high load instances from RC^{hl},
3. a set RC^{bl} of tuples whose elements describe initial blockage instances and their root causes.

RC^{all} is the final result of the method. It allows explaining how undesirable performance scenarios developed, and what caused them.

Last but not least, we consider a running example throughout Sections 8.3.3-8.3.9, whose final result we provide in Sections 8.3.10.

In the following, we explain each step in more detail.

8.3.3 Step 1. Detecting Undesirable Performance Patterns in the PS-P

In this step, given

1. PQR-system S, and
2. PS-P PS,

the analysts *manually detects* and *describes* all blockage and high load instances in the given PS-P. For a performance pattern instance description, the following information is provided:

- a pattern instance type,
- a pattern instance textual label for referring to the instance,
- segment occurrences comprising the instance.

Definitions of the sets of pattern instance types, labels, and pattern instances read as follows.

Definition 8.6 (Sets of pattern instance types and labels)**.** *PIType* = {*blockage*, *highLoad*} *is the* set of pattern instance types. *PILab is the* set of pattern instance labels.

Definition 8.7 (Pattern instance and set of pattern instances)**.** *Let cn be a case notion in* CN_{PQR} *(see Definition 6.13), ET be an event table (see Definition 2.8), SEG be a segment series (see Definition 3.5),* $\mathbb{C}$ *be a performance classifier (see Definition 3.3), and* $\mathbb{PS}^{\mathrm{lr}}_{L^{ET}_{cn}}((SEG,\mathbb{C}))$ *be a performance spectrum (see Definition 3.7). Let piType* $\in$ *PIType be a pattern instance type, and piLab* $\in$ *PILab be a pattern instance label (see Definition 8.6). Let segment* $(a,b) \in SEG$*. Let* $t_{\mathrm{left}}, t_{\mathrm{right}} \in \mathbb{T}, t_{\mathrm{left}} \leq t_{\mathrm{right}}$ *such that set* $OC = \{(e,e') \mid (e,e') \in occ(a,b,L^{ET}_{cn}), t_{\mathrm{left}} \leq \#_{\mathrm{time}}(e), \#_{\mathrm{time}}(e') \leq t_{\mathrm{right}}\}$ *of occurrences is not empty. A tuple* $(piType, OC, t_{\mathrm{left}}, t_{\mathrm{right}}, piLab)$ *is a* pattern instance.

For a pattern instance z, time interval $[t_{\mathrm{left}}, t_{\mathrm{right}}]$ describes the time interval when it is observed. The start and end timestamps of all segment occurrences in OC are in this interval. We write $piStart(z)$ and $piEnd(z)$ for t_{left} and t_{right} of z respectively. We write $piSeg(z)$ for the segment of z.

An example of a blockage instance is shown in Figure 8.10(a). In this example, slower occurrences $(e_1,e_4),(e_2,e_5)$, and (e_3,e_6) (orange lines in a dashed rectangle) are shown in the context of faster occurrences (blue lines). The instance label bl_1 is shown on the right-hand side of the instance. According to Definition 8.7, the analyst describes this blockage as follows:

- $(blockage, \{(e_1,e_4),(e_2,e_5),(e_3,e_6)\}, bl_1)$.

In the following, we use the pictogram shown in Figure 8.10(b) for drawing blockage instances in performance spectra.

An example of a high load instance is shown in Figure 8.10(c), where more frequent occurrences $(e_1,e_6),\ldots,(e_5,e_{10})$ (orange lines in a red dashed rectangle) are shown in the context of less frequent ones (blue lines). The instance is labeled as hl_1. The analyst describes this instance as follows (Definition 8.7):

- $(highLoad, \{(e_1,e_6),\ldots,(e_5,e_{10})\}, hl_1)$.

In the following, we use the pictogram shown in Figure 8.10(d) for depicting high load instances in performance spectra.

Result. The result of this step is a set PI of all pattern instances detected by the analyst. We call it a *pattern instance set*.

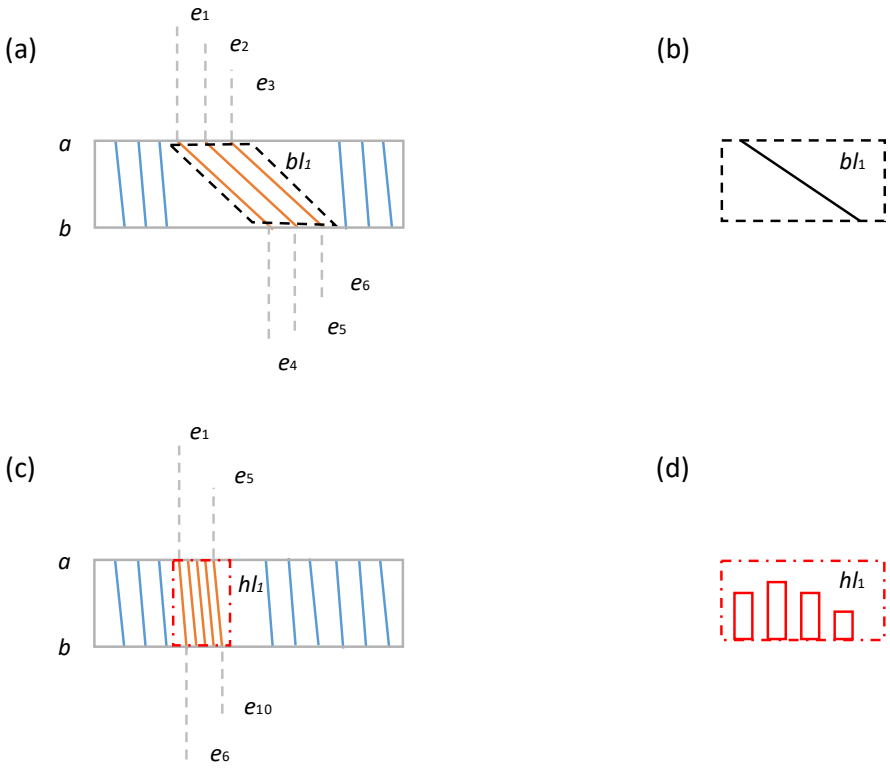


Figure 8.10: Instances and pictograms of blockage (a,b) and high load (c,d) respectively.

For our PS-P example in Figure 8.8(a), the detected blockage and high load instances are shown in Figure 8.11, where each blockage instance is highlighted using a dashed black rectangle, and each high load instance is highlighted using a dashed red rectangle. Each instance has a label. Figure 8.8(b) shows the same pattern instances as pictograms, the lines of occurrences are not shown to avoid clutter. In the following, we use a pictogram-based visualization for the PS-P of this running example.

For this PS-P, pattern instance set $PI =$

- $\{bl_1, \ldots, bl_5$,
- $hl_1, \ldots, hl_5\}$.

It consists of both blockage (bl_i) and high load (hl_j) instances. Note, we use pattern instance labels (see Definition 8.6) for referring to the instances.

Next, we consider how dependencies among the instances in PI can be discovered, using given PQR-system S.

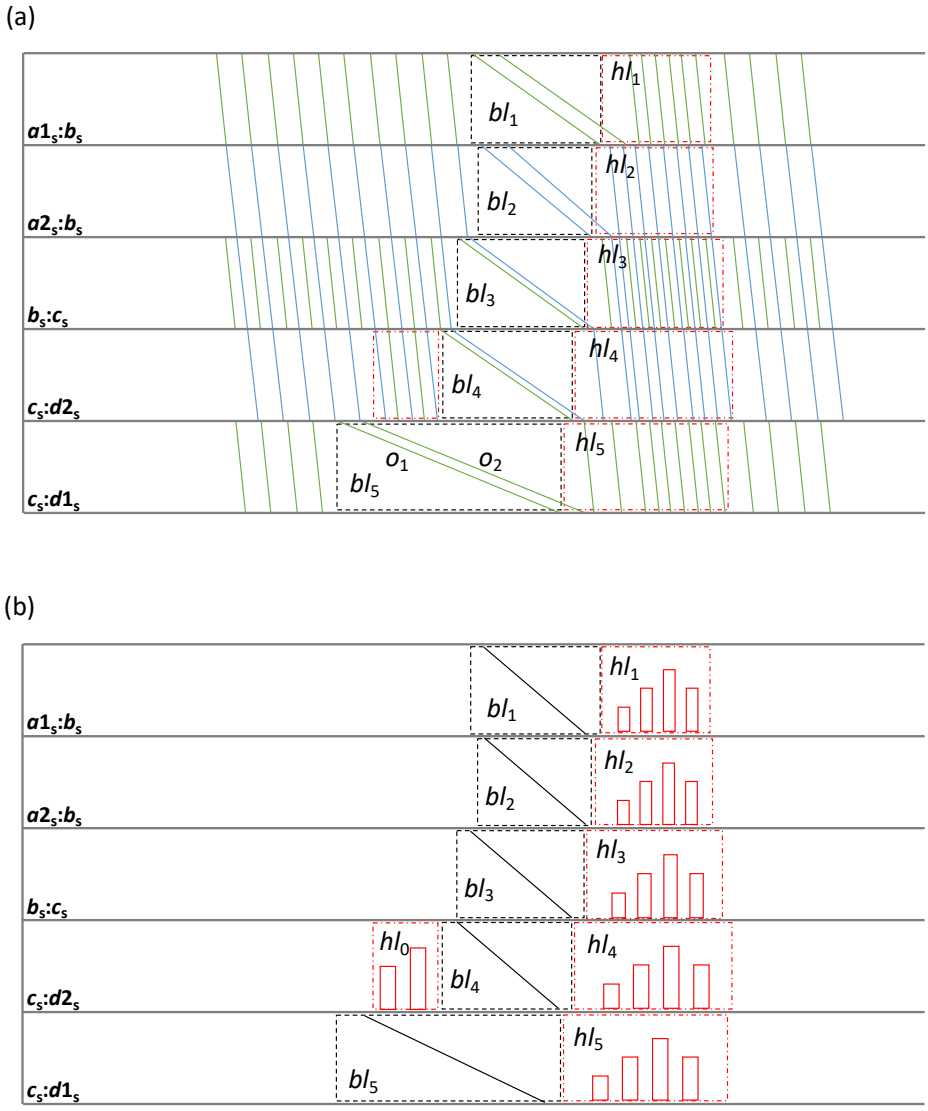


Figure 8.11: Detected pattern instances in the PS-P (a), and as pictograms (b).

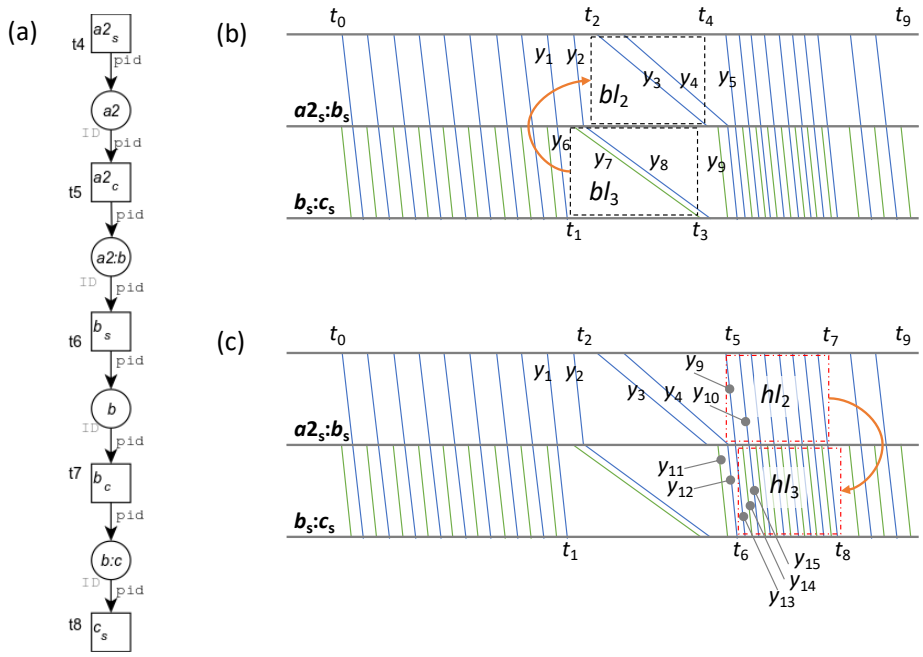


Figure 8.12: P-proclet fragment of the PQR-system in Figure 8.3 (a) and its PS-P with detected pattern instances (b).

8.3.4 Understanding Propagation of Blockage and High Load Instances

In the first step, the analyst detects blockage and high load instances. However, often such instances are not independent but cause each other, i.e., one instance can *propagate* as a blockage or high load instance(s). Now, we consider this phenomenon for both pattern types in more detail.

Blockage Propagation Phenomenon. Let us explain the blockage propagation phenomenon with a simple example. For that, we use a P-proclet fragment of our running example PQR-system, shown in Figure 8.12(a), and the corresponding PS-P, provided in Figure 8.12(b). The P-proclet fragment describes the baggage handling process from process step a_2 till c, and the PS-P describes the performance of baggage handling during time interval $[t_0, t_9]$.

During this time interval, multiple bags were handled. Let us assume that during period $[t_0, t_1]$ the system operated as expected, and the bags traveled at maximum speed. As a result, the corresponding segment occurrences have the shortest possible durations, for example, occurrences y_1, y_2, and y_6. However, at time t_1 conveyor $b:c$ was stopped for some short time period. As a result, bags represented by occurrences

y_7 and y_8 of segment (b_s, c_s) were delayed because they had to wait till conveyor $b:c$ resumed.

At the same time, the preceding conveyor $a2:b$ could work till the bag corresponding to occurrence y_3 had to be handed over onto $b:c$. Since it was stopped, that could not be done. As a result, conveyor $a2:b$ had to stop as well, thereby delaying all its bags (occurrences y_3 and y_4). When $b:c$ resumed, its bags could continue their travel, thereby giving room to bags waiting on $a2:b$ to be handed over to $b:c$. Conveyor $a2:b$ resumed at the same time as well.

However, this situation caused longer travel duration for the affected bags (y_3, y_4, y_7, and y_8). Thus, the analyst can identify blockage instances formed by these delays. In Figure 8.12(b), these instances are shown as blockage pictograms with labels bl_2 and bl_3 over segments $(a2_s, b_s)$ and (b_s, c_s) respectively.

This behavior can be explained using the PQR-system (see Figure 8.3). When queue *Q-b:c* is unavailable for enqueuing, resource *R-merge-b* cannot hand over bags from $a^2:b$ to $b:c$, so queue $a^2:b$ becomes eventually unavailable for enqueuing as well, and resource *rid*2 (*R-operators*), in turn, cannot hand over any bags to *Q-a2:b*. So, the number of queues where groups of bags are delayed gradually increases. We use the term *blockage propagation* to refer to this phenomenon, and say that bl_3 *propagates as* bl_2. When bl_3 propagates as bl_2, it means that one blocked queue eventually blocks another queue, so these blockages overlap in time. For example,

- propagated blockage bl_2 starts not earlier than blockage bl_3 that propagates, i.e., $piStart(bl_2) \in [piStart(bl_3), piEnd(bl_3)]$.

As we discussed, a blockage instance propagates through a resource to the preceding queue. If such a resource is a merge unit, blockage propagates on both incoming queues that this resource serves. In PQR-systems (Definition 6.7), the P-proclet describes this behavior through synchronization of the P-, Q- and R-proclets. So, the P-proclet can be used for understanding possible paths of blockage propagation. However, blockage instances propagate backward in the control flow direction.

For example, in Figure 8.12(b), instance bl_3 is in segment (b_s, c_s) that corresponds to a path between the start transitions t_6 and t_8 of directly following process steps b and c (see the P-proclet in Figure 8.12(a)). At time t_2, bl_3 propagates in segment $(a2_s, b_s)$ because $a2$ is followed by b.

When one pattern instance propagates as a new one, we call this pair a *propagation link*. Its definition reads as follows.

Definition 8.8 (Propagation link)**.** *Let $z, z', z \neq z'$ be pattern instances (see Definition 8.7) such that z propagated as z'. A tuple (z, z') is a* propagation link*.*

For example, blockages bl_3 and bl_2 comprise a propagation link (bl_3, bl_2).

Next, we consider how to discover propagation links in pattern instance set *PI* obtained in Step 1, and what larger structure such links can form.

Ideas behind Blockage Propagation Discovery. As the previous step result, we obtained blockage and high-load instances in pattern instance set *PI*. Now, we want to discover which blockage instances in *PI* are the result of the propagation of the others. For that, we introduce *propagation tracking* reasoning.

Any $z \in PI$ can propagate as other instance(s), or be the result of the other instance(s) propagation. Discovery of the former we call *propagation back-tracking*, and discovery of the latter we call *propagation forward-tracking*. As we discuss earlier, blockage instance propagation paths follow *backward* the control-flow direction described by the P-proclet. As a result,

- propagation forward-tracking is done *backward* in the control-flow direction, as blockage instances propagate over time, and
- propagation chain back-tracking is done in the control-flow direction, to identify the segment(s) where the chain started.

Let us explain how propagation forward and back tracking can be done using our running example (see the PQR-system in Figure 8.3, and the PS-P with detected pattern instances in Figure 8.11(b)).

Let us start by choosing blockage instance bl_3 (segment (b_s, c_s)). First, we track the propagation chain forward, which means backward in the control-flow direction. This blockage instance can potentially propagate on the segments that connect *start* transitions $t3$ or $t6$ (labeled b'_s and b_s) with the other directly preceding *start* transitions. To identify them, we refer to the P-proclet net layout. There are two directly preceding *start* transitions t_1 and t_4 (labeled $a1_s$ and $a2_s$). Note, we skip the *complete* transitions, e.g., t_2 and t_5, as we analyze the PS-P built from events of process step *start* life-cycle transitions.

As the process execution advances from t_1 and t_4 toward t_3 and t_6, these transition labels form the following segments in the PS-P (note that the tag $'$ in b'_s is dropped):

- (a^1_s, b_s),
- (a^2_s, b_s).

For them, we check whether the pattern instance set contains blockage instances that overlap with bl_3 in time. Segment $(a1_s, b'_s)$ has blockage bl_1 such that $piStart(bl_1) \in [piStart(bl_3), piEnd(bl_3)]$. Similarly, segment $(a2_s, b_s)$ has blockage bl_2 such that $piStart(bl_2) \in [piStart(bl_3), piEnd(bl_3)]$, i.e., they overlap in time with bl_3. We add corresponding propagation $links_1 = (bl_3, bl_1)$ and $links_2 = (bl_3, bl_2)$ to a resulting set of propagation links *Links*. These links are shown in Figure 8.13 as arrows with labels $links_1$ and $links_2$ respectively.

Note, instance bl_3 is the first element of $links_1$ and $links_2$ because bl_3 propagates as bl_1 and bl_2. As transitions t_1 and t_4 do not have any incoming arcs in the P-proclet, propagation forward-tracking stops, and propagation back-tracking starts from the initial pattern instance bl_3.

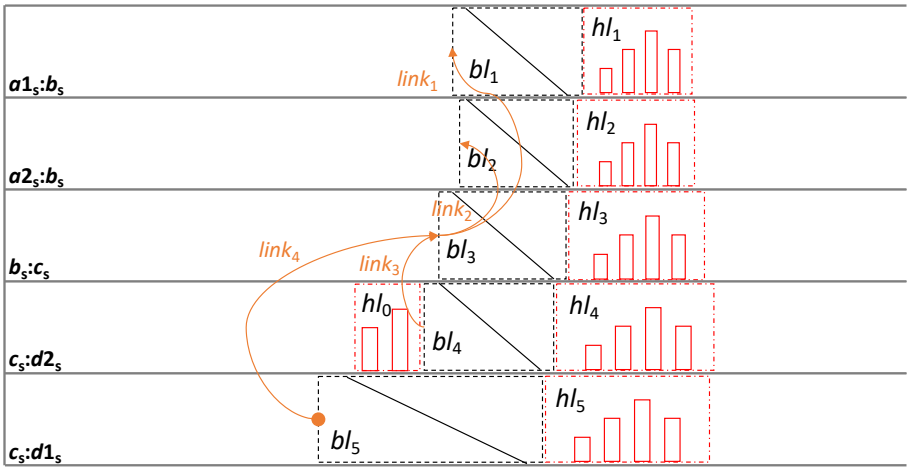


Figure 8.13: Propagation chain of blockage instances.

For that, we identify the *start* transitions that directly follow $t8$ (labeled c_s), i.e., we track propagation forward in the control-flow direction. These transitions are $t23$ ($d1_s$) and $t8$ ($d2_s$), and the corresponding segments are $(c_s, d1_s)$ and $(c_s, d2_s)$. Again, they have blockage instances bl_5 and bl_4 that overlap in time with bl_3, i.e., $piStart(bl_3) \in [piStart(bl_5), piEnd(bl_5)]$ and $piStart(bl_3) \in [piStart(bl_4), piEnd(bl_4)]$. We add propagation links (bl_5, bl_3) and (bl_4, bl_3) to set *Links* (see the corresponding arrows in Figure 8.13). Then propagation back tracking stops because *complete* transitions $t24$ and $t9$ of process steps $d1$ and $d2$ do not have outgoing arcs.

As a result, set *Links* contains propagation links $(bl_3, bl_1), (bl_3, bl_2), (bl_4, bl_3), (bl_5, bl_3)$. Blockage instances in $\{bl_1, \ldots, bl_5\}$ and set *Links* can be seen as nodes and arcs of a directed graph that describes related propagation links, that we call a *propagation chain*. Its definition reads as follows.

Definition 8.9 (Propagation chain)**.** *Let Z be a non-empty set of pattern instances (see Definition 8.7), let Links be a possibly empty set of propagation links (see Definition 8.8) such that*

1. *a pattern instance comprising any link in Links is in Z, i.e., $\forall (z_1, z_2) \in Links, z_1, z_2 \in Z$,*
2. *if Z contains more than one instance, all these instances form links in Links, i.e., if $|Z| > 1, \forall z_1, z_2 \in Z, z_1 \neq z_2$, either $(z_1, z_2) \in Links$ or $(z_2, z_1) \in Links$.*

The directed graph chain $= (Z, Links)$ (see Section 2.1) is a propagation chain *if there is an undirected path between any two nodes in Z, i.e., the corresponding undirected graph $(Z, Links)$ is connected (see Chapter 2).*

Note, if $Links = \emptyset$, Z contains exactly one pattern instance, thereby describing a special case when a pattern instance does not belong to any propagation chain.

A propagation chain shows how involved pattern instances propagated as other instances. Instances corresponding to nodes without incoming arcs are ones that triggered the whole propagation chain, and instances corresponding to nodes without outgoing arcs are ones where propagation ended. For example, in propagation chain *Links,* shown in Figure 8.13,

1. instances bl_1 and bl_2 triggered the whole chain by
2. propagating as bl_3 via links $(bl_4, bl_3), (bl_5, bl_3)$,
3. that consequently propagated as bl_1 and bl_2 via links $(bl_3, bl_1), (bl_3, bl_2)$.

High Load Propagation Phenomenon. Now, let us explain the high load propagation phenomenon, using the same P-proclet fragment in Figure 8.12(a) that we used to explain blockage propagation above, and the same PS-P, provided in Figure 8.12(c). In this example, the system operated normally in time interval $[t_0, t_1]$, then experienced blockages in $[t1, t6]$, and eventually resumed at t_5.

However, during $[t_0, t_1]$, a line of passengers accumulated at check-in counters $a1, a2$. When the system resumed at t_5, the awaiting passengers checked in their baggage one by one as quickly as the counters allowed. As a result, new occurrences in segment $(a2_s, b_s)$ started two times more frequently than before t_1. For example, the time distance between the beginning of y_9 and y_{10} is two times shorter than between y_1 and y_2. So, the load on this segment in period $[t_5, t_7]$ was *higher* than the load on this segment earlier in period $[t_0, t_2]$. The analyst can detect a high load instance hl_2 shown in Figure 8.12(c).

According to the P-proclet in Figure 8.12(a), baggage from $(a2_s, b_s)$ is handed over by resource $rid3$ onto (b_s, c_s). The time distance between the neighboring occurrences preserves on (b_s, c_s). For example, time distance between y_{13} and y_{15} is the same as the time distance between y_9 and y_{10}. The same is true for the other occurrences of hl_2. As a result, this higher load propagated to the next segment. The same scenario for another check-in counter $a1$ made the load even higher. The analyst can detect another high load instance hl_3 (Figure 8.12(c)).

To summarize, high load instances propagate in the control flow direction, and an instance that triggered propagation starts earlier than the instance that was triggered. However, the latter may not hold, for example, if a high load instance started earlier because of a different reason. For example, in Figure 8.12(c), occurrences y_{11} and y_{12} started almost "together" because bl_1 and bl_2 ended, so the analyst may consider them as the start of high load instance hl_3. Note, high load instances going at the same time via a merge unit propagates as an instance with an even higher load (load from conveyor $a1 : b$ does in our example), while a diverting unit can divide the load between its outputs, so the instance can even dissolve.

Ideas behind High Load Propagation Discovery. To track high load propagation, we use the same approach as for blockage propagation discovery. The only difference is in the direction of blockage and high load propagation: they propagate in opposite directions. That is,

- high load propagation forward-tracking is done in the control-flow direction,
- and high load propagation back-tracking is done *backward* in the control-flow direction, to identify the segment(s) where propagation started.

Let us explain how propagation forward and back tracking can be done using our running example (see the PQR-system in Figure 8.3, and the PS-P with the detected pattern instances in Figure 8.11(b)).

The analyst chooses high load instance hl_3 (segment (b_s, c_s)), and starts tracking propagation backward, which means also backward in the control-flow direction. hl_3 can potentially be the result of propagation from segments that connect *start* transitions $t3$ or $t6$ (labeled b'_s and b_s) with the other directly preceding *start* transitions. As in our blockage propagation tracking example, we refer to the P-proclet net layout to identify them. These are transitions t_1 and t_4. Again, we skip *complete* transitions t_2 and t_5, as we analyze the PS-P built from *start* events only. These transition labels form the following segments in the PS-P:

- (a^1_s, b_s), and
- (a^2_s, b_s).

We check whether pattern instance set *PI* contains high load instances in these segments that overlap with hl_3 in time. Segment $(a1_s, b_s)$ has blockage hl_1 such that $piStart(hl_1) \in [piStart(hl_3), piEnd(hl_3)]$. Similarly, segment $(a2_s, b_s)$ has hl_2 such that $piStart(hl_2) \in [piStart(hl_3), piEnd(hl_3)]$, i.e., they overlap in time with hl_3. We add corresponding propagation links $links_5 = (hl_1, hl_3)$ and $links_6 = (hl_2, hl_3)$ to a resulting set of propagation links *Links*. These links are shown in Figure 8.14.

Transitions t_1 and t_4 do not have any incoming arcs in the P-proclet, so propagation back-tracking stops, and propagation forward-tracking starts from the initial pattern instance hl_3.

For that, we identify the next segments in the control-flow direction using the P-proclet, and check if they contain overlapping in time high load segments that are already in *PI*. These segments are $(c_s, d1_s)$ and $(c_s, d2_s)$, and the high load instances are hl_4 and hl_5. We add propagation links $link_7 = (hl_3, hl_4)$ and $link_8 = (hl_3, hl_5)$ to set *Links* (see the corresponding arrows in Figure 8.14). Then propagation forward tracking stops because *complete* transitions $t24$ and $t9$ of process steps $d1$ and $d2$ do not have outgoing arcs.

As a result, set *Links* contains links $link_5 - link_8$, comprising a propagation chain $(\{hl_5, \ldots, hl_8\}, \{link_5, \ldots, link_8\})$ (see Definition 8.9). Note, in Figure 8.14 high load in-

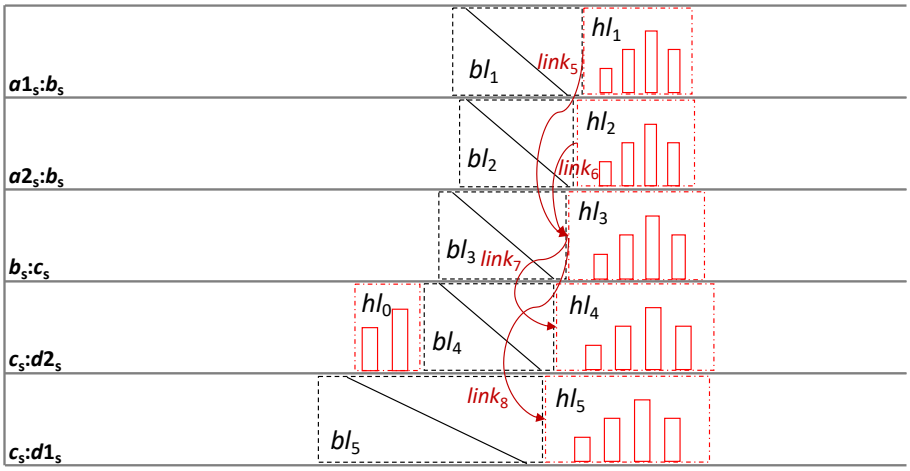


Figure 8.14: Propagation chain of high load instances.

stance hl_0 is not connected with any other instances, it forms its own propagation chain $(\{hl_0\}, \emptyset)$.

Next, we propose a propagation chain discovery algorithm for both blockage and high load propagation.

8.3.5 Step 2. Propagation Chain Discovery

In this step, we use

- given PQR-system S, and
- pattern instance set PI, obtained in Step 1,

to discover blockage propagation chains comprised of blockage instances in PI. The analyst also introduces an empty set *Chains* to add discovered propagation chains.

To discover propagation chains, we iteratively consider each instance in PI, and discover a propagation chain that contains it. All already considered instances, as well as instances already included in propagation chains, are excluded from the further analysis by marking them as *visited* in PI. Iterating stops when all instances in PI are marked as visited. For each iteration, our propagation chain discovery method takes as input

1. given PQR-system S, and
2. pattern instance set PI containing a non-zero number of unvisited instances.

At the start of each iteration, the analyst introduces four empty sets:

1. the *set of unvisited pattern instances notVisitedPI*, where the instances belonging to the current propagation chain are "waiting" for the analysis of their possible propagation to/from the neighboring segment(s),
2. the set of already visited segments *visitedSegments*,
3. the set *Links* for storing discovered propagation links, and
4. the set Z for storing pattern instances comprising propagation links in *Links*.

To start an iteration, the analyst

1. selects a pattern instance $z_0 \in PI$ such that z_0 is not marked as *visited* in PI,
2. inserts z_0 into *notVisitedPI*,
3. marks z_0 as *visited* in PI, and
4. inserts segment $seg = piSeg(z_0)$ in *visitedSegments*.

Then, the analyst iterates over the elements of *notVisitedPI*. On each iteration, the following sub-steps are executed for the current instance $z_i \in notVisitedPI$.

- **Sub-Step D1.** Tracking propagation in the control-flow direction.
- **Sub-Step D2.** Tracking propagation backward in the control-flow direction.
- **Sub-Step D3.** Removing z_i from *notVisitedPI* and checking whether iterating must stop.

In sub-steps D1 and D2,

- new instances are added to *notVisitedPI* for the consequent propagation tracking, and
- newly discovered propagation links and corresponding instances are added to the resulting sets *Links* and Z.

When iterating stops, a directed graph $(Z, link)$ is the resulting propagation chain. Note, the type of the initial pattern instance z_0 defines whether blockage or high load propagation is discovered, i.e., if $piType(z_0) = blockage$, a blockage propagation chain is discovered, and a high load propagation chain is discovered otherwise. In the following, we describe sub-steps D1-D3 in more detail.

Sub-Step D1. Tracking Propagation in the Control-Flow Direction. As we discussed earlier, high load instances propagate to the neighboring segments in the control-flow direction, while blockage instances propagate in the opposite direction. So, in the control-flow direction, we do forward tracking of high load instances propagation and back tracking of blockage instances propagation.

Let $z_i \in notVisitedPI$ be the instance we are currently tracking. For tracking propagation of z_i in the control-flow direction, its directly following segments in this direction have to be determined. For that, a transition t_2, corresponding to the second process step of instance z_i segment, is taken as a starting point. Then, it is used to identify all segments whose first process step label is $\ell(t_2)$, and the second one is a

label of a *start* transition of a directly following process step. That is, these segments $SEG_{df} = \{(\ell(t_2), \ell(t)) \mid t \in T, \ell(t) = (a, start, tag), a \in Act, tag \in Tags, \exists p_1, p_2 \in P, p_1 \neq p_2, t' \in T$ such that $(t_2, p_1), (p_1, t'), (t', p_2), (p_2, t) \in F$ and $(\ell(t_2), \ell(t)) \notin visitedSegments\}$.

Then, each segment $seg_j \in SEG_{df}$ is analyzed for the presence of a pattern instance z_j such that it:

1. has the same type (i.e., blockage or high load) as instance z_i, i.e., $piType(z_i) = piType(z_j)$,
2. overlaps with z_i such that the following holds:
    - if we track blockage propagation, z_i ends not earlier than z_j starts and not later than z_j ends, i.e., $piEnd(z_i) \in [piStart(z_j), piEnd(z_j)]$ if $piType(z_j) = blockage$, or
    - if we track high load propagation, z_j starts not earlier than z_i starts and not later than z_i ends, i.e., $piStart(z_j) \in [piStart(z_i), piEnd(z_i)]$ if $piType(z_i) = highLoad$.

When z_j satisfies these conditions, the analyst:

- adds a propagation link (z_j, z_i) to *Links* if the type of z_j is *blockage*, and adds a link (z_i, z_j) otherwise,
- marks z_j as *visited* in *PI*,
- adds z_j to *notVisitedPI*, and
- adds seg_j to *visitedSegments*.

Sub-Step D2. Tracking Propagation Backward in the Control-Flow Direction. This step is similar to the previous one but has the following differences.

1. Considering $z_i \in notVisitedPI$ be the instance we are currently tracking, all its directly *preceding* segments are considered for tracking instead of directly following. For that, a transition t_1, corresponding to the *fist* process step of instance z_i segment, is taken as a starting point. Then, it is used to identify all segments whose *second* process step label is $\ell(t_1)$, and the *first* one is a label of a *start* transition of a directly preceding process step. That is, these segments $SEG_{dp} = \{(\ell(t), \ell(t_1)) \mid t \in T, \ell(t) = (a, start, tag), a \in Act, tag \in Tags, \exists p_1, p_2 \in P, p_1 \neq p_2, t' \in T$ such that $(t, p_1), (p_1, t'), (t', p_2), (p_2, t_1) \in F$ and $(\ell(t), \ell(t_1)) \notin visitedSegments\}$.
2. overlapping in time is checked differently:
    - if we track blockage propagation, z_j starts not earlier than z_i starts and not later than z_i ends, i.e., $piStart(z_j) \in [piStart(z_i), piEnd(z_i)]$ if $piType(z_i) = blockage$, or
    - if we track high load propagation, z_i ends not earlier than z_j starts and not later than z_j ends, i.e., $piEnd(z_i) \in [piStart(z_j), piEnd(z_j)]$ if $piType(z_j) = highLoad$.

3. If instance z_j is found, the corresponding propagation links are created in the opposite directions, i.e., the analyst adds a propagation link (z_i, z_j) to *Links* if the type of z_j is *blockage*, and adds a link (z_j, z_i) otherwise.

Sub-Step D3. Excluding Current Pattern Instance from Further Tracking. This step is trivial, z_i is removed from *notVisitedPI*. Discovery ends when *notVisitedPI* is empty, otherwise, the next iteration (Sub-Step D1) starts.

When the discovery phase completes, set *Links* contains all the discovered propagation links, and set Z contains pattern instances included in the chain, thereby forming propagation chain $chain = (Z, Links)$ (see Definition 8.9). It is added to the resulting set *Chains*.

8.3.6 Step 3. Merging Propagation Chains Due To High Load Propagation to Alternative Routes

Typically, multiple routes to the same destination or area exist in an MHS in order to provide:

- *high availability* for avoiding single points of failure, and
- *load balancing* (see Section 1.1.2).

When one or multiple alternative routes are unavailable, bags follow the routes remaining available. As a result, the load on the segments of these routes becomes higher, and high load instances can emerge.

For example, the system in Figure 8.3 has two routes from location c to the baggage off-loading area at locations $d1$ and $d2$ (conveyors $(c, d1)$ and $(c, d2)$ respectively). When one of them is unavailable, the baggage follows through the other. The PS-P in Figure 8.15 shows that segment $(c_s, d1_s)$ is unavailable during blockage instance bl_5. As a result, the baggage goes through another still available segment $(c, d2_s)$, thereby causing high load instance hl_0. We say that bl_5 propagates to alternative segment (route) $(c_s : d2_s)$ as a high load instance hl_0. As a result, a propagation link $link_0 = (bl_5, hl_0)$ is identified. Note, in this case, the high load instance must start within the time interval when the blockage instance that triggered it is observed.

When a blockage instance propagates as a high load one, it connects two chains: one that contains the blockage instance, and one that contains the high load instance. As a result, these chains can be merged into one to show a more complete propagation chain. For example, after propagation link $link_0$ is identified, the corresponding propagation chains

- $chain_0 = (\{hl_0\}, \emptyset)$,
- $chain_1 = (\{bl_1, \ldots, bl_5\}, \{link_1, \ldots, link_4\})$,

can be merged into a new chain

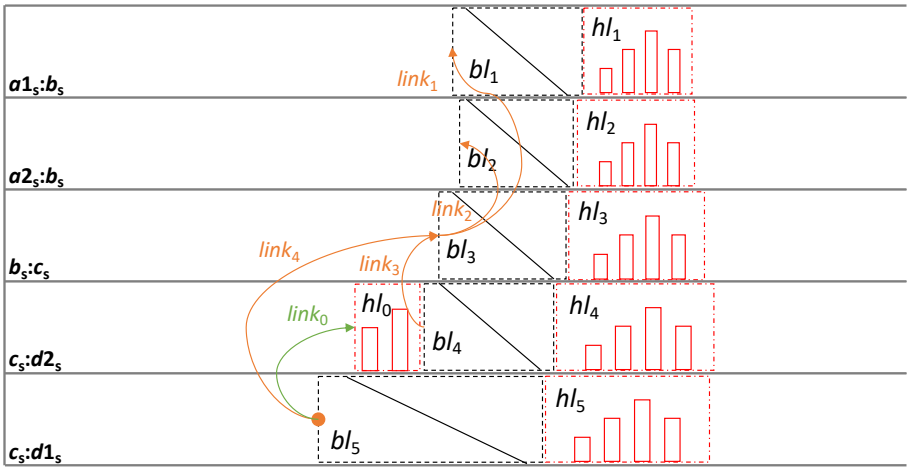


Figure 8.15: Propagation link $link_0$ shows that bl_5 propagated as high load instance hl_0.

- $chain_{01} = (\{bl_1, \ldots, bl_5, hl_0\}, \{link_0, \ldots, link_4\})$.

The propagation chain discovery approach, which we considered earlier, does not discover propagation of blockage instances as high load ones. To discover more complete propagation chains, showing how high load is triggered by blockages, we formulate the following sub-problem.

- Given the PQR-system, PS-P and the set of discovered chains *Chains*, how to merge chains if blockage instance(s) of one propagated as high load instance(s) of another?

Approach for Discovery of High Load Instance Triggered by Blocked Alternative Routes.

To solve this problem, we propose the following approach.

1. For each high load propagation chain in *Chains*, we identify initial high load instances that are not triggered by the other high load instances of the same chain.
2. For each initial instance, we look for a propagation chain, whose blockage instance(s) triggered the current initial instance. If such a chain is found, we create the corresponding propagation link(s) and merge these two chains into one.

Note that each blockage instance is visited only once, but while searching for a related chain, all the other chains are considered. This is required to be able to merge multiple chains. Let us describe these steps in more detail.

1. At the beginning, we create a set $Chains_{\text{hl}}$ of all high load propagation chains in $Chains$, $Chains_{\text{hl}} \subseteq Chains$, and a set $Chains_{\text{blm}}$ of all blockage propagation chains in $Chains$, $Chains_{\text{blm}} \subseteq Chains, Chains_{\text{hl}} \cap Chains_{\text{blm}} = \emptyset$.
2. We iterate over each chain $chain_i = (Z_i, Links_i) \in Chains_{\text{hl}}$.
3. In $chain_i$, all initial pattern instances, i.e., the instances that are not the result of propagation of the other instances in Z_i, are identified:
   - $Z_{\text{init}} = \{z_j \mid z_j \in Z_i\}$ such that $\forall z_k \in Z_i, z_k \neq z_j, (z_k, z_j) \notin Links_i$.
4. For each initial high load instance $z_{\text{init}} \in Z_{\text{init}}$, the following steps are performed to identify propagation chains in $Chains_{\text{blm}}$ that triggered z_{init}:
   (a) for segment $((a, start, tag_a), (b, start, tag_b)) = piSeg(z_{\text{init}})$, alternative segments are determined as follows:
      - $SEG_{\text{alt}} = \{((a, start, tag_a), (x, start, tag_x)) \mid (x, start, tag_x) \neq (b, start, tag_b), \exists t, t' \in T, \ell(t) = (a, start, tag_a), \ell(t') = (x, start, tag_x), \exists p_1, p_2 \in P, p_1 \neq p_2, t'' \in T$ such that $(t, p_1), (p_1, t''), (t'', p_2), (p_2, t') \in F\}$.

   (b) For each chain $chain_j = (Z_j, Links_j) \in Chains_{\text{blm}}$, a set of propagation links from blockage instances of $chain_j$, overlapping in time with z_{init}, to high load instances of $chain_i$ on segments SEG_{alt} are identified:
      - $Links_{j,\text{init}} = \{(z_j, z_{\text{init}}) \mid z_j \in Z_j, piType(z_j) = blockage, piSeg(z_j) \in SEG_{\text{alt}}, piStart(z_{\text{init}}) \in [piStart(z_j), piEnd(z_j)]\}$.

      If $Links_{j,\text{init}} \neq \emptyset$,
      i. $chain_j$ is removed from set $Chains_{\text{blm}}$,
      ii. chains $chain_j$ and $chain_i$ are merged into a new chain $chain_{j,\text{init}} = (Z_j \cup Z_i, Links_j \cup Links_i \cup Links_{j,\text{init}})$,
      iii. chain $chain_{j,\text{init}}$ is inserted in $Chains_{\text{blm}}$,

      else $chain_i$ is inserted in $Chains_{\text{blm}}$.

As a result, set $Chains_{\text{blm}}$ contains both chains that were merged, and the chain remains intact, i.e., high load propagation chains whose initial instances were not triggered by blockage instances on alternative routes.

8.3.7 Step 4. Merging Propagation Chains Due To High Load After Blockage Completion

Earlier, we considered how a high load instance can be triggered by the propagation of another high load instance or a blockage from an alternative route. Now, we consider how load increases right after a blockage instance ends on the same segment. That happens when cases (bags) were accumulating on the preceding segment(s) during a

blockage and then were handed over to the segments that just recovered. As a result, the load becomes higher than usual, and a high load instance can emerge.

For example, in Figure 8.12(c), segment occurrences y_{11} and y_{12} have a shorter time distance than the segment occurrences in context before t_1, and can be considered as a high load instance, or as a part of hl_3. However, this higher load was triggered by blockage instance bl_3 (Figure 8.12(b)) that resumed right before the high load on this segment began.

Similarly to the situation with blockage instances propagating to alternative segments, blockage instances that propagated as high load instances can potentially connect multiple propagation chains. Such chains can be merged to describe propagation more accurately. For example, chain

- $chain_{01} = (\{bl_1, \ldots, bl_5, hl_0\}, \{(bl_3, bl_1), (bl_3, bl_2), (bl_4, bl_3), (bl_5, bl_3), (bl_5, hl_0)\})$,

shown in Figure 8.15, can be merged with chain

- $chain_2 = (\{hl_1, \ldots, hl_5\}, \{(hl_1, hl_3), (hl_2, hl_3), (hl_3, hl_4), (hl_4, bl_5)\})$,

shown in Figure 8.13, via propagation links $(bl_1, hl_1), (bl_2, hl_2), (bl_3, hl_3), (bl_4, hl_4), (bl_5, hl_5)$, resulting in chain

- $chain_{012} = (\{bl_1, \ldots, bl_5, hl_0, \ldots, hl_5\},$
- $\{(bl_3, bl_1), (bl_3, bl_2), (bl_4, bl_3), (bl_5, bl_3), (bl_5, hl_0),$
- $(hl_1, hl_3), (hl_2, hl_3), (hl_3, hl_4), (hl_4, bl_5),$
- $(bl_1, hl_1), (bl_2, hl_2), (bl_3, hl_3), (bl_4, hl_4), (bl_5, hl_5)\})$,

shown in Figure 8.16.

Approach for Discovery of High Load Instance Triggered by Ended Blockage Instances. To solve this problem, we propose the following approach similar to the approach of the previous section. The differences are as follows.

1. Identification of segments SEG_{alt} is not needed because in this case a blockage instance always propagates from the segment of the current high load instance.
2. Propagation links $Links_{j,\text{init}}$ are identified within the segment of z_{init}, and the instances do not overlap but the high load instance follows the blockage one, i.e., $Links_{j,\text{init}} =$
    - $\{(z_j, z_{\text{init}}) \mid z_j \in Z_j, piType(z_j) = blockage, piSeg(z_j) = piSeg(z_{\text{init}}), piStart(z_{\text{init}}) = piEnd(z_j)\}$.

8.3.8 Step 5. Analysis of High Load Instances

In this step, the analyst determines all *initial* high load instances, i.e., instances that are not triggered by the other blockage or high load instances, and identifies their root causes as follows.

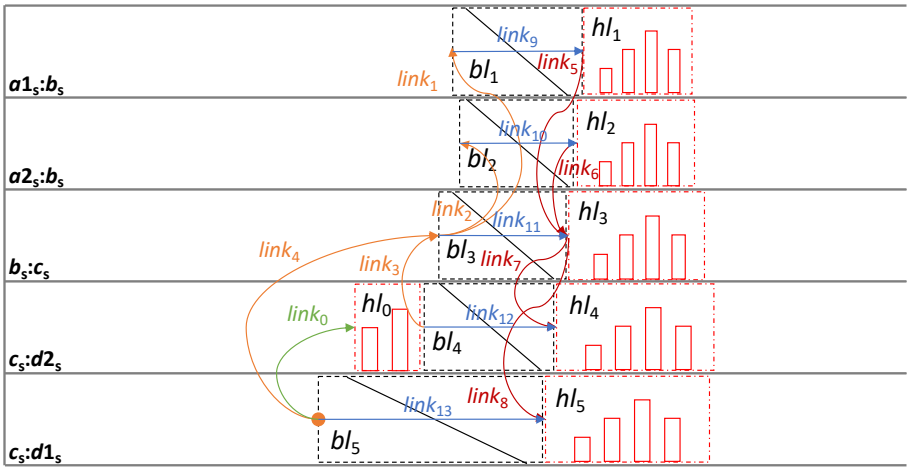


Figure 8.16: Merging propagation chain due to ended blockage instances propagated as high load instances.

1. The analyst introduces an empty set RC^{hl} of initial high load instances.
2. Each $chain_i = (Z_i, Links_i) \in Chains_3$ is considered.
3. In $chain_i$, initial high load instances are the set $Z_i^{\text{hl}} = \{z_i \mid z_i \in Z_i, piType(z_i) = highLoad$, such that $\forall (z_j, z_k) \in Links_i$ holds that $k \neq i\}$.
4. The analyst identifies the root cause of these instances as *triggered by arrival process*. That is, we conclude that if a high load instance is not triggered by any other high load or blockage instances in a chain, it is caused by the system arrival process.
5. The analyst inserts a tuple $(chain_i, Z_i^{\text{hl}})$ into RC^{hl}.

As a result, set RC^{hl} contains such a tuple per each propagation chain in $Chains_3$ that has high load instances triggered by the arrival process.

In our running example, $chain_{012}$ does not contain any initial high load instances, so $RC^{\text{hl}} = \emptyset$.

In the next section, we consider blockage analysis using MDC views.

8.3.9 Step 6. Multi-Dimensional Analysis of Blockage Instances

In Step 5, we identified the root causes of initial high load instances for chains in $Chains_3$. In this step, we do the same but for initial blockage instances, and consolidate the root causes of high load and blockage initial instances as follows.

1. The analyst introduces the empty set RC^{all} of final results.
2. The analyst iterates over chains in $Chains_3$,

3. In the current $chain_i$, the analyst determines initial high load instances as a set $Z_i^{\text{bl}} = \{z_i \mid z_i \in Z_i, piType(z_i) = blockage$, such that $\forall (z_j, z_k) \in Links_i$ holds that $k \neq i\}$.
4. For each initial blockage instance $z_i \in Z_i^{\text{bl}}$, the analyst obtains an MDC view and identifies what queue or resource $id_i \in \mathscr{I}$ triggered z_i.
5. As a result, a tuple (z_i, id_i) describes the root cause of z_i.
6. After all initial blockage instances in Z_i^{bl} are analyzed, a set of all tuples $RC^{\text{bl}} = \{\ldots, (z_i, id_i), \ldots\}$ describes the root causes of all blockage instances of $chain_i$.
7. To complete the analysis of $chain_i$, the analyst merges the results of its high load initial instances analysis from RC^{hl}, i.e., the set of initial high load instances Z_i^{hl} for chain $chain_i$, into a new tuple $(chain_i, Z_i^{\text{hl}}, RC^{\text{bl}})$ and inserts it into RC^{all}.

As a result, set RC^{all} contains tuples describing root causes of initial high load and blockage instances for each chain in $Chains_3$.

We split the step of analysis of blockage instances z_i using MDC views in multiple sub-steps.

- **Sub-Step M1. Determining an MDC view for Analysis.** Given blockage instance z_i in the PS-P, the analyst determines an MDC segment series (see Definition 8.4) for this blockage instance segment, and obtains the corresponding MDC view (see Section 8.2.5).
- **Sub-Step M2. Detecting Blockage Instances in the MDC View.** The analyst detects all blockage instances in the PS-R and PS-Q of the MDC view, high load instance detection is not required.
- **Sub-Step M3. Identifying Root Causes.** The analyst identifies possible RCs depending on the combination of detected blockage instances in the PS-R and PS-Q segments.

In the following, we describe these sub-steps in more detail.

8.3.9.1 Sub-Step M1. Determining and Obtaining PS-Q and PS-R Segments for Analysis.

Given blockage instance z_i, and its segment $((a, start, \epsilon), (b, start, \epsilon)) = piSeg(z_i)$, the analyst determines

- its MDC according to Definition 8.4, i.e., $MDC_i = (\mathscr{I}_{\text{R}}^a, \mathscr{I}_{\text{R}}^b, qid^{a,b}) = MDC(S, (a, b))$,
- the segment series of MDC_i according to Definition 8.5, i.e., $SEG^{a,b} = MDCSeg(MDC_i)$.

Afterward, $SEG^{a,b}$ is used to choose the corresponding segment performance spectra in the given PS-R and PS-Q, and arrange them in a top-bottom order according to the ordering in $SEG^{a,b}$. As a result, the corresponding MDC view is obtained.

In our running example, to analyze blockage bl_5,

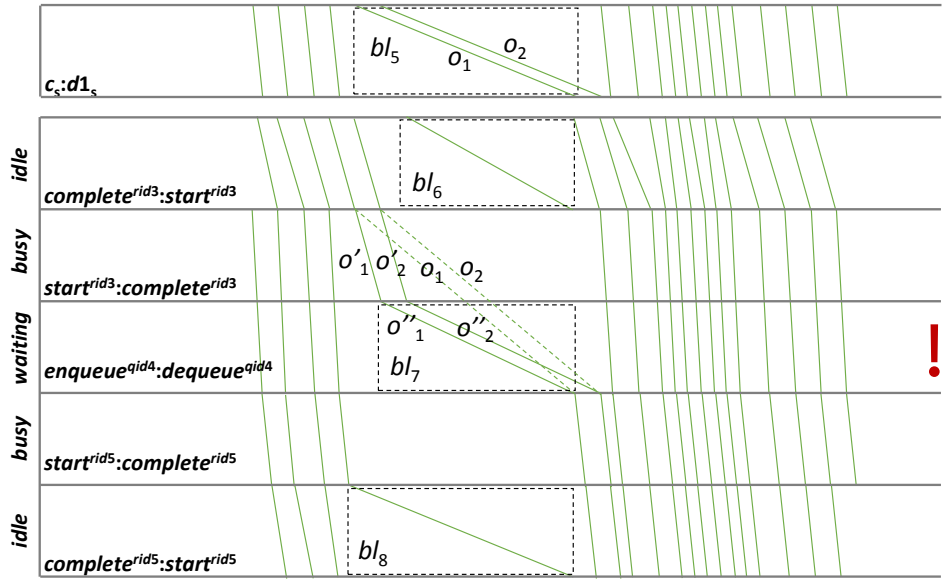


Figure 8.17: MDC view for PS-P segment $(c_s, d1_s)$ showing Variant 1 of Table 8.5.

- $MDC_{bl_5} = (\{rid3\}, \{rid4\}, qid4)$, and
- the only possible MDC segment series $SEG_{bl_5} =$
  - $\langle (complete^{rid3}\, start^{rid3}), (start^{rid3}, complete^{rid3}),$
  - $(enq^{qid4}, deq^{qid4}),$
  - $(start^{rid4}, complete^{rid4}), (complete^{rid4}\, start^{rid4}) \rangle$.

Blockage instance bl_5 and the corresponding MDC view is shown in Figure 8.17. The view describes the PS-R of resources $rid3, rid4$ and PS-Q of queue $qid4$.

8.3.9.2 Sub-Step M2. Detecting Performance Patterns in the PS-Q and PS-R.

During this step, the analyst manually detects all blockage instances in the MDC view that overlap with z_i in time, i.e., each blockage instance z_j in SEG_{bl_5} such that

- $piStart(z_j) \in [piStart(z_i), piEnd(z_i)]$,
- or $piEnd(z_j) \in [piStart(z_i), piEnd(z_i)]$,
- or $piStart(z_i) \in [piStart(z_j), piEnd(z_j)]$,
- or $piEnd(z_i) \in [piStart(z_j), piEnd(z_j)]$

are detected. The analyst introduces a set per each segment in the MDC segment series $SEG^{a,b}$ for "storing" these blockage instances as follows:

1. Z^{top}_{idle} for blockages in $(complete^{rid_t}\, start^{rid_t})$,

Variant No	$\lvert Z^{\text{top}}_{busy}\rvert$	$\lvert Z_{queue}\rvert$	$\lvert Z^{\text{btm}}_{busy}\rvert$	Root cause in *RCs*
1	–	>0	0	*rcQueue*
2	>0	0	–	*rcTopContResource*,
3	–	>0	>0	*rcBtmContResource*
4	0	0	–	*rcDetectionError*

Table 8.5: Mapping cardinality combinations of sets $Z^{\text{top}}_{busy}, Z_{queue}$ and Z^{btm}_{busy} to root cause in the resource and queue dimensions.

2. Z^{top}_{busy} for blockages in $(start^{rid_t}, complete^{rid_t})$,
3. Z_{queue} for blockages in $(enq^{qid^{a,b}}, deq^{qid^{a,b}})$,
4. Z^{btm}_{busy} for blockages in $(start^{rid_b}, complete^{rid_b})$, and
5. Z^{btm}_{idle} for blockages in $(complete^{rid_b}\, start^{rid_b})$.

For example, for the MDC view in Figure 8.17, the following blockage instances are detected:

1. $Z^{\text{top}}_{idle} = \{bl_6\}$,
2. $Z^{\text{top}}_{busy} = \emptyset$,
3. $Z_{queue} = \{bl_7\}$,
4. $Z^{\text{btm}}_{busy} = \emptyset$, and
5. $Z^{\text{btm}}_{idle} = \{bl_8\}$.

All these instances overlap with bl_5, e.g., $piStart(bl_6) \in [piStart(bl_5), piEnd(bl_5)]$, $piStart(bl_7) \in [piStart(bl_5), piEnd(bl_5)]$, and so on.

8.3.9.3 Sub-Step M3. Identifying Root Causes of the Given Pattern Instance

Finally, the analyst infers what caused the given blockage instance z_i using a mapping between various combinations of the cardinality of sets $|Z^{\text{top}}_{busy}|$, $|Z_{queue}|$, and $|Z^{\text{btm}}_{busy}|$, detected in the previous step, to the root cause in the queue and resource dimensions. We distinguish the following root causes:

1. *problems with queue* $qid^{a,b}$,
2. *problems with top context resource* rid^t,
3. *problems with bottom resource* rid^b, and
4. the special situation *pattern detection errors*.

We write *rcQueue*, *rcTopContResource*, *rcBtmContResource*, and *rcDetectionError* for them respectively, and introduce the set of the queue- and resource-related root causes $RCs = \{rcQueue, rcTopContResource, rcBtmContResource, rcDetectionError\}$. The mapping is shown in Table 8.5. If a cell contains the symbol '–', the cardinality of the corresponding set does not affect conclusions. Note, the cardinality of sets Z^{top}_{idle}

Variant No	$\lvert Z^{\text{top}}_{busy}\rvert$	$\lvert Z_{queue}\rvert$	$\lvert Z^{\text{btm}}_{busy}\rvert$	Root cause
1	0	1	0	queue *rcQueue*

Table 8.6: Mapping cardinality of sets Z^{top}_{busy}, Z_{queue}, and Z^{btm}_{busy} to a root cause.

and Z^{btm}_{idle} are not used in Table 8.5. However, information about blockage instances detected in the corresponding segments can help to reason about possible root causes if pattern detection was done inaccurately, for example, because of a small number of observed cases.

In the following, we consider each one of them and provide the corresponding examples.

Variant 1. Queue Problems. The first line of Table 8.5 describes a cardinality combination when set Z_{queue} has a non-zero cardinality, i.e., there are detected blockages in the corresponding segments, while set Z^{btm}_{busy} have zero cardinality, i.e., no blockage instances are detected there. The cardinality of Z^{top}_{busy} does not matter. This combination is mapped to the root cause *problems with queue*.

This mapping is created due to the following reasoning.

- The segment occurrences of the given blockage instance z_i in the PS-P can be seen as an approximation of occurrences in segments $(start^{rid^t}, complete^{rid^t})$ and $(enq^{qid^{a,b}}, deq^{qid^{a,b}})$.
- Only segment $(enq^{qid^{a,b}}, deq^{qid^{a,b}})$ contains blockages that overlap with z_i in time ($\lvert Z^{\text{top}}_{busy}\rvert = 0$ and $\lvert Z_{queue}\rvert > 0$). So,
  - the top context resource rid^t did not "contribute" to z_i,
  - the queue itself caused z_i either due to its own problems or because the bottom context resource rid^b could not dequeue cases from this queue.
- However, segment $(start^{rid^b}, complete^{rid^b})$ has no blockage instances ($\lvert Z^{\text{btm}}_{busy}\rvert = 0$), i.e., no cases were not delayed due to any serving problems caused by rid^b.
- By excluding both resources from potential root cause sources, we conclude that the queue itself caused blockage instance z_i.

For our running example, cardinality of sets Z^{top}_{busy}, Z_{queue}, and Z^{btm}_{busy} are shown in Table 8.6. According to Table 8.5, this configuration is mapped to root cause *problems with queue* $qid^{a,b}$, i.e., with queue $qid4$. Let us explain the reasoning behind Variant 1 of Table 8.5 by the example shown in Figure 8.17.

- Given blockage bl_5 is comprised by occurrences o_1 and o_2, as shown in Figure 8.11(b). These occurrences can be seen as approximations of occurrences o'_1, o''_1 and o'_2, o''_2 in Figure 8.17.

- Occurrences o'_1 and o'_2 ($rid4$) have almost the same durations as the other occurrences of the same segments in context around, so segment ($start^{rid4}, complete^{rid5}$) does not contain blockages instances. At the same time, o''_1 and o''_2 ($qid4$) have significantly longer durations and form blockage bl_7. So,
  - the top context resource $rid4$ did not "contribute" to bl_5,
  - the queue caused bl_5 either itself or due to the bottom context resource $rid5$.
- However, segment ($start^{rid5}, complete^{rid5}$) has no blockage instances, i.e., $rid5$ was not busy with any other cases longer than usual.
- We conclude that the queue itself caused blockage instance bl_5.

We can additionally consider the *idle* segments of $rid4$ and $rid5$, to validate our reasoning. Both segments have blockage instances bl_6 and bl_8 respectively. They show that these resources were not serving any cases during the time interval of bl_5, and thereby did not delay cases of occurrences o_1, o_2, or dequeuing from $qid4$.

Next, we explain the reasoning behind the remaining Variants 2 and 3 of Table 8.5.

Variant 2. Top Context Resource Problems. The second line of Table 8.5 describes a cardinality combination that is mapped to the root cause *top context resource problems*. It is created due to the following reasoning.

- Similarly to Variant 1, we consider the occurrences of segments ($start^{rid^t}, complete^{rid^t}$) and ($enq^{qid^{a,b}}, deq^{qid^{a,b}}$) as ones that can directly form given blockage instance z_i.
- Among these segments, only ($start^{rid^t}, complete^{rid^t}$) has blockage instance(s) ($|Z^{\text{top}}_{busy}| > 0$ and $|Z_{queue}| = 0$). So,
  - the top context resource rid^t "contributed" to z_i, and
  - the queue itself did not.
- Since the queue segment does not contain blockage instances, the behavior of the bottom context resource did not cause any delays related to cases of z_i (symbol "-" in the corresponding cell of Table 8.5).
- By excluding the queue and bottom context resource from the potential root cause sources, we conclude that the top context resource rid^t caused blockage instance z_i.

To explain Variant 2 by example, we slightly modified the performance spectra shown in Figure 8.17 as follows:

- we removed all the occurrences corresponding to the case of o_1 from the spectra,
- we introduced a delay for o'_2.

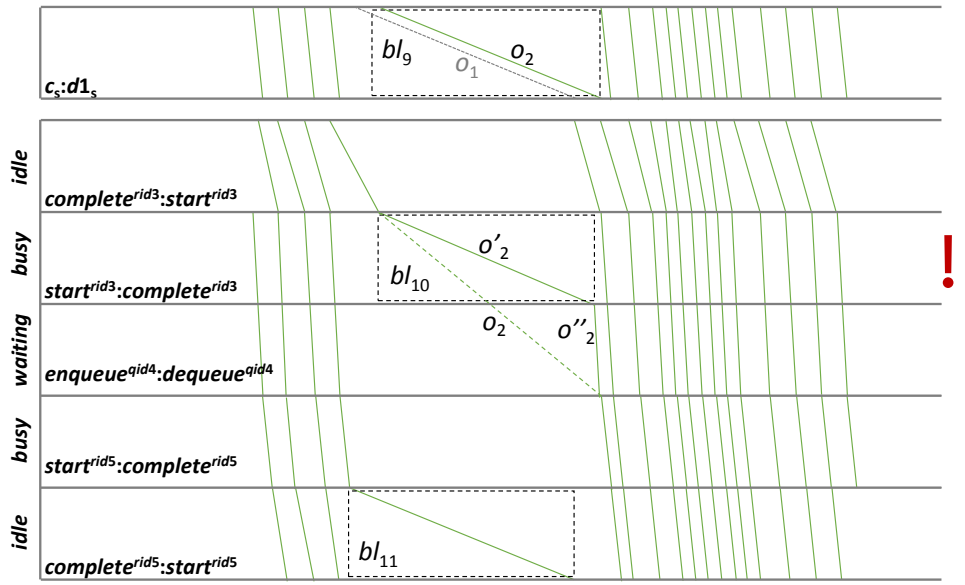


Figure 8.18: MDC view for PS-P segment $(c_s, d1_s)$ showing Variant 2 of Table 8.5.

Variant No	$\lvert Z^{\text{top}}_{busy}\rvert$	$\lvert Z_{queue}\rvert$	$\lvert Z^{\text{btm}}_{busy}\rvert$	Root cause
2	1	0	0	*rcTopContResource*

Table 8.7: Mapping cardinality of sets Z^{top}_{busy}, Z_{queue}, and Z^{btm}_{busy} to a root cause.

The resulting spectra and detected blockage instances are shown in Figure 8.18, and the cardinality combination of the sets is shown in Table 8.7. According to Table 8.5, it is mapped to root cause *problems with top context resource* $rid^{a,b}$, i.e., resource *rid*3 (Variant 2).

Let us explain the reasoning behind Variant 2 by our modified running example in Figure 8.18.

- Given blockage bl_5 comprises occurrence o_2. This occurrence has a longer duration than the others in context because resource *rid*4 served the corresponding case longer than the others in context (occurrence o'_2 in Figure 8.18).
- Queue segment (enq^{qid4}, deq^{qid4}) does not contain any blockage instances, so *qid*4 did not cause z_i, and consequently, the behavior of *rid*5, which could potentially delay dequeuing from *qid*3, did not matter (symbol "-" in Table 8.5).
- We conclude that resource *rid*4 caused blockage instance bl_5.

Variant No	$\lvert Z^{\text{top}}_{busy}\rvert$	$\lvert Z_{queue}\rvert$	$\lvert Z^{\text{btm}}_{busy}\rvert$	Root cause
3	0	1	1	*rcBtmContResource*

Table 8.8: Mapping cardinality of sets Z^{top}_{busy}, Z_{queue}, and Z^{btm}_{busy} to a root cause.

If we additionally consider the *idle* segments of *rid*4, we see it does not contain any occurrences during the time interval of bl_5, i.e., *rid*4 was busy with serving a single case.

Next, we explain the remaining Variants 3 of Table 8.5.

Variant 3. Bottom Context Resource Problems. The third line of Table 8.5 maps a cardinality combination that represents blockage instances in both queue and bottom resource segments to *bottom context resource problems* because of the following reasoning.

- Similarly to Variant 1, we consider the occurrences of segments ($start^{rid^t}$, $complete^{rid^t}$) and ($enq^{qid^{a,b}}$, $deq^{qid^{a,b}}$) as ones that can directly form given blockage instance z_i.
- Among segments ($start^{rid^t}$, $complete^{rid^t}$) and ($enq^{qid^{a,b}}$, $deq^{qid^{a,b}}$) that can cause z_i, only ($start^{rid^t}$, $complete^{rid^t}$) has blockage instance(s) ($\lvert Z_{queue}\rvert > 0$). So, similarly to Variant 1, either queue itself caused z_i or the bottom context resource rid^b could not dequeue cases from this queue.
- Segment ($start^{rid^t}$, $complete^{rid^t}$) contains blockage instances ($\lvert Z^{\text{btm}}_{busy}\rvert > 0$), so it could not dequeue cases from the queue and caused z_i.
- We conclude that the bottom context resource rid^b caused blockage instance z_i.

Note, this variant is only possible if the bottom context resource is *terminal*, i.e., if it hands over cases outside the system. Otherwise,

- it would have at least one queue that takes the cases it serves,
- blockage instances in ($start^{rid^b}$, $complete^{rid^b}$) would be detected in the PS-P.

As a result, we would observe Variant 2 where this resource would be a top context one.

We slightly modified the performance spectra shown in Figure 8.17 to explain this variant:

- we removed any delays for o_1 in segments ($start^{rid4}$, $complete^{rid4}$) and (enq^{qid4}, deq^{qid4}),
- we introduced a delay for occurrence o_1''' in segment ($start^{rid5}$, $complete^{rid5}$).

The resulting spectra and detected blockage instances are shown in Figure 8.19 where the modified occurrences o_1, o_1', o_1'' and o_1''' are shown as thick lines. The corresponding cardinality combination is shown in Table 8.8.

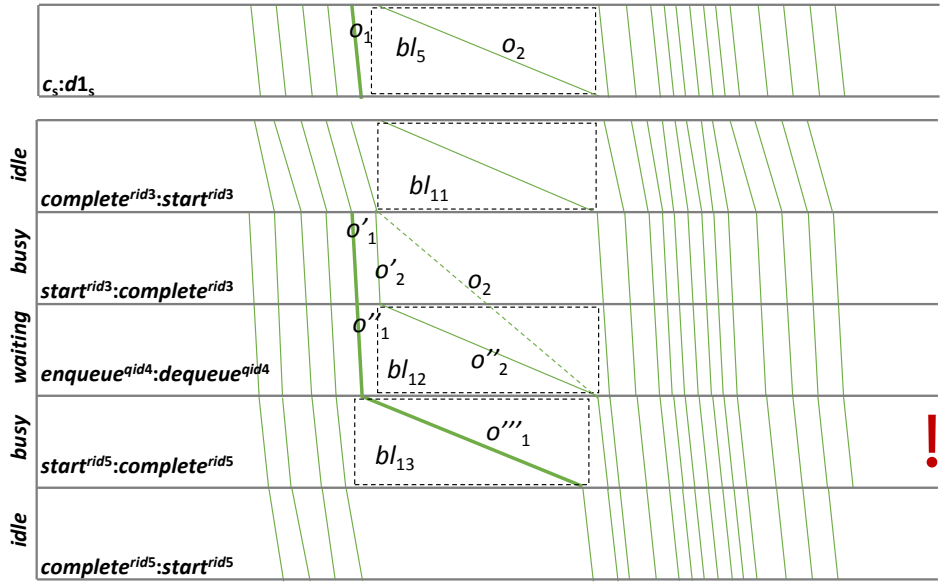


Figure 8.19: MDC view for PS-P segment $(c_s, d1_s)$ showing Variant 3 of Table 8.5.

The reasoning behind Variant 3 for our example in Figure 8.19 is as follows.

- Given blockage bl_5 is comprised by occurrences o_2. This occurrence has a longer duration because of occurrence o_2'' that has a longer duration (blockage instance bl_{12}).
- Resource segment $(start^{rid4}, complete^{rid4})$ does not contain any blockage instances, so it did not cause z_i.
- However, resource segment $(start^{rid5}, complete^{rid5})$ contains blockage instance bl_{13}, i.e., it could not dequeue cases from $qid4$.
- We conclude that resource $rid5$ caused blockage instance bl_5.

If we additionally consider the *idle* segments of $rid5$, we see it does not contain any occurrences during the time interval of bl_5, i.e., $rid5$ was indeed busy with serving a single case.

So far, we considered cardinality combinations of Variants 1-3 that lead to a conclusion. Finally, we consider Variant 4, which indicates errors in the input.

Variant 4. Input Errors. The fourth and last line of Table 8.5 maps a cardinality combination that represents errors in the input caused by, for example, inaccurate pattern detection.

Variant No	$\lvert Z^{\text{top}}_{busy}\rvert$	$\lvert Z_{queue}\rvert$	$\lvert Z^{\text{btm}}_{busy}\rvert$	Root cause
3	0	1	1	*rcBtmContResource*

Table 8.9: Mapping cardinality of sets Z^{top}_{busy}, Z_{queue}, and Z^{btm}_{busy}, related to blockage bl_4 in Figure 8.11(b), to its root cause.

Variant 4 describes no blockages in the top context resource and queue. However, given blockage instance z_i describes a blockage whose occurrences are actually approximations of occurrences in segments of the resources and queue. So, we conclude that blockage instance detection for this MDC view was done with errors and should be reconsidered.

Set RC^{all} is the final result of the analysis that describes

- all detected propagation chains, including pattern instances they contain,
- initial high load instances, triggered by arrival process and caused propagation chains, and
- tuples of initial blockage instances and queue/resource identifiers that triggered them.

Next, we complete the analysis of our running example and show the resulting set RC^{all}.

8.3.10 Completing the Running Example Analysis.

Let us finalize the analysis for our running example. In propagation chain

- $chain_{012} = (\{bl_1, \ldots, bl_5, hl_0, \ldots, hl_5\},$
- $\{(bl_3, bl_1), (bl_3, bl_2), (bl_4, bl_3), (bl_5, bl_3), (bl_5, hl_0),$
- $(hl_1, hl_3), (hl_2, hl_3), (hl_3, hl_4), (hl_4, bl_5),$
- $(bl_1, hl_1), (bl_2, hl_2), (bl_3, hl_3), (bl_4, hl_4), (bl_5, hl_5)\}),$

shown in Figure 8.16, we identified two blockages bl_4 and bl_5 that were not triggered by any other instances in $chain_{012}$. We analyzed bl_5 above. Now, we analyze the remaining bl_4. For simplicity, we do not provide the corresponding MDC view but provide already the cardinality of the sets in Table 8.9. In this table, *rcBtmContResource* points to bottom context resource *rid*6, and we associate bl_4 with this resource:

- $(bl_4, rcBtmContResource, rid6)$.

As a result, the final set of root causes $RC^{\text{all}} =$

- $\{(chain_{012}, \emptyset, \{(bl_5, rcQueue, qid4), (bl_4, rcBtmContResource, rid6)\})\}$.

Now we "translate" propagation chain $chain_{012}$, shown in Figure 8.16, and its root causes in RC^{all} into plain English. We start from the root causes that happened earlier in time.

1. Blockage instance bl_5 happened because of queue $qid4$ (tuple $(bl_5, rcQueue, qid4)$).
2. Afterward, bl_5 caused high load instance hl_0 due to increased load on still working segment $c:d2$ (propagation link (bl_5, hl_0)).
3. Immediately after the end of hl_0, we detected bl_4 in the same PS-P segment. It was caused by bottom context resource $rid6$ (tuple $(bl_4, rcBtmContResource, rid6)$). We assume that it could not handle the increased load, for example, due to the lack of space for baggage off-loading.
4. Afterward, blockage instances bl_5 and bl_4 propagated to segments $(b_s, c_s), (a1_s, b_s)$ and $(a2_s, b_s)$ (propagation links $(bl_4, bl_3), (bl_5, bl_3), (bl_3, bl_2)$ and (bl_3, bl_1).
5. Eventually, blockage instances bl_4 and bl_5 ended at the same time, e.g., due to manual intervention.
6. As a result, blockage instances $bl_1 - bl_3$ ended as well because resource $rid4$ could hand baggage over to $(c_s, d1_s)$ and $(c_s, d2_s)$, unblocking thereby $qid3$, that allowed resources $rid1$ and $rid2$ to resume there work as well.
7. However, baggage that accumulated during $bl_1 - bl_5$ caused high load on the corresponding segment (propagation links $(bl_1, hl_1), \ldots, (bl_5, hl_5)$). Moreover, high load instances propagated to the corresponding outgoing segments (propagation links $(hl_1, hl_3), (hl_2, hl_3), (hl_3, hl_4)$ and $((hl_3, hl_5))$.
8. High load instances $hl_1 - hl_5$ eventually ended, and the system started work normally, i.e., without any blockage or high load instances.

The obtained result can be used to improve the baggage handling process. For example, if conveyor $c:d1$ is stopped, either

- additional resource (e.g., $rid5$) can be scheduled to help $rid6$, or
- arrival process can be changed, e.g., the flow of incoming baggage can be decreased by taking resource $rid1$ or $rid2$ (check-in counters $a1$ and $a2$) out of service.

In the next section, we provide the evaluation of this method.

8.4 Evaluation

We had two use cases to evaluate the performance spectra of the process, queue, and resource dimensions (**RQ-6.1**), and the multi-dimensional performance analysis method (**RQ-6.2**). For one use case, we used the fully controlled environment of our simulation model (see Section 6.7) to implement an undesirable performance scenario in the simulated BHS and generate synthetic event data for analysis. For

another use case, the performance analysis of real Vanderlande-built MHSs was done by Vanderlande's domain experts. However, the amount of information that we can share about the evaluation at Vanderlande is limited by privacy and confidentiality regulations in the project agreement. So, we share only its summary, which does not disclose any sensitive information.

The section is organized as follows. We describe the experimental setup for the first part of the evaluation in Section 8.4.1 and introduce the tool we developed for the method evaluation in Section 8.4.2. We demonstrate post-mortem analyses in Section 8.4.3 and discuss the challenges of real-time monitoring in Section 8.4.4. We summarize the evaluation with real datasets at Vanderlande in Section 8.4.5 and discuss our experimental results in Section 8.4.6.

8.4.1 Experimental Setup

For evaluation, we used the simulation model that is introduced in Section 6.7. The corresponding MFD is shown in Figure 6.14, while the visualization of its PQR-system is shown in Figure 6.15.

We designed the following scenario to generate event data and evaluate our method. Initially, after a cold start, the simulation model worked for six minutes of model time, to obtain some average load to all system conveyors. Afterward, the conveyor on the sorting loop got blocked. After one minute of model time, it resumed. As a result, some bags did not make it to the flight because of this failure.

For this scenario, we formulated the following evaluation analysis question:

- **EvalAQ-1**. What undesirable performance scenario(s) happened, and why?

Answering **EvalAQ-1** implied post-mortem analysis. Then, we also considered the real-time performance monitoring setting.

Real-Time Settings. Potentially, the method can be also applied in the real-time setting to enable performance *monitoring*. It can aim, for example, to trigger correcting actions as soon as an undesirable performance incident starts. For instance, as soon as a propagation chain starts developing,

1. it is discovered,
2. its root causes are identified, and
3. the operator takes countermeasures, e.g., by repairing equipment at the location of the initial blockage instance.

As a result, the impact of the scenario can be minimized.

To understand the applicability of our method (**RQ-6.2**) in the real-time setting, we formulated an additional question:

- **EvalAQ-2**. Can this method be used in the real-time setting for early detection and analysis of undesirable performance scenarios?

8.4.2 Implementation

For the method evaluation, we designed a proof-of-concept implementation [54][1], which we refer to as the *tool*. This tool consists of two components called

1. the *PQR-system interactive viewer* (just *viewer* in the following), and
2. a modification of the PSM [46] presented in Chapter 3, which we call in this section *Multi-Dimensional Performance Spectrum Miner* (MDPSM) for clarity.

The viewer visualizes the PQR-system. It has the following features that make it easier for the analyst (or domain expert) to adopt the PQR-system.

1. The PQR-system layout closely repeats the MFD layout familiar to the analyst.
2. Some verbose model elements, such as arc inscription and place colors (types), are not shown in the visualization.
3. The Q- and R-proclets can be either shown or hidden, depending on the analyst's choice.
4. Zooming, scrolling, and rotating are supported for working with large PQR-systems.

In the MDPSM, we (compared to the PSM) additionally supported the performance spectra of the queue and resource dimensions, visualization of MDC views, and integration with the viewer. The MDPSM

1. imports event data from disk,
2. computes the corresponding PS-P, PS-Q, and PS-R, and
3. visualizes the PS-P and various MDC views in an interactive UI.

Last but not least, the viewer and MDPSM are *integrated* to make the method application easier for the analyst. Thus, the viewer allows seeing which segments of the performance spectra correspond to a transition or place of the P-, Q-, and R-proclets. For example, by clicking on a P-proclet place, the analyst is navigated to the corresponding PS-P segment in the MDPSM, and by clicking on the queue place of a Q-proclet, the corresponding MDC view is shown in the MDPSM.

The other way around, the MDPSM allows navigating from performance spectra back to the corresponding elements of the PQR-system. That helps, for example, to easier track high load and blockage propagation (see Section 8.3.4), using the P-proclet to determine related segments in the PS-P. Additionally, an MDC view for a

[1]The source code, binaries, and documentation are available on `https://github.com/processmining-in-logistics/psm/tree/pqr`

PS-P segment can be automatically obtained from the MDPSM UI without interacting with the viewer.

In our tool, we configured the viewer to visualize the PQR-system of the simulated BHS in Figure 6.14, and imported the recorded event table, corresponding to the scenario of Section 8.4.1, in the MDPSM. The resulting PS-P is shown in Figure 8.20.

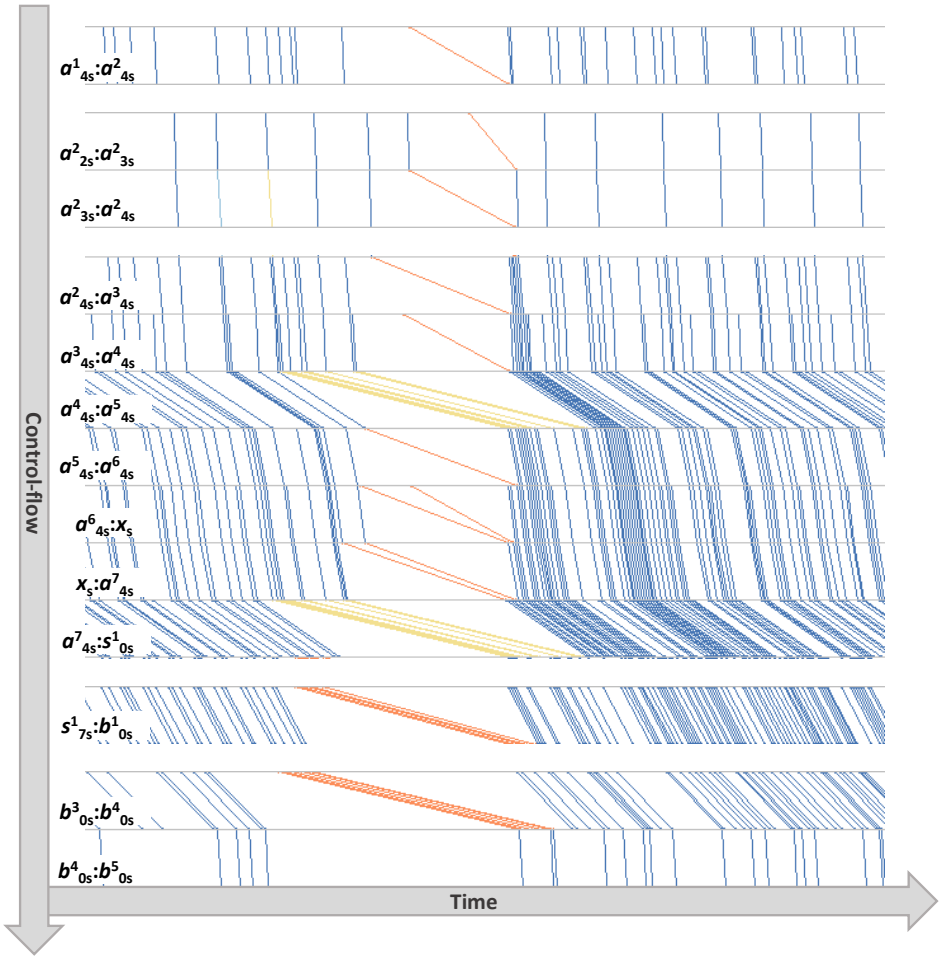


Figure 8.20: Obtained performance spectrum (PS-P).

In the next section, we demonstrate the post-mortem analysis using our method.

8.4.3 Analysis Using Synthetic Data

In this section, we follow our method step by step in order to answer **EvalAQ-1**.

For analysis, the input was:

- the PQR-system shown in Figure 6.15, and
- the PS-P, PS-Q, and PS-R computed and visualized by the MDPSM.

The PS-P is shown in Figure 8.20, while relevant fragments of the PS-Q and PS-R we provide later. Note, a part of the PS-P segments is not shown in Figure 8.20 to avoid clutter. Thus, the segments of most incoming and outgoing conveyors, the segments of scanner shortcuts, and the sorter segments between $a^7_{4s} : s^1_{0s}$ and $s^1_{7s} : b^1_{0s}$, and segments between $s^1_{7s} : b^1_{0s}$ and $b^3_{0s} : b^4_{0s}$ are not shown. We started the analysis with performance pattern detection.

Step 1. Performance Pattern Detection. In this step, we used the MDPSM as a visual analytics tool, i.e., the analyst's brain was used for detecting high-load and blockage instances (see Section 8.3.3). The result is shown in Figure 8.21, where blockage instances $bl_1 - bl_{12}$ are shown by shapes with black dashed lines, and high load instances $hl_1 - hl_{11}$ are shown by shapes with red dotted lines. The resulting set $PI = \{bl_1, \ldots, bl_{12}, hl_1, \ldots, hl_{11}\}$. Note, the instances on the segments that are not shown are not in PI. However, we detected blockage instances on the sorter segments between $a^7_{4s} : s^1_{0s}$ and $s^1_{7s} : b^1_{0s}$, and between $s^1_{7s} : b^1_{0s}$ and $b^3_{0s} : b^4_{0s}$.

Step 2. Propagation Chain Discovery. In this step, we iterated over the instances in PI. We started by choosing (randomly) blockage instance bl_4 in segment $seg_4 = (a^2_{4s}, a^3_{4s})$ as a starting point for propagation discovery algorithm (see Section 8.3.5). To understand what segments are connected to seg_4, we referred to the P-proclet visualized in the viewer. For that, we double-clicked the segment space in the MDPSM to tell the tool to show the corresponding model elements in the viewer. As a result, in the corresponding P-proclet fragment we could see that the beginning of seg_4 was connected to segments $(a^1_{4s}, a^2_{4s}), (a^2_{3s}, a^2_{4s})$, and the end of seg_4 was connected to (a^3_{4s}, a^4_{4s}). When we clicked on the corresponding place and got navigated to the corresponding segments in the MDPSM, we could see there already detected blockage instances bl_1, bl_3, and bl_5. According to Step 2 in Section 8.3.5, we

1. added propagation chain links (bl_4, bl_3), (bl_4, bl_1) and (bl_5, bl_4) to a set *Links* (note, propagation *splits* on merge unit a^2_4 toward segments (a^1_{4s}, a^2_{4s}), and (a^2_{3s}, a^2_{4s})),
2. added segment (a^2_{4s}, a^3_{4s}) to the list of visited segments *visitedSegments*,
3. added instances bl_1, bl_3, bl_4, and bl_5 to the set of instances Z_1, and
4. repeated the discovery steps for blockage instances bl_1, bl_3, and bl_5.

Propagation forward-tracking stopped on instances bl_1 and bl_2. Propagation backtracking stopped on instance bl_{12} (as the segments in between, not shown in the figure, had detected blockages as well) because next segment (b^4_{0s}, b^5_{0s}) did not have a

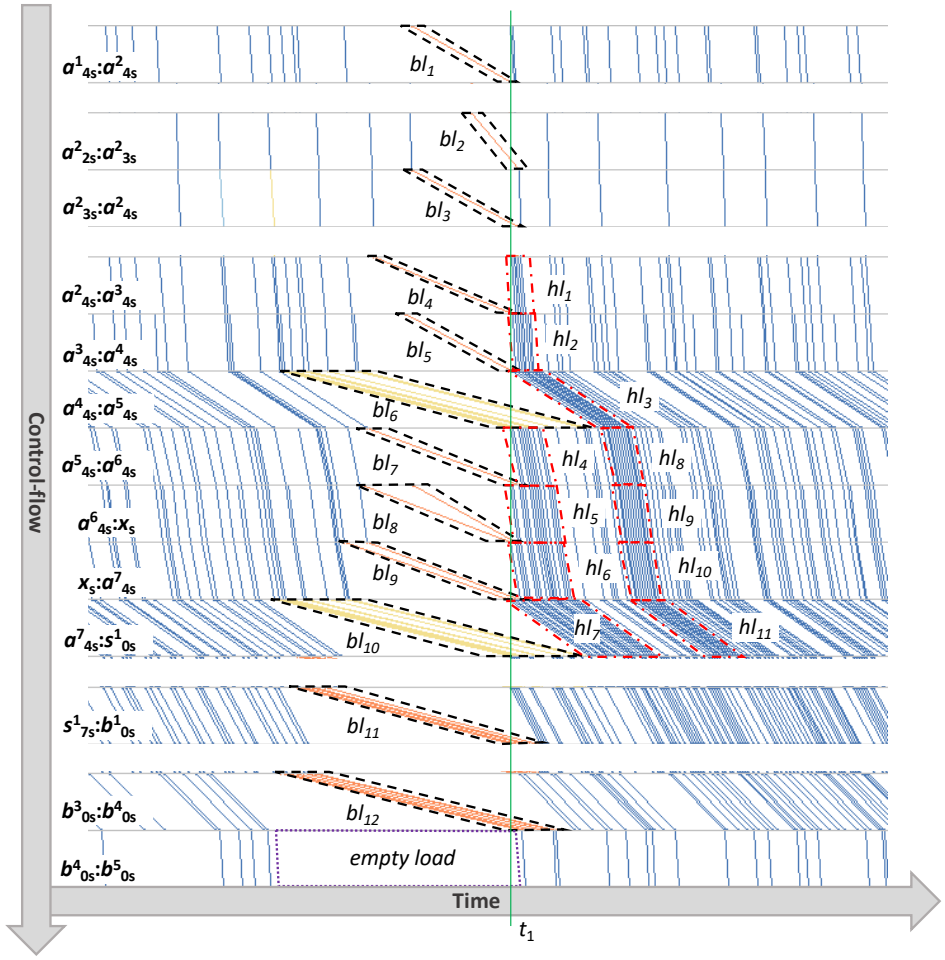


Figure 8.21: Detected blockage and high load instances.

blockage instance during the time interval of interest. The resulting propagation links were $Links_1$ =

- $\{(bl_{10}, bl_9), (bl_9, bl_8), (bl_8, bl_7), (bl_7, bl_6), (bl_6, bl_5),$
- $(bl_5, bl_4), (bl_4, bl_3), (bl_3, bl_2), (bl_3, bl_1), \ldots\},$

where … stand for the segments that are not shown, i.e., links related to instances bl_{11} and bl_{12} are not in $chain_1$ because the related instances are excluded from the figure. These links connect instances $Z_1 = \{bl_1, ; bl_{12}\}$. The resulting propagation chain

is $chain_1 = (Z_1, Links_1)$. It is shown in Figure 8.22 by orange arrows. We added $chain_1$ to set $Chains_1$.

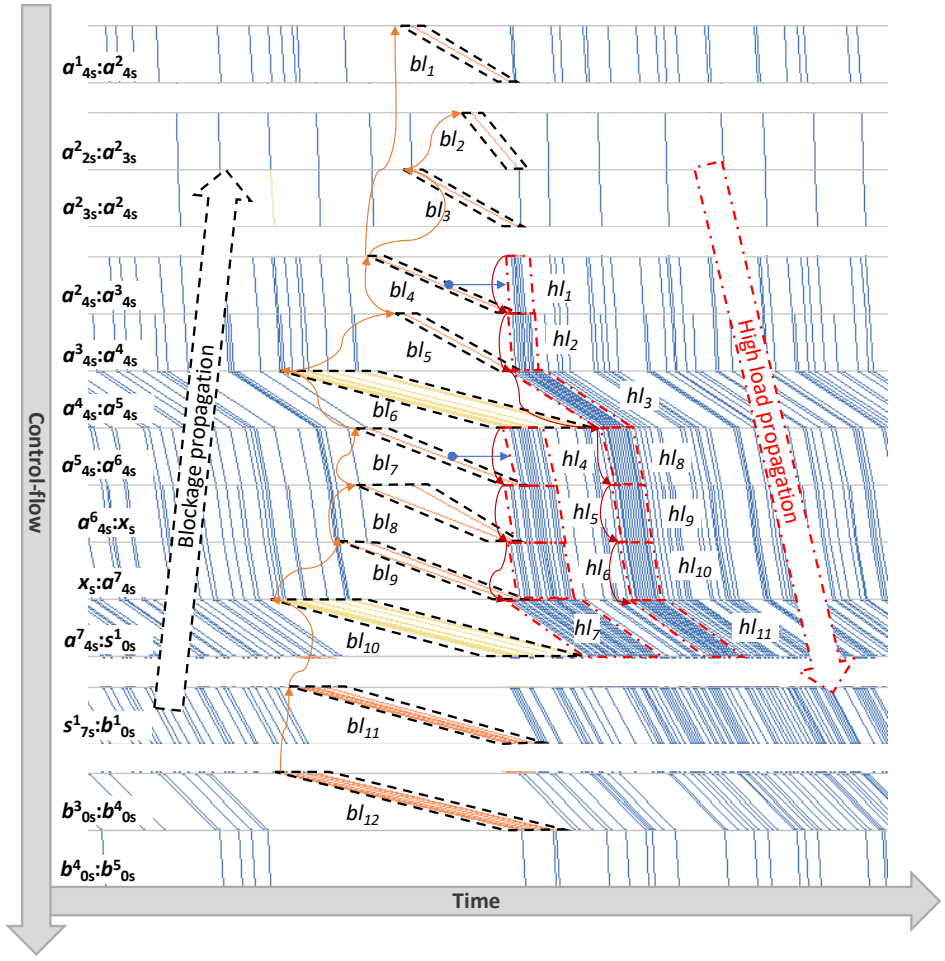


Figure 8.22: Blockage and high load instance propagation.

After $chain_1$ was discovered, only high load instances were among those not visited yet, so we continued with the discovery of high load instance propagation chains. Applying the same, we connected high load instances

- $Z_2 = \{hl_1, hl_2, hl_3, hl_8, \ldots, hl_{11}\}$ and
- $Z_3 = \{hl_4, \ldots, hl_7\}$

through propagation links

- $Z_2 = \{(hl_1, hl_2), (hl_2, hl_3), (hl_3, hl_8), (hl_8, hl_9), (hl_9, hl_{10}), (hl_{10}, hl_{11})\}$, and
- $Z_4 = \{(hl_4, hl_5), (hl_5, hl_6), (hl_6, hl_7)\}$

as propagation chains

- $chain_2 = (Z_2, Links_2)$, and
- $chain_3 = (Z_3, Links_3)$.

These chains are shown in Figure 8.22 by red arrows. The result of Step 1 was set $Chains_1 = \{chain_1, chain_2, chain_3\}$.

Steps 3 and 4. Merging Propagation Chains. In the PS-P, we did not observe any propagation due to the alternative paths' unavailability (Step 3). Next, we did Step 4 and check if any high load instances were triggered due to the ending of blockage instances. The initial instances of $chain_2$ and $chain_3$ were hl_1 and hl_4 respectively. The beginning time of these instances was equal to the end time of same segment blockage instances bl_4 and bl_7 respectively. We concluded that these blockage instances triggered high load instances immediately after they ended. The resulting links were

- $link_{41} = (bl_4, hl_1)$, and
- $link_{74} = (bl_7, hl_4)$.

Link $link_{41}$ connected $chain_1$ and $chain_2$, while $link_{74}$ connected $chain_1$ and $chain_3$. As a result, we merged these three chains into one. The resulting chain was $chain_{123}$ =

- $(Z_1 \cup Z_2 \cup Z_3, Links_1 \cup Links_2 \cup Links_3 \cup \{link_{41}, link_{74}\})$.

It comprised all the instances and links shown in Figure 8.22. The result of Step 4 was set $Chains_3 = \{chain_{123}\}$.

Step 5. High Load Instance Analysis. In chain $chain_{123}$, no high load instances were initial ones, so the result of this step was $RC^{\text{hl}} = \emptyset$.

Step 6. Multi-Dimensional Analysis of Blockage Instances. In chain $chain_{123}$, blockage instance bl_{12} in segment (b^3_{0s}, b^4_{0s}) was initial, i.e., it was not triggered by any other instance of $chain_{123}$. The corresponding MDC view is shown in Figure 8.23. In this view, we could detect one blockage instance bl_q in the queue segment and no blockage instances in the *busy* segments of the top and bottom context resources. According to Table 8.5 we chose Variant 1 and root cause *rcQueue*. That is, the problem with queue $b^3_{0s} : b^4_{0s}$ caused the whole propagation chain $chain_{123}$.

So far, we considered the post-mortem analysis. Next, we discuss the same scenario assuming real-time monitoring instead, to identify the challenges of applying our method in the run-time setting.

8.4.4 Monitoring Using Synthetic Data

In the real-time setting, the MDPSM can receive a *stream* of events via the network, incrementally compute new spectrum occurrences, and update the performance spectra visualization in the UI in run-time. For answering **EvalAQ-2**, we set up the MDPSM to listen to the stream of events that the simulation model generated.

However, there were two problems: the last activity and timestamp of still ongoing occurrences could not be known. As a result, they needed to be *estimated* first. The former is the problem of the next activity prediction, it addressed in many works, e.g., in [39]. The latter is the problem of next timestamp prediction [32]. However, these problems are notoriously hard and are not fully solved for systems with shared resources and queues. Instead, we came up with a simple solution that still allowed us to evaluate our method in the monitoring mode, assuming that the aforementioned problems can be eventually solved.

Naive Next Activity Prediction Approach. In the PQR-systems, the P-proclet describes which process steps (activities) are possible after the execution of the current process step. Interestingly, the P-proclet allows alternatives only for splits, while for the other elements the next activity is always known. As a result, the next activities that do not follow a diverting step can be precisely estimated.

Naive Next Activity Timestamp Prediction. To detect longer occurrences, the analyst uses end segment occurrence timestamps to estimate if the occurrence duration is longer than some usual duration δ, i.e., for the occurrences (t_a, t_b) of a segment (a, b) duration $t_b - t_a$ is compared with δ. If $t_b - t_a > \delta$ the analyst can immediately consider this occurrence as part of a blockage instance. So, we only care whether the occurrence duration is longer than some threshold and if yes, we do not need its exact value. If we know δ, we can classify a segment occurrence as delayed as soon the time interval passed since the occurrence started is greater than δ, i.e., $t_{\text{now}} - t_a > \delta$.

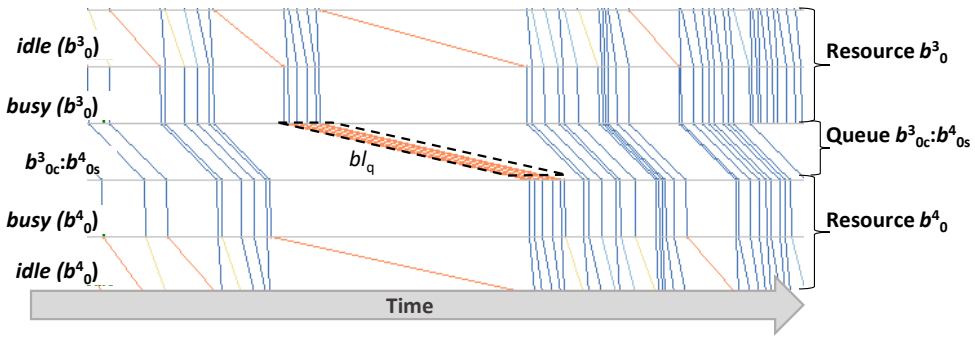


Figure 8.23: MDC view for segment (b^3_{0s}, b^4_{0s}).

As a result, by applying these approaches together, we can detect blockage instances in segments whose first process step is not diverting.

We implemented these approaches in the MDPSM. For that, it obtains the next activity from the P-proclet and compares the ongoing occurrence duration with some threshold δ, computed for each segment using previously observed segment occurrence durations. The MDPSM visualizes the ongoing segment in grey if its duration is currently equal to or less than δ, and in pink otherwise. For the segments started by diverting steps, no ongoing occurrences are shown. Such a PS-P is shown in Figure 8.24 for time t_1. In this spectrum,

- the fully observed segment occurrences are shown in blue,
- the ongoing segment occurrences are shown in grey if they are not classified as *slow* (yet), and in pink otherwise.

This PS-P shows that many ongoing (pink) occurrences already form blockage instances. Thus, all blockage instances in Z_1 except bl_{12} can be already detected at t_1. The missing blockage instance bl_{12} in segment (b^3_{0s}, b^4_{0s}) does not contain any ongoing occurrences because process step b^3_0 is diverting, so the next process step cannot be estimated using our approach (it can be either b^4_0 or b^3_1). Nevertheless, the blockage propagation chain, consisting of links between $bl1 \ldots bl_{11}$, can be already discovered.

So we concluded that during an ongoing blockage propagation chain development, the chain or its part can be detected *before* its development completes, and the initial segments can be (at least partially) localized. In our example, an engineer could be sent for the completion of the propagation chain discovery on-site, to start the investigation at the location bl_{11}, and undertake correcting actions.

8.4.5 Analysis Using Real Datasets

Our performance spectrum-based methods, gradually developed over the project span and proposed in this thesis, were extensively evaluated at Vanderlande. Respectively, process model-unaware analysis was evaluated using the PSM [46] on the data of a large European airport BHS (see Section 3.5) at the beginning of the project. Afterward, the concept of the analysis based on an integrated performance spectrum and process model gradually evolved. For its evaluation, an internal software tool was developed by Vanderlande. They used as input (1) our PSM code, and (2) the event data processing framework that we developed using Scala parallel collections[2] and Apache Spark[3] for

- pre-processing MHS event data,
- extracting event log and event tables from large event datasets, and

[2] `https://docs.scala-lang.org/overviews/parallel-collections/overview.html`
[3] https://spark.apache.org/

- advanced filtering on the level of events, event attributes, and traces.

Note, this framework is not open-sourced due to the presence of information about Vanderlande's internal data structures and other confidential information in its code.

Further, this tool was used for evaluation. For that, multiple sessions of performance analysis for different MHSs were accomplished by Vanderlande's process engineers. The evaluation showed that:

1. the tool allowed to identify scenarios and root causes that the other tool used at Vanderlande could not reveal,

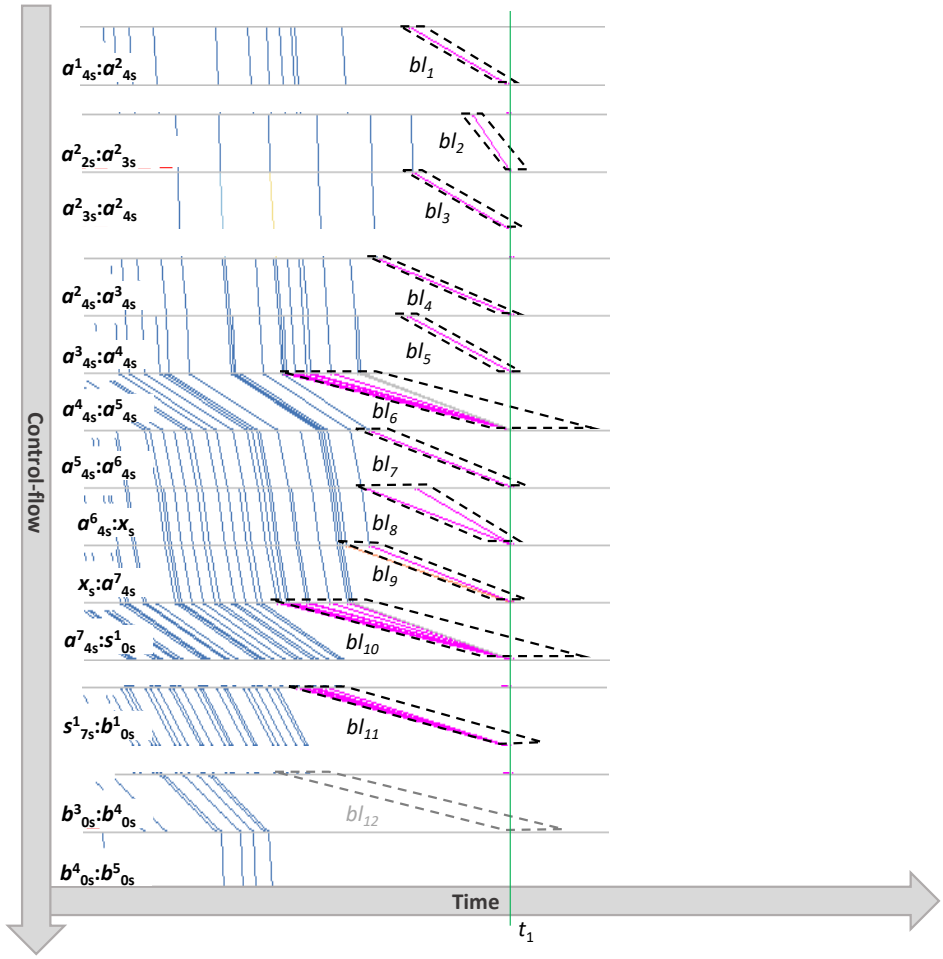


Figure 8.24: PS-P with estimated ongoing segments at time t_1.

2. the learning curve for process engineers was shallow since a lot of domain knowledge was already incorporated in the tool, and the performance spectrum was easy to read and interpret, and
3. the resulting tool made it much easier to do the analysis than the model-unaware performance spectrum-based approach of Chapter 3.

Next, we summarize the evaluation results.

8.4.6 Evaluation Results

During the evaluation, **RQ-6.1** and **RQ-6.2** were considered as the sub-questions of **RQ-6**. The method for multi-dimensional performance analysis (**RQ-6.2**) extensively uses the performance spectra of the process, queue, and resource dimensions of PQR-systems, so **RQ-6.1** was effectively evaluated within the evaluation of **RQ-6.2**, i.e., by addressing **EvalAQ-1**.

The tool we implemented could clearly demonstrate how to relate the elements of the PQR-system to the corresponding performance spectrum, and vice versa (**RQ-6.1**). While the PS-P was used to detect undesirable performance patterns instances, such as high load and blockages, the spectra of the queue and resource dimensions showed what behavior caused it in the queue or resource dimension. Last but not least, the synchronization among the transitions of different PQR-system proclets could be seen in the performance spectra when they were arranged in an MDC view. Such views helped to understand the full dynamics related to a PS-P segment.

The method for multi-dimensional performance analysis (**RQ-6.2**) was evaluated with two different use cases: using a synthetic dataset with our own scenario and tool [54], and on the use cases of analysis of real MHSs, using a software tool implemented by Vanderlande. These evaluations showed the following.

- Analysis could be done manually, without any automated pattern detection capabilities.
- The method allowed us to discover propagation chains of blockage and high load instances, and explain their root causes in the queue and resource dimensions.
- Integration of performance spectra and PQR-systems:
  - helped to interpret performance segments and the process steps comprising them,
  - allowed to quickly sort segments according to the control flow,
  - allowed to interpret segment occurrences durations by referring to the minimum service time of resources, and minimum waiting time of queues,
  - helped identify segments, related to analysis questions, among many other segments,
  - allowed to *not* refer to the MFD during analysis, and

– minimized the need for the involvement of domain experts.

These capabilities overcame all the limitations of performance spectrum-based analysis, considered in Section 8.2, and the main limitation of [77], i.e., incapability of RCA. Additionally,

- the integrated spectra and PQR-system enabled the discovery of propagation chains without relating to exact time boundaries of the detected pattern instances,
- applicability of manual pattern detection allowed fast adoption without the need for automatic pattern detection,
- and the presence of a process model (PQR-system) allowed for benefits of generalized conformance checking, i.e., the use of complete event tables instead of incomplete.

As a result, the remaining limitations of [77, 78] considered in Section 8.2 were addressed as well. We conclude that **EvalAQ-1** was answered.

Last but not least, evaluation in real-time settings showed that the method is potentially applicable for monitoring (**EvalAQ-2**) if techniques for predicting the next activity and timestamp are in place.

In the following, we consider the limitations of the method and summarize this chapter.

8.5 Chapter Summary

In this chapter, we addressed the problem of process performance RCA. For that, we first considered the state-of-the-art approaches and discuss their limitation. Then, we formulated a general research question **RQ-6** about relating performance spectra and the process, queue, and resource proclets of PQR-systems. To answer it, we divide this question into two sub-questions. The first one (**RQ-6.1**) addresses the problem of performance description of the queue and resource dimensions using performance spectra. The second one (**RQ-6.2**) addresses the problem of RCA of systems with shared resources and queues.

For answering **RQ-6.1**, we defined the event logs of queues and resources required for computing corresponding performance spectra PS-Q and PS-R. We discussed what kind of information can be inferred from these spectra, and what performance patterns are of interest there. Additionally, we considered how synchronization among the P-, Q- and R-proclets of PQR-systems allows for grouping the segments of PS-Q and PS-R into MDC views that explain the behaviors of the resources and queues related to two process steps forming a segment in a PS-P. We provided a running example that showed how the performance of an individual case, observed in the PS-P, can be explained through the corresponding PS-Q and PS-R.

Further, for answering **RQ-6.2**, we proposed a method for multi-dimensional performance analysis using PQR-systems and performance spectra, computed from complete event tables. This method's steps include

- performance pattern detection in performance spectra,
- discovery of propagation chains comprised from the detected performance pattern instances,
- merge of smaller propagation chains if under certain conditions derived from the spectra and PQR-systems, and
- RCA using MDC view obtained from the PS-Q and PS-R.

The final result of this method includes a detailed description of detected and discovered undesirable performance scenarios (i.e., propagation chains) and their root causes.

We evaluated these techniques using synthetic data with our software tool, and Vanderlande evaluated it using the tool they implemented in-house. The evaluation showed that

- the PS-Q and PS-R described the behaviors of queue and resources, and MDC views described the behaviors of interest with respect to the corresponding segment of the PS-P.
- the analysis following the method's steps allowed to identify the root causes of undesirable performance scenarios,
- the analysis could be done manually by the analyst, and
- the method could be applied for monitoring in the real-time setting, but only under certain conditions that we discuss in the following as limitations.

Additionally, we showed how this method overcame the discussed limitations of state-of-the-art methods. As a result, it answers **RQ-6.1** and **RQ-6.2**, and the general **RQ-6** as well.

However, we identified the following limitations.

1. *Resource availability unawareness.* Not taking resource availability into account can impede the accuracy of inferred root causes if a questionable resource was unavailable.
2. *Merging priority unawareness.* Absence of information about priorities for merging resources in the PQR-system does not allow to discover when high load instances trigger blockage instances.
3. *Unobservability of ongoing segments.* If the method is applied in the real-time setting for monitoring, the end process step activity and time of still ongoing segments are not observed yet. Although we suggested a simple method that partially solves this problem, it is desirable to have in place the solutions for predicting the next process step activity and time.

4. *Manual analysis.* Our method implies that the analyst manually does pattern detection, propagation chain discovery, and RCA. It is an advantage if a quick adoption or ad-hoc analysis of a system is required. However, automated detection, discovery, and RCA are highly desirable for regular method applications.

These limitations can be addressed in future work by extending the PQR-system with information about resource availability and merge priority, adopting existing performance pattern detection algorithms, and automating discovery and RCA steps.

Chapter 9

Predictive Performance Monitoring

In this chapter, we address the problem of *Predictive Performance Monitoring* (PPM) for systems with shared resources and queues. We categorize this general problem into two categories: predicting PPIs for individual cases separately and predicting aggregate PPIs for many cases together. The latter corresponds to **AQ8** (predicting load on a specific system part). We formulate an RQ addressing the problem of predicting aggregate process performance indicators given an event table and PQR-system.

To answer it, we show how performance spectra can capture the dynamics of systems and processes. We formulate our problem over performance spectra, thereby solving the RQ in the part of capturing system dynamics. Afterward, we propose a method for identifying relevant features for training predictive ML models, building on the PQR-system and the performance analysis method of Chapter 8. We provide evaluation results obtained using both synthetic and real datasets and discuss the method limitations.

9.1 Motivation

In this section, we provide motivation for addressing the problem of PPM for systems with shared resources and queues, formulate our RQ, outline our method for answering this RQ, and summarize evaluation results.

9.1.1 Predictive Performance Monitoring

Along with descriptive performance analysis, PPM is of crucial importance for process support. It aims to alarm agents supporting a process, or operators supporting a

system, about coming performance issues. To make predictions, historic data about the process or system execution are usually used [32]. As a result, performance issues can be potentially prevented, or at least mitigated timely. A classical example is predicting Service Level Agreement (SLA) violations for ongoing cases. Whenever an SLA violation for a case is predicted, a responsible manager is notified. This manager can take countermeasures, for example, the problematic case can be scheduled to be handled ahead of the others for preventing the SLA breach.

In MHSs, TSU mishandling can be predicted and prevented using the historic information about handling TSUs in the system. For example, a situation when a bag in a BHS is going to be late for a flight can be predicted and avoided by the correcting actions of the system operators. This problem is formulated as **AQ4** in Section 4.2. It requires PPM on the *case* level.

However, PPM is not restricted to the case level. In systems with shared resources and queues, multiple cases, co-located in the same system queue, can be delayed together when the corresponding conveyor stops, or cases in different queues can be delayed due to competing for the same system resource. In MHSs, the former leads to congestion, and the latter leads to longer waiting times for such a resource for multiple cases. As a result, the corresponding TSUs can be mishandled. If information about coming congestion or a longer waiting time is available in advance, such situations can be prevented, or timely mitigated. That is, PPM on the *system* level is required for that.

Let us consider a system-level PPM problem example for the BHS fragment shown in Figure 9.1(a). In this system, baggage enters the system at the check-in islands and travels via the Links onto the preliminary sorting loops $P1$ and $P2$. On these loops, each bag must be screened for prohibited items by the scanners for obtaining clearance and proceeding to its destination. The availability of these scanners is critical for the overall system performance. Whenever a bag cannot be diverted toward them, it starts recirculating on the loop, thereby reducing its free capacity. As a result, it becomes more difficult for the other bags to join this loop, including bags leaving the scanners. Travel time to destination increases, and bags can be late for the flight. In the worst-case scenario, the situation can deteriorate fast and cause the sorter to halt, causing even longer delays for many bags. Potentially, such situations can be prevented by predicting load peaks at location S_1 (see Figure 9.1(a)) to reduce the incoming baggage flow coming from the check-in areas timely.

We can formulate the following problem for this example. Let a PPI that shows load at S_1 (e.g., a number of bags within a 30-second interval), and a prediction horizon t_{ph} defining in what time from *now* we want to predict load at S_1. For that,

- recent historic data about the load on check-in counters can be used to capture the recent states at check-in areas, and

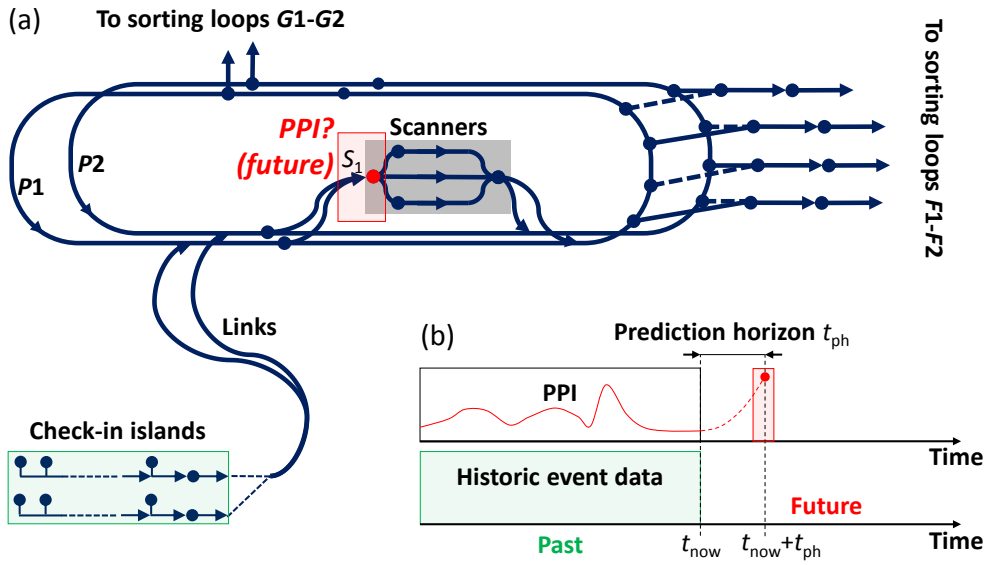


Figure 9.1: PPM problem example: predicting load on the scanners at location S_1 using historic event data recorded from the check-in islands.

- historic data about the system execution for past weeks or months can be used to learn a model for estimating (predicting) future load at S_1 within t_{ph}.

Figure 9.1(b) illustrates this problem:

- historic data are available till *now* (t_{now}),
- historic PPI values are also available till t_{now},
- a predictive model can be learned from historic information about dependencies between historic data and future PPI values, and
- future value of the PPI at time $t_{\text{now}}+t_{\text{ph}}$ is estimated (predicted) using the recent historic data as input.

This problem addresses predicting *aggregate* PPIs, i.e., PPIs describing the performance of many cases *together* co-located in a particular part of a system or process (system-level PPM), i.e., it addresses **AQ8** (see Section 4.2).

In the MHS domain, PPM on both levels is actual, as **AQ4** and **AQ8** prove. However, in which order should they be approached, assuming there are no reliably working techniques yet? To decide, we consider two factors:

1. feasibility of addressing each problem in this thesis scope,
2. impact of solving each problem on improving the overall system performance.

Feasibility. How feasible to target predicting the performance of individual TSUs? To estimate it, let us reconsider the running example of Section 8.3. In this example, some bags in the BHS, shown in Figure 8.2, could not make it to the flight due to blockage instances in the system. The corresponding performance spectrum is shown in Figure 9.2, which shows that, for example, bag *pid*2 was delayed by blockage instance bl_4 and could not make it to the flight. Potentially, case-level PPM could help to avoid this situation.

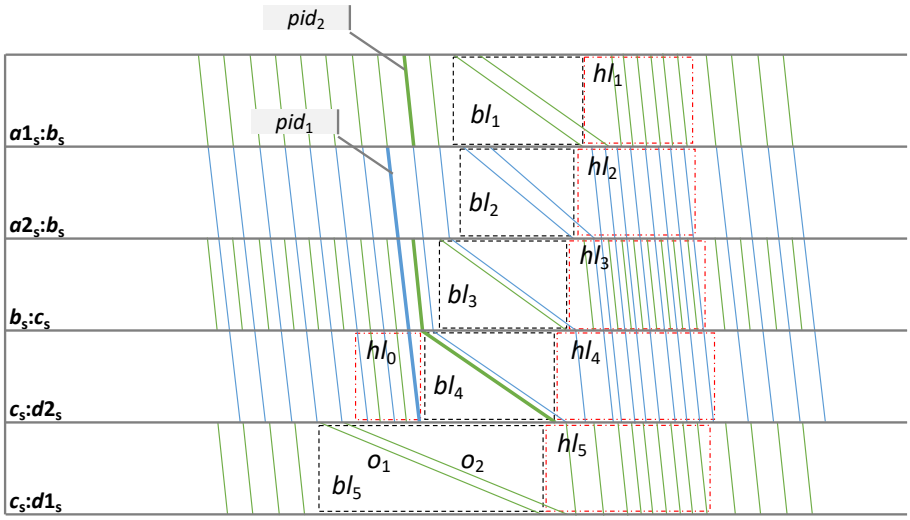


Figure 9.2: In the PS-P with detected pattern instances, bag pid_1 could make it to the destination $d2_S$, while bag pid_2 was delayed by blockage instance bl_4.

Let us consider the handling of bags *pid*1 and *pid*2 in Figure 9.2 in more detail. While *pid*1 could make it to the final destination without delays, *pid*2 was delayed by bl_4. However, if bl_4 emerged a bit earlier, *pid*1 would not make it to the flight as well. Alternatively, if resource b merged *pid*2 first, and *pid*1 afterward, *pid*2 would not be delayed, while *pid*1 would be late for the flight. That is, even such subtle things are the factors that can dramatically change the outcome of an individual case. In real large systems, the number of such factors is enormous. It makes it hard to consider all of them for providing reliable predictions on the case level.

In contrast, factors affecting the performance of many cases together are more "stable". For example, bl_4 would have happened disregarding whether $pid1, pid2$, or both formed it. Additionally, our observations about the propagation of high load and blockage (see Chapter 8) showed how the performance of many cases together in past affects the performance of many cases together in the future. It already lays some foundation for reasoning about predicting the future performance of many collocated cases, i.e., for system-level PPM.

Impact. Case-level PPM allows for correcting actions per TSU. However, the capacity of the system operators for preventing actions per TSU is limited. As a result, only the performance of a limited number of TSUs, whose impact on the system performance is also limited, can be improved. In contrast, predicting performance issues for many TSUs would allow for correcting the situation for many TSUs simultaneously. As a result, issues with a higher impact on the overall system performance would be prevented.

All things considered, we focus on system-level PPM in this thesis, i.e., on **AQ8**. Next, we consider a state-of-the-art approach we used as a baseline, and we refer to the literature review in Chapter 5 for a detailed discussion of other related approaches.

9.1.2 Data-Driven Feature Identification

After an extensive literature search, we identified a state-of-the-art approach [35] of PPM for processes with shared limited resources, such as systems with shared resources and queues we consider in this thesis. It allows encoding *inter-case* features, i.e., features that capture the interplay of multiple ongoing cases that are being executed concurrently, for training an ML model for predicting a chosen PPI on the case level. Additionally, *intra-case* features, capturing the properties of each individual case, are used for the model training.

In [35], inter-case feature identification is data-driven, i.e., it requires no domain knowledge about the process. For that, concurrently executed cases of interest are automatically categorized and grouped by their temporal and control-flow proximity. Proximity assumes competing for the same limited resources, i.e., only the cases in the same category assumably interplay. For temporal proximity, the *city distance* and *snapshot metric* are used. The former considers the beginning of trace prefixes, and the latter considers their endpoints. The control-flow proximity uses *edit distance* between trace prefixes. Afterward, various properties of these categories, such as the number of cases in a category, are used as inter-case features describing the case interplay. This data-driven approach extends a knowledge-driven approach [173] that uses domain knowledge about a process for categorization. Intra-case features are extracted from the information about the recent history of a trace prefix. The resulting feature set contains both inter- and intra-case features. It can be used for training an ML model of a chosen architecture.

Although this approach is designed for the case-level PPM, it can be used for system-level PPM by aggregating predictions for multiple concurrently executed cases, relevant to a target PPI. However, considering its application for PPM of systems with shared resources and queues, we identified important drawbacks of inter-case feature selection based on the suggested proximity metrics. To show that, we introduce an example of a simple BHS, and several trace prefixes first.

start time	prefix	$\langle a,b,c\rangle$	$\langle e,b,c\rangle$	$\langle a,b,f\rangle$	$\langle e,b,f\rangle$
0	$\langle a,b,c\rangle$	-	1,1	1,0	2,1
1	$\langle e,b,c\rangle$		-	2,1	1,0
0	$\langle a,b,f\rangle$			-	1,1
1	$\langle e,b,f\rangle$				-

Table 9.1: Trace prefix proximity.

This BHS is shown in Figure 9.3. It has a main flow $(a,b),(b,c),(c,d)$ going from check-in counter a toward exit d, an additional incoming conveyor (e,b) (connecting check-in counter e), and a sorting loop s connected via (b,f) and (g,d). The figure also shows the minimum travel time for each conveyor, and loop s. The target PPI is defined as a *time to complete* for each case.

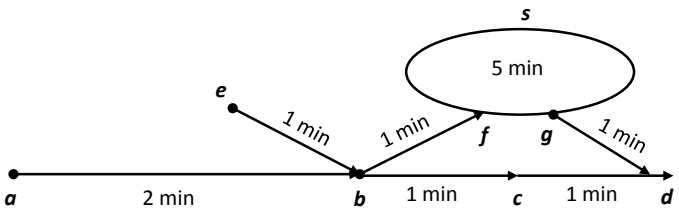


Figure 9.3: Simplified MFD of a BHS showing two possible paths from check-in counters a and e toward exit d.

Let us consider four trace prefixes that just reached locations c or f at time $t = 2min$. They are shown in column *prefixes* of Table 9.1. Each one of them has the same snapshot metric value because they reached c or f at the same time t. It is usually the case for PPM settings because a prediction is made at a particular moment in time, so all active cases have close snapshot metric values. However, the city distance between their possible pairs is different because these traces started at different times (see column *start time* of Table 9.1). The remaining columns show the pairs of the simplified city and edit distances between the prefixes.

Using these values, we identify two categories of prefixes that are close to each other with respect to those metrics:

1. $\{\langle a,b,c\rangle,\langle a,b,f\rangle\}$, and
2. $\{\langle e,b,c\rangle,\langle e,b,f\rangle\}$.

So, these categories imply that the cases in each group compete for the same shared resources. However, the system MFD in Figure 9.3 shows that cases with prefixes $\langle a,b,c\rangle$ and $\langle e,b,c\rangle$ share the same conveyor (b,c) and compete for the same resource c, while prefixes $\langle a,b,f\rangle$ and $\langle e,b,f\rangle$ share (b,f) and compete for another resource f. So, the correct categories are

1. $\{\langle a,b,c\rangle, \langle e,b,c\rangle\}$, and
2. $\{\langle a,b,f\rangle, \langle e,b,f\rangle\}$.

The chosen metrics do not work for our example.

Moreover, it is not enough to encode only features of these categories because:

- the merge time at f depends on the current load and baggage spatial configuration on s, and
- the merge time at b depends on the current load on (b,c).

For example, if sorter s is full, waiting for merging at f can take minutes. In real MHSs, a greater number of dependencies usually exists among the system resources and conveyors.

To summarize, we identify the following drawbacks of the data-driven inter-case feature encoding for MHSs:

1. trace prefixes do not necessarily capture information about resources to compete for in near future,
2. start-time proximity does not necessarily capture any information related to the case interplay, and
3. the load in different parts of a process/system, that is not reached yet by the prefixes, can significantly affect target PPIs.

All things considered, we conclude that there is still a knowledge gap in the problem of inter-case feature encoding for PPM of systems with shared resources and queues. A method, bridging this gap, does not necessarily have to be data-driven. Instead, a knowledge-driven approach can address this problem, considering a given process model as the source of missing domain knowledge.

Next, we formulate the research question of this chapter, addressing the identified knowledge gap.

9.1.3 Research Questions

Before we formulate the corresponding RQ, we recap the thesis architecture. Figure 9.4 shows that this chapter addresses the final RQ of the thesis. As a result, all previously proposed models and methods are available for building on them. Thus, we can use

1. a complete and correct event table (see Chapter 7) as given event data, and
2. a PQR-system (see Chapter 6) as a given process model.

The former allows us to overcome the problem of MHS event data incompleteness (see Section 7.3.2), and the latter provides some domain knowledge for potential feature selection improvement. The RQ for this chapter reads as follows.

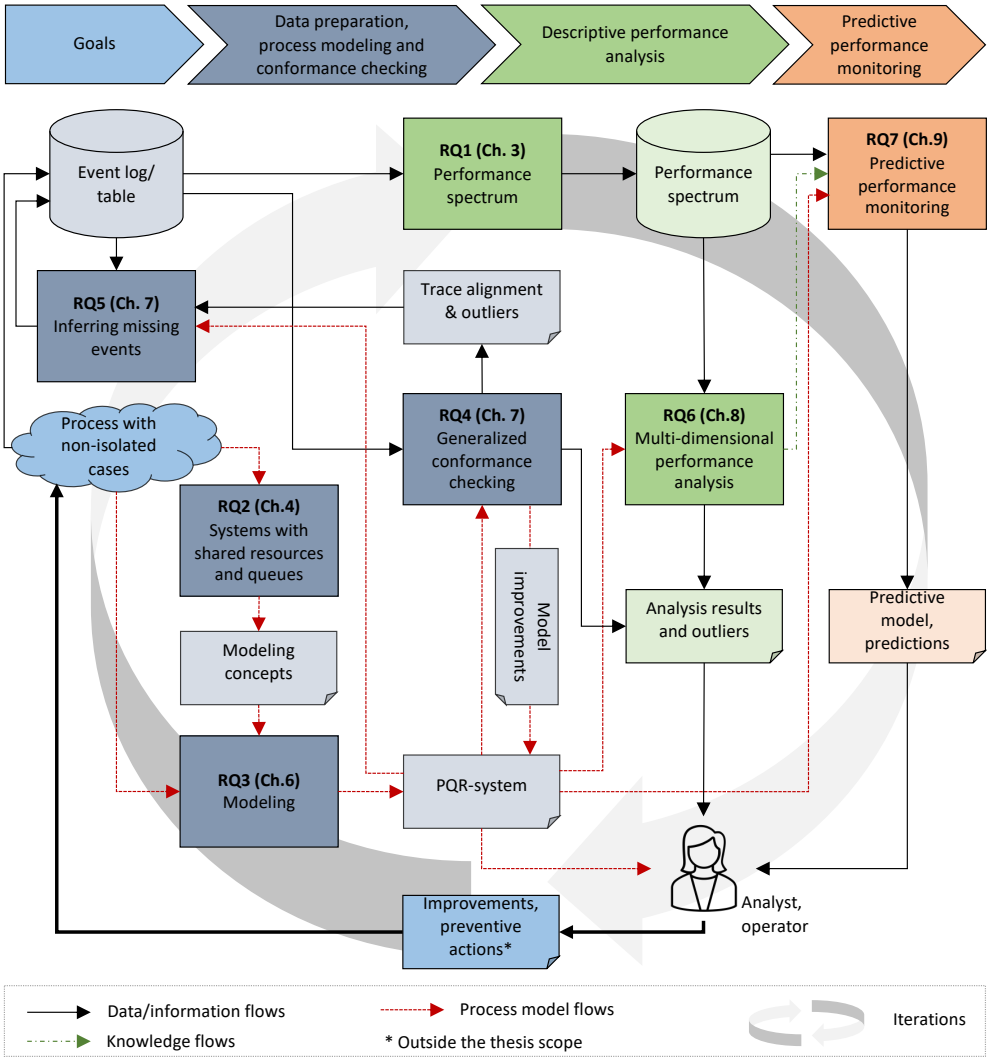


Figure 9.4: Thesis architecture.

- **RQ-7**. *Given a PQR-system, a complete event table, an aggregate process performance indicator, and a prediction horizon, how to predict this indicator so that both its gradual and sudden changes are predicted?*

Interestingly, by adding domain knowledge (PQR-system) to its input, we build to some extent on the previous version of [35], i.e., the knowledge-driven inter-case feature encoding method of [173].

9.1.4 Method Outline and Evaluation Results

In this section, we summarize the problem formulation for addressing **RQ-7**, method outline, and evaluation results.

In **RQ-7**, an event table is considered as input event data. However, formulating the problem for solving **RQ-7** over an event table would effectively ignore all the results of Chapters 3 and 8, that provided numerous insights about dynamics of systems with shared resources and queues, and the technique for capturing such dynamics (i.e.,, performance spectra). To avoid it, we consider how the outcome of these chapters can contribute to answering **RQ-7**. For that, we use an analogy with vehicle traffic to explain the intuition behind capturing system dynamics by performance spectra, in Section 9.2.1. Afterward, we consider a performance spectrum of a real BHS to explain the same for the MHS domain in Section 9.2.2. Finally, we formulate the problem over a multi-channel performance spectrum, and provide problem instance examples, in Section 9.2.3.

To solve the problem formulated in Section 9.2, we propose our method in Section 9.3. It takes a PQR-system, target PPI, prediction horizon, and multi-dimensional performance spectrum as input, to obtain an ML model for predicting the PPI. In a nutshell, this method uses the sliding window technique over the performance spectrum to extract a feature set. To apply this technique, parameters to extract the dependent variable and feature values in each window are defined first. We call them the target and historic spectrum respectively. The parameters of the former are defined by the analyst. The parameters of the latter are identified using:

- information about load and blockage propagation in the P-proclet of the given PQR-system, learned in Chapter 8, and
- temporal parameters of the Q- and R-proclets of the given PQR-system,

with respect to the given prediction horizon.

After obtaining the feature set, an ML model is learned, and its metrics are evaluated. Multiple iterations over the method steps may be required if the model metrics do not show the required prediction accuracy.

We evaluated our method with both synthetic and real datasets for two problem instances, defined in Section 9.2.4. We used the method in [35] as a baseline. Additionally, we introduced a naive baseline for the problem instance that could not be solved by [35]. For all methods, we learned linear and non-linear models and measured the root mean squared and mean absolute error. Additionally, we did meticulous residual diagnostics of obtained predictions.

The evaluation showed that our method outperformed the baseline methods with respect to the error values. Additionally, residual diagnostics showed that our methods could predict peaks and dips well for the simpler problem instance, but consistently underestimated their actual amplitude. We concluded that the obtained models

were technically sound and the method was feasible. We identify and discuss its limitations in Section 9.5.

9.2 Problem Formulation over Performance Spectra

In this section, we discuss why and how the problem of **RQ-7** can be formulated over performance spectra rather than over event tables. For that, we first provide some intuition about it by the example of a well-known problem of predicting Estimated Time to Destination (ETD) in navigation systems and link its solution with the performance description available from performance spectra in Section 9.2.1. Then, we show how performance spectra allow for inferring the near future performance by the example of the scenario observed in the real MHS, and what information in performance spectra is needed for that, in Section 9.2.2. Finally, we formulate **RQ-7** over performance spectra in Section 9.2.3.

9.2.1 Intuition behind Using Performance Spectrum for Capturing System Dynamics

In this section, we discuss how performance spectra capture system dynamics required for both *expressing* and *predicting* future dynamics. For that, before diving into the MHS domain, we provide some intuition by the example of a well-known problem of predicting ETD in navigation systems like Google Maps. We discuss how this problem can be converted to the problem of predicting aggregate PPIs over many vehicles, which is similar to **RQ-7**, and which information is required to express and predict such PPIs. Finally, we show how to obtain this information from performance spectra.

Let us plan a route from the RWTH Aachen University to the Eindhoven University of Technology using a popular navigation application. The result in Figure 9.5(a) shows that the ETD is one hour and 27 minutes. It is rather slow for a distance of 100 kilometers. However, the application explained the reason for the delay. In Figure 9.5(a), one part of the route is highlighted in red, indicating congestion. If we zoom in on the map (see Figure 9.5(b)), we see that near Echt “normal” speed (shown in blue) is gradually changing to slower (orange), and finally to slow (red). The same is shown again before the bridge across Maas near Roermond.

Let us reformulate this navigation problem in the terms of MHSs. Assuming the movement of a vehicle from its starting point to its destination as a case, the navigation software predicts its time to complete (case-level prediction). For that, information about the current situation on the roads, and historic information about it can be used.

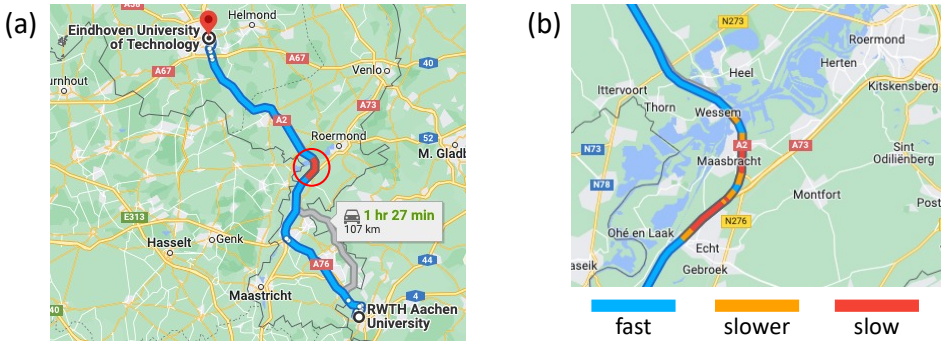


Figure 9.5: Route and the estimated time of arrival (a) based on information about congestion (b).

Although we do not know the exact method that is used to predict ETD in this application, we can hypothesize that it:

1. predicts dynamics on the route legs at different times from *now*, and
2. uses this information to estimate the vehicle speed on the entire route to compute the ETD.

However, what forms the route leg dynamics? In highway traffic, most vehicles usually drive at (almost) the same speed that is close to the speed limit, while some outliers like slow-moving trucks or speeding cars are also possible. So, to estimate traffic dynamics on a road leg, one most probably would describe the speed ranges of driving cars with respect to the speed limit. For example, if 99% of cars before Echt (blue zone) are driving at 120 kilometers per hour, and 1% are driving significantly slower, the description of the dynamics is:

1. $\{(99\%, \textit{normal}), (1\%, \textit{slower})\}$,

Speed is classified into *normal* and *slower* with respect to the speed limit of 120 kilometers per hour. In contrast, the description of dynamics near Echt (red zone) is:

1. $\{(100\%, \textit{slow})\}$.

By predicting such dynamics along the route, the speed of most vehicles on a road leg at a particular time can be derived, and ETD for the whole route can be inferred. That is, the system-level predictions are used to predict a case-level metric.

However, what is required to make such predictions? Let us speculate on this using our observations.

1. The total number of vehicles (load) on a route leg affects the speed on this and the next legs (see the speed-density effect in Chapter 4).

2. The speed of vehicles (except outliers) on a route leg affects the speed on the preceding legs, e.g., slow traffic ahead delays traffic behind.
3. Additionally, information about traffic acceleration/deceleration helps infer future situations, e.g., deceleration over some periods of time often indicates congestion development.
4. Last but not least, the same information about traffic on incoming/outgoing roads helps reason about the situations around junctions.

Let us now link this information to performance spectra. For that, let us consider the combined spectrum of segment (a, b) in Figure 9.6, assuming it represents a one-way single-lane road leg between junctions a and b with the speed limit of 120 km/h. Each segment occurrence there represents a vehicle moving from point a to b, their

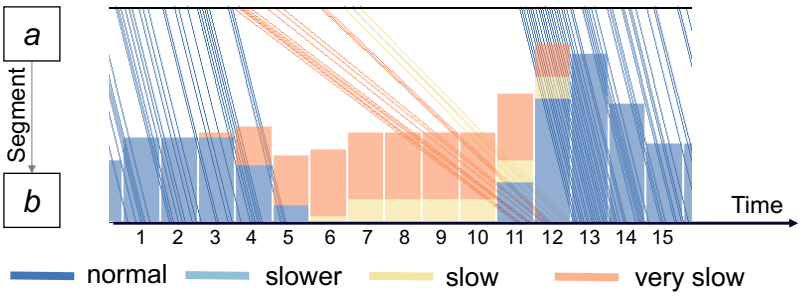


Figure 9.6: Combined performance spectrum shows the speed of vehicles in traffic on highway leg (a, b).

start and end time describe the travel time and speed (assuming the distance between a and b is known). The performance classes are defined as follows depending on speed v in km/h:

- *normal* if $v \geq 120$,
- *slower* if $60 \leq v < 120$,
- *slower* if $40 \leq v < 60$,
- *very slow* otherwise.

Each bin of the aggregate performance spectrum in Figure 9.6 shows the aggregated performance for one-minute time intervals, i.e., the number of vehicles for each performance class. For example, bin 11 describes 12 occurrences (vehicles) with class *very slow*, 4 occurrences with class *slower*, and seven with class *normal* (see grouping *intersect* in Chapter 3).

Now, we interpret this performance spectrum in the terms of road traffic. During the time intervals of bins 1 and 2, we observe some stable load and normal speed. Then, bins 2-6 show how traffic speed gradually decreases from normal to very slow.

Afterward, it slowly returns back to normal (bins 7-13). Additionally, we observe a load peak during the intervals of bins 11-14.

That is, this spectrum describes:

- static load and speed per vehicle performance class for a bin time interval in a single bin, and
- changes of speed and load per vehicle performance class over time in a series of consequent bins.

By providing performance spectra for the legs comprising the whole route, and additionally incoming/outgoing road legs, we describe the information required for predicting aggregate PPI over traffic.

Last but not least, each spectrum bin contains information for inferring the speed of most cars during a bin interval. For example, it is $v \geq 120$ for bins 1-4 and 12-15, and $v < 40$ for the others. Predicting how many cases have a particular class in a spectrum bin allows for solving the initial problem of ETD prediction.

To summarize, we show a simple example of how the performance spectrum expresses an aggregate PPI to be predicted and also captures the information for making predictions. Next, we consider how the same is applicable to MHSs.

9.2.2 Capturing MHS Dynamics Using Performance Spectra

In this section, we consider a PPM problem example for a large BHS of a major European airport. We show how it can be formulated over the performance spectra, and solved using the information in the real performance spectrum, computed from the event data of this BHS.

Let us first introduce the BHS. For clarity, we consider only the fragment related to the problem. It is a part of a preliminary sorting loop, shown in Figure 9.7. Baggage comes from check-in counters and is screened in the X-ray screening machine s. It is a mandatory step. Afterward, baggage that obtained clearance is diverted to the aircraft (exits $a_1 - a_3$), while "unsafe" bags are dumped out of the system (not shown in the MFD). If a bag, going from the check-in counters to s for screening, cannot be diverted to s because the corresponding conveyor entry is unavailable (e.g., occupied), it makes a full round on the sorter before another try. It is called *recirculation*. Note, this MFD omits some details and aggregates pieces of equipment for simplicity.

In BHSs, recirculation is highly undesirable because recirculating bags waste equipment capacity, and have greater chances of being late for the flight. So, its prediction and prevention are the actual problems of BHS support. Let us formulate the corresponding PPI as follows:

- the recirculation metric is the number of bags that started a recirculation round within the interval of 15 seconds.

Figure 9.7: MFD of a sorter loop fragment.

However, how to derive this metric from the performance spectrum? For that, we consider the real example of a combined performance spectrum in Figure 9.8.

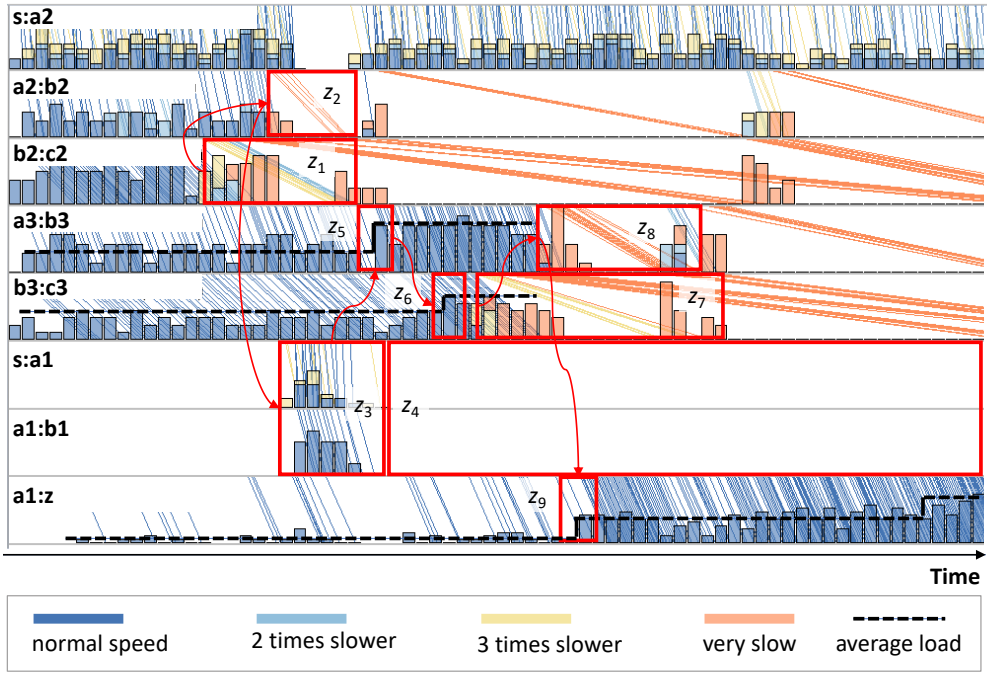


Figure 9.8: Performance dynamics at the locations of the sorter loop shown in Figure 9.7.

In this spectrum, each bin has a duration of 15 seconds, and the performance classes are defined and color-coded as shown in the legend at the bottom. Additionally,

- the real segment names are anonymized to not disclose sensitive information,
- some locations are aggregated into a single one for simplicity, so some segments can have intersecting occurrence lines as if bags were overtaking each other, and

- the segment occurrences are computed from incomplete event data exactly as they were recorded by the system, for avoiding any biases of log repair approaches.

As a result of the latter, the segments show some longer paths rather than paths between two neighboring locations in the MFD. For example, segment $(s, a2)$ "skips" location $a1$ in between. However, in this system under certain conditions, each bag is always registered at locations $a1$ and z. As a result, segment $(a1, z)$ effectively shows recirculation on the sorter. So, the total occurrence number of each bin of $(a1, z)$ is equal to the target PPI metric value.

Now, we consider what information in this spectrum can be used to predict this PPI in near future. For that, we interpret the scenario described by this spectrum, using "zones of interest", surrounded by red boxes in Figure 9.8.

1. Initially, the sorter operates normally, i.e., incoming bags are screened at s, and diverted at $a2$ or $a3$. Load is balanced between paths $a2-c2$ and $a3-c3$, while $a1-c1$ is not used.
2. Suddenly, segment $(b2, c2)$ gets blocked (spectrum zone z_1). z_1 is a blockage instance (see Chapter 8).
3. z_1 propagates backward in the control-flow direction to $(a2, b2)$ and causes z_2. As a result, baggage cannot be anymore diverted at $a2$.
4. To ensure load balancing, the system engages path $a1-c1$ (non-zero load in z_3).
5. However, soon this path becomes unavailable (zero load in z_4).
6. As a result, the full baggage flow starts going via $(a3, b3)$, causing doubled load on this segment (z_5). In Figure 9.8, the average load is shown by black dashed lines.
7. Higher load from $(a3, b3)$ propagates forward in the control-flow direction to $(b3, c3)$ (z_6).
8. Apparently, higher load cannot be handled by $(b3, c3)$ and causes another blockage instance z_7.
9. z_7 propagates backward to $(a3, b3)$ as z_8.
10. As a result, diverting at locations $a1-a3$ is impossible, so all bags start recirculating. In z_9 we can see how almost zero load showing recirculation dramatically increases because of z_8.

In this example, massive recirculation started in z_9 can be predicted as soon as blockage instance z_7 starts. Potentially, it can be predicted even earlier if z_7 was caused by the propagation of other blockage instances originating farther in the system segments.

Our example shows how a peak of recirculation in a BHS can be explained, and potentially predicted, using dynamics on related segments observed earlier. However, it operates on rather coarse pieces of information in the spectrum ($z_1 - z_9$) because

it is easier for human perception. At the same time, the spectrum provides quite fine-grained information. For example,

- inside z_1 speed of bags gradually drops over multiple bins,
- inside z_3 load drops to zero not instantly but over multiple bins,
- blockage instance z_7 was developing over multiple bins before triggering z_8 and z_9.

We assume that such fine-grained information can be used for:

- deriving accurate PPI values,
- increasing prediction accuracy,
- allowing a more distant prediction horizon, and
- allowing us to predict even small changes in dynamics, in contrast to the dramatic change observed in z_9.

As an example of the latter, a relatively small increase or decrease of the load on one segment can lead to a small increase or decrease of the load on another segment, allowing for accurate predictions of the corresponding PPIs.

To summarize, information in the aggregate performance spectrum bins allows for more fine-grained information than, for example, performance patterns detected in "regular" performance spectra. All things considered, we conclude that aggregate performance spectra capture the information required for both:

- deriving aggregate PPIs over the performance of many cases, and
- capturing the system/process dynamics required to predict such PPIs.

In the next section, we formulate the problem for answering **RQ-7** over aggregate performance spectra instead of event tables.

9.2.3 Formulation of the Problem over Multi-Channel Performance Spectrum

Now, we recap what a multi-channel performance spectrum is, and discuss why it is a richer source of information about system/process dynamics than "just" an aggregate (i.e., single-channel) spectrum. Afterward, we formulate the problem of this chapter over multi-dimensional performance spectra, and introduce two instances of this problem, to be used as examples and for the evaluation in the remainder of the chapter.

System Dynamics in Multi-Channel Performance Spectra. A multi-channel performance spectrum comprises one or multiple aggregate performance spectra (see Definition 3.8), computed for the same segment series (segment Definition 3.5) and time period, but for different channels (see Section 3.2.3). A channel defines:

- performance classifier $\mathbb{C}$ (see Definition 3.3),
- grouping $g \in \{start, pending, end\}$ (see Definition 3.8),
- and period (bin length) p.

For example, the aggregate performance spectrum of the combined spectrum in Figure 9.6 can be seen as a multi-channel performance spectrum, define for segment series $SEG = \langle (a, b) \rangle$, some performance classifier $\mathbb{C}_1$, mapping occurrence duration into classes $\{normal, slower, slow, veryslow\}$, and channel $ch_1 = (\mathbb{C}_1, pending, 15 \text{ seconds})$. In ch_1, grouping *pending* means that each bin shows how many occurrences intersect it. However, this information may be insufficient for obtaining accurate dynamic description, and/or inferring a required PPI. Alternatively, channels with grouping *start* and *end*, showing how many occurrences start and end within each bin, can be defined for adding the corresponding aggregate spectra to the initial multi-dimensional spectrum. This information can be used to capture the number of cases (e.g., vehicles or bags) entering and leaving a segment (e.g., a road leg or BHS conveyor) respectively.

Additionally, other performance classifiers can be introduced for defining even more channels. For example, another performance classifier $\mathbb{C}_2$, mapping each occurrence to the flight number, can be introduced for a performance spectrum of a BHS. As a result, dynamics of any event attribute, available in the given event table, can be captured in a single multi-channel performance spectrum *uniformly* as a tensor (see Figure 3.6 in Section 3.2.3).

In the following, we explain how to formulate the problem over a multi-dimensional performance spectrum.

Problem Formulation. In research question **RQ-7**, the following input is provided:

- a PQR-system S,
- a complete event table $ET = (CN_{\text{PQR}}, E, \#)$ obtained by applying generalized conformance checking,
- a prediction horizon $t_{\text{ph}} \in \mathbb{T}$, and
- a target PPI such that its value can be derived from $ET = (CN_{\text{PQR}}, E, \#)$.

To formulate the corresponding problem over a multi-dimensional performance spectrum instead of ET, we require that this spectrum is computed beforehand from ET, for example, by the analyst during the training phase and by software during the predicting phase.

. For that, a start-only process event log L^{ET}_{pid} (see Definition 8.1) is derived first. Then, the analyst defines what multi-dimensional performance spectrum is to be computed from this log by defining:

- a segment series SEG,
- a channel sequence CH, and
- a time interval $[s, e], s, e \in \mathbb{T}$.

As a result, multi-channel performance spectrum $\mathbb{APS}_{L^{ET}_{\text{pid}}}(SEG, CH, [s, e])$ is computed to replace the given event table in the input.

Along the dimensions of $\mathbb{APS}_{L^{ET}_{cn}}(SEG, CH, [s, e])$, the problem is to predict the performance characteristics for the segments of interest for a time-interval in the future, which we call a *target spectrum*, based on the performance of relevant segments during a recent time interval, which we call a *historic spectrum*. An estimate for the given PPI is to be derived by aggregating the performance-related features of the target spectrum.

Using the sliding window technique [55], we assume a time window for the bins of $\mathbb{APS}_{L^{ET}_{cn}}(SEG, CH, [s, e])$. In this window, we introduce bin indices such that index 0 points to a bin corresponding to time *now*, bins with negative indices correspond to the observed spectrum in past, while bins with indices > 0 correspond to unobserved yet bins in the future. Then, a target spectrum is specified by

- a segment series $SEG_{\text{t}} =\subseteq SEG$,
- a bin interval $[s_{\text{t}}, e_{\text{t}}]$, where the boundaries $e_{\text{t}} > s_{\text{t}} > 0$ define the *offsets* from the current bin *now* (with index 0) to the right in the sliding window, and
- target channels $CH_{\text{t}} \subseteq CH$.

That is, the target spectrum points to future bins. Index e_{t} defines the *prediction horizon* t_{ph}.

A historic spectrum is specified by

- a segment series $SEG_{\text{h}} =\subseteq SEG$,
- a bin interval $[s_{\text{h}}, e_{\text{h}}]$, where the boundaries $s_{\text{h}} < e_{\text{h}} \leq 0$ define the *offsets* from the current bin *now* to the left in the sliding window, and
- historic channels $CH_{\text{h}} \subseteq CH$.

In contrast to the target spectrum, the historic spectrum point to historic bins in past. In general, these spectra can be defined over different channels, i.e., it is possible that $CH_{\text{h}} \cap CH_{\text{t}} = \emptyset$.

In Figure 9.9, the target and historic spectra are shown for a sliding window of a single channel. Both spectra contain multiple segments and bins. The prediction horizon t_{ph} is s_{t}, and the historic spectrum includes bins from *now* till s_{h}. Similarly, the target and historic spectra can be defined for the other channels.

As a result, the problem can be formulated as a *regression problem*, using the historic and target spectra as the sources of independent and dependent variables over a common time parameter T. It is formalized in Equation 9.1 as follows:

$$\mathbb{APS}_{L^{ET}_{cn}}(SEG_{\text{t}}, CH_{\text{t}}, [s_{\text{t}} + T, e_{\text{t}} + T]) = \rho(\mathbb{APS}_{L^{ET}_{cn}}(SEG_{\text{h}}, CH_{\text{h}}, [s_{\text{h}} + T, e_{\text{h}} + T])) + R, \quad (9.1)$$

or $\mathbb{APS}_{L^{ET}_{cn},t}(T) = \rho(\mathbb{APS}_{L^{ET}_{cn},h}(T)) + R$ for short. Function ρ *predicts* the values of the target spectrum, and R is a residual, i.e., the deviation between the observed and predicted values.

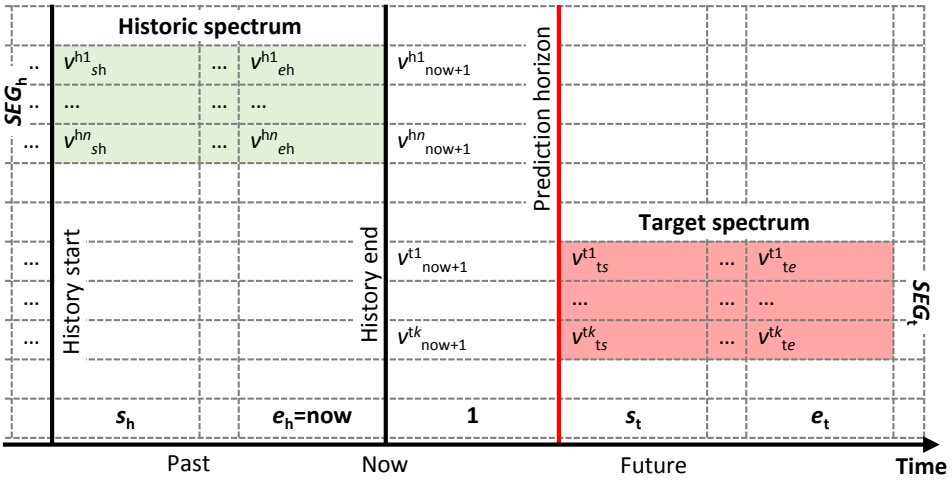


Figure 9.9: Historic and target spectra "around" the current time of the sliding window.

To learn function ρ, we use the sliding window method [55] for selecting w samples $(\mathbb{APS}_{L_{cn}^{ET},\mathrm{h}}(T_i), \mathbb{APS}_{L_{cn}^{ET},\mathrm{t}}(T))$ of the historic and target spectrum for time $T_1,\ldots,T_w$, and apply a ML technique to learn function ρ from these samples. By comparing the actual values $y = \langle \mathbb{APS}_{L_{cn}^{ET},\mathrm{t}}(T_1),\ldots,\mathbb{APS}_{L_{cn}^{ET},\mathrm{t}}(T_w)\rangle$ in the target spectrum with the values predicted by the learned function ρ, $y' = \langle \rho(\mathbb{APS}_{L_{cn}^{ET},\mathrm{h}}(T_1)),\ldots,\rho(\mathbb{APS}_{L_{cn}^{ET},\mathrm{h}}(T_w))\rangle$, we can estimate the prediction error R in ρ by a function $error(y,y') \in \mathbb{R}$.

In general, the target spectrum does not contain the target PPI directly but contains performance-related features sufficient to compute it. For that, we define the following function:

$$ppi(T) = g(\mathbb{APS}_{L_{cn}^{ET}}(SEG_\mathrm{t}, CH_\mathrm{t}, [s_\mathrm{t}+T, e_\mathrm{t}+T])) + \varepsilon, \tag{9.2}$$

where error $\varepsilon = ppi(T) - ppi'(T)$ and $ppi'(T)$ is the predicted target PPI observed over interval $[s_\mathrm{t}, e_\mathrm{t}]$. Next, we provide the instances of this problem, actual for the MHS domain.

9.2.4 Problem Instance Examples

We now instantiate the generic problem formulation from Equation 9.1 for the concrete real-life performance prediction problems of a major European airport BHS. The fragment of its simplified MFD is shown in Figure 9.10. We formulate the problem instances in the terms of the MFD segments. As input, we assume an event log L_{pid}^{ET}, recorded for case notion pid, containing all the segments of the MFD.

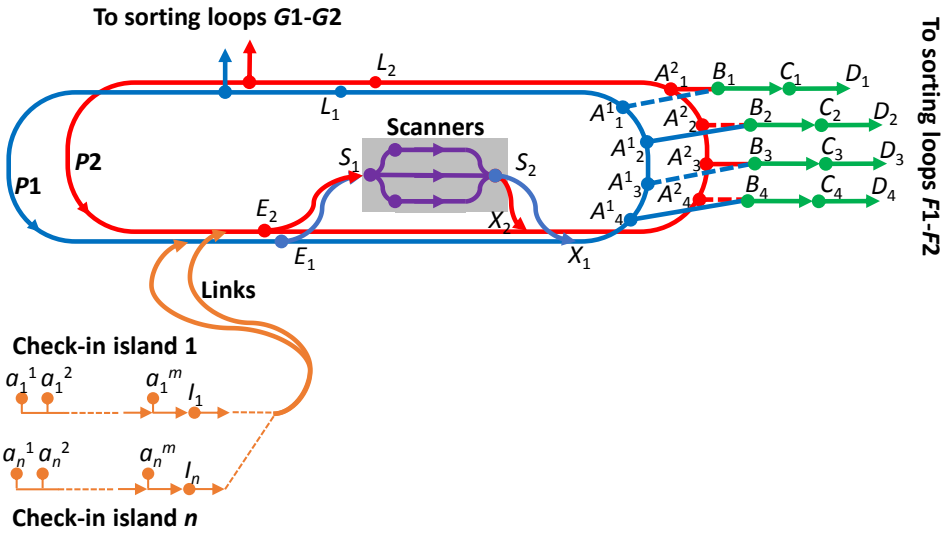


Figure 9.10: Check-in and pre-sorting areas.

Problem Instance 1 (PrIn_1). We first consider the bag handling process part from checking in until security screening. Bags enter the system via one of many check-in counters a_1^1-a_n^m, and then move via conveyor belts to one of two pre-sorter loops P1, P2 where each bag has to pass through one of the X-ray baggage screening machines, e.g., entering via (E_1, S_1) and leaving via (S_2, X_1). For operational support, the main concern is to keep the BHS performance steady at some desired level. In particular, the workload in a processing step or system part may not exceed its capacity, as this otherwise leads to long queues or stalling of the sorting loops. Here, workload *prediction* is central for proactive management. One concrete problem PrIn_1 is to predict the load (in bags per minute) at the X-ray baggage screening machines on $t_{\text{ph}} = 4$ minutes in advance for pre-sorter P1. In Figure 9.10, this load corresponds to the load on segment $SEG_{t,\text{PrIn}_1} = \langle (E_1, S_1) \rangle$.

To express this problem in the terms of Equation 9.1, we define the target and historic spectrum. First, to represent the load, we define a single performance spectrum channel with

- a trivial performance classifier $\mathbb{C}_{\text{PrIn}_1}$ to map each segment occurrence into the same performance class c (as load does not distinguish different classes),
- a grouping *start*, and
- a bin with length 60 seconds,

i.e., the resulting set CH_{PrIn_1} of the historic and target channels $CH_{\text{PrIn}_1} = \langle (\mathbb{C}_{\text{PrIn}_1}, start, 60) \rangle$. For event log L_{pid}^{ET}, the target spectrum for PrIn_1 is

- $PS_{\mathrm{PrIn}_1}(T) = \mathbb{APS}_{L^{ET}_{\mathrm{pid}}}(SEG_{t,\mathrm{PrIn}_1}, CH_{\mathrm{PrIn}_1}, [t_{\mathrm{ph}} + T, t_{\mathrm{ph}} + T])$.

The predicted load is the sum of all its values, in bags per minute. Then we make the following hypothesis: the load depends on the average load of the check-in counters in 1-3 minutes before *now*. To capture that, we include the check-in segments to the historic spectrum: $SEG_{h,\mathrm{PrIn}_1} = \langle (a^1_1, I_1), \ldots, (a^m_1, I_1), \ldots, (a^1_n, I_n), \ldots, (a^m_n, I_n) \rangle$ and time-interval $[s_{\mathrm{h}}{}^{\mathrm{PrIn}_1}, e_{\mathrm{h}}{}^{\mathrm{PrIn}_1}]$ as $[-3, -1]$. This leads to the following regression problem derived from Equation 9.1:

$$PS_{\mathrm{PrIn}_1}(T) = \rho_{\mathrm{PrIn}_1}(\mathbb{APS}_{L^{ET}_{\mathrm{pid}}}(SEG_{h,\mathrm{PrIn}_1}, CH_{\mathrm{PrIn}_1}, [T-3, T-1])) + R. \tag{9.3}$$

The target PPI is defined as $ppi_{\mathrm{PrIn}_1}(T) = PS_{\mathrm{PrIn}_1}(T)_1$, where the index means the aggregate of the only first performance class c of performance classifier $\mathbb{C}_{\mathrm{PrIn}_1}$.

Problem Instance 2 (PrIn_2). Another concern for the BHS operational support is predicting the risk that baggage being late for a flight, so we now instantiate Equation 9.1 for this problem. The second part of the process in Figure 9.10 moves bags from the screening machines to sorting loops $F1$ and $F2$ (exit the pre-sorters $P1$ and $P2$ via A^1_1-A^4_2, B^1_1-B^4_2). It may happen that, for instance, a bag on pre-sorter P1 that has to go to $F1$, cannot be diverted onto any of the conveyors (A^1_i, B_i) because they are unavailable, e.g., due to high load on all (C_i, D_i). In this case, the bag starts looping on P1 until it can be finally diverted successfully. Each round increases the bag *estimated time to destination* t_{est} by the loop duration t_P. If the new estimate $t'_{est} = t_{est} + t_p$ exceeds the deadline when the bag has to arrive at its destination to reach the flight, the bag is expected to be late and correcting actions, e.g., making the bag priority higher, can be undertaken.

So, to predict such late bags, it is sufficient to predict extra re-circulation due to the unavailability of diverts A^1_1-A^4_2. We formulate PrIn_2 as the problem of predicting this re-circulation for P1. On P1, any bag traveling the segment (A^1_1, L_1) is re-circulating (as it could not be diverted to $F1$ or $F2$), thus the segments of the target spectrum are $SEG_{\mathrm{t}} = \langle (A^1_1, L_1) \rangle$. Selecting $t_{\mathrm{ph}} = 60$ seconds, duration-based classifier $\mathbb{C}_{\mathrm{PrIn}_2}$ (whether $t = t_b - t_a$ is in the 25%-quartile of the histogram $H(a, b, L)$), and $CH_{h,\mathrm{PrIn}_2} = \langle (\mathbb{C}_{\mathrm{PrIn}_1}, start, 30), (\mathbb{C}_{\mathrm{PrIn}_2}, pending, 30) \rangle$, we make a hypothesis that the target spectrum depends on the load and delays of $SEG_{h,\mathrm{PrIn}_2} =$

- $\langle (S_2, X_1), (S_2, X_2), (A^1_i, B_i), (A^2_i, B_i), (B_i, C_i), (C_i, D_i) \mid i = 1, \ldots, 4 \rangle$

for two bins before *now*. We predict the target spectrum, defined for a channel set $CH_{t,\mathrm{PrIn}_2} = \langle (\mathbb{C}_{\mathrm{PrIn}_1}, start, 30) \rangle$, $PS_{\mathrm{PrIn}_2}(T) = \mathbb{APS}_L(SEG_{t,\mathrm{PrIn}_2}, CH_{t,\mathrm{PrIn}_2}, [T+1, T+1])$ as follows:

$$S_{\mathrm{PrIn}_2}(T) = \rho_{\mathrm{PrIn}_2}(\mathbb{APS}_L(SEG_{h,\mathrm{PrIn}_2}, CH_{h,\mathrm{PrIn}_2}, [T-2, T-1])) + R. \tag{9.4}$$

The PPI is defined as $ppi_{\mathrm{PrIn}_2}(T) = \mathbb{APS}_L(SEG_{t,\mathrm{PrIn}_2}, CH_{t,\mathrm{PrIn}_2}, [T+1, T+1])_1 + \varepsilon$, i.e., we select the channel with grouping *start* and the first performance class in $\mathbb{C}_{\mathrm{PrIn}_1}$.

Next, we propose our approach for identifying the target and historic spectrum for a given problem instance.

9.3 Method for Predictive Performance Monitoring

In Section 9.8, we showed how the prediction of aggregate performance measures for systems with shared resources and queues, answering **RQ-7**, can be expressed as a generic regression problem over the multi-channel performance spectrum. Now, we present our method for solving it. We start by providing the method overview.

9.3.1 Overview

The method input, intermediate/final results, and main steps are shown in Figure 9.11. According to the problem formulation in Section 9.2.3, it takes as input:

1. a PQR-system S (the orange box in the diagram in Figure 9.11),
2. target PPI PPI_{t} to be predicted (the grey box on the top),
3. prediction horizon t_{ph} (the grey box on the left-hand side), and
4. multi-channel performance spectrum PS_{mc} (the blue box on the right-hand side).

The final result is the predictive model with metrics (the grey box at the bottom) to be deployed for the PPM of the system or process.

In a nutshell, our method uses the sliding window technique over PS_{mc} to extract the values of features and dependent variables from the historic and target spectra in each window position. For that, the method comprises the following steps.

1. **Step 1. Define target spectrum parameters.** First, the analyst defines which channels, bins, and segments are required for deriving the value of PPI_{t} from PS_{mc}. The output of this step is the target spectrum parameters (see Figure 9.9):
   - target channels CH_{t},
   - target segments SEG_{t}, and
   - a target bin interval $[s_{\mathrm{t}}, e_{\mathrm{t}}]$.
2. **Step 2. Define target spectrum parameters.** In this step, the analyst defines the same parameters for the historic spectrum. Potentially, an enormous number of features can be derived from PS_{mc}. A large number of features would lead to a feature explosion problem, and an inability to obtain a model with the required accuracy. So, the main challenge is to identify a historic spectrum that includes only information relevant for predicting PPI_{t}. Reasoning about load and blockage propagation paths and speed, provided in Section 9.3.3, is used to define such a spectrum. The output of this step is the historic spectrum parameters (see Figure 9.9):

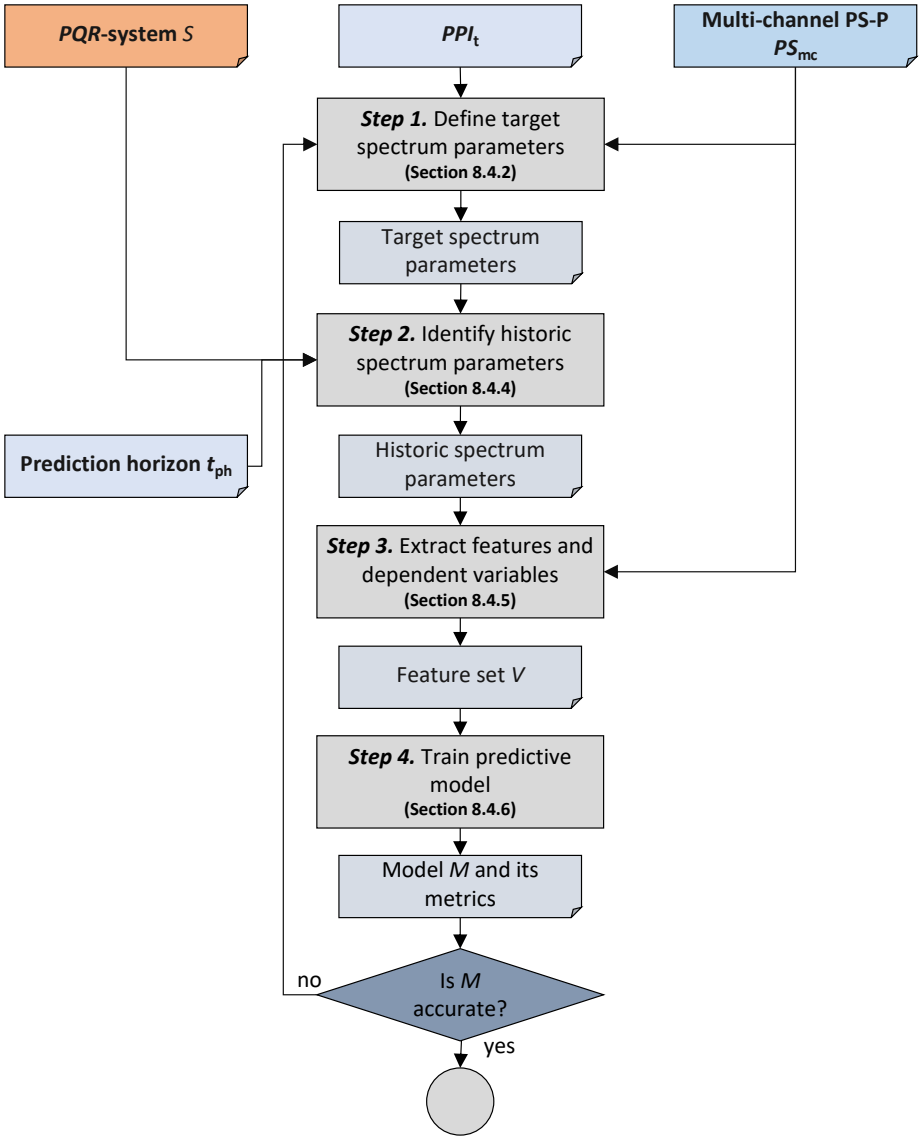


Figure 9.11: Method for PPM.

- target channels CH_{h},
- target segments SEG_{h},
- and target bin interval $[s_{\mathrm{h}}, e_{\mathrm{h}}]$.

Note, that the choices of the channels and segments for the target and historic spectra are independent, and completely different channels or segments can be used. That is, it is possible that $CH_t \cap CH_h = \emptyset$, and $SEG_t \cap SEG_h = \emptyset$.

3. **Step 3. Extract the values of features and dependent variables.** After the target and historic spectra parameters are defined, the sliding window technique can be applied. For each window, those parameters are used to instantiate the target and historic spectra and extract their bin values as the feature and dependent variable values. As a result, a feature set V is created.
4. **Step 4. Train predictive model.** Finally, set V can be divided into training and test sets for training a predictive model M. The analyst chooses a desired model architecture and hyper-parameters. Afterward, a standard ML pipeline is used for training. If the resulting model metrics indicate an accurate model, M is the final result. Otherwise, Steps 1-4 can be repeated to obtain a more accurate model.

Next, we describe all the steps in more detail.

9.3.2 Step 1. Define Target Spectrum Parameters

The target PPI is given as input. However, we need to define

1. a target spectrum in the sliding window (see Figure 9.9) for obtaining the training and test sets for the model training, and
2. function g for the model evaluation and use (see Equation 9.2).

For defining a target spectrum, the analyst determines the following parameters of the target spectrum in the sliding window:

1. target start and end indices $s_t, e_t \in \mathbb{N}, s_t > e_t, s_t = t_{ph}$ for defining the target spectrum interval $[s_t, e_t]$,
2. target segments $SEG_t \subseteq SEG$, and
3. target channels $CH_t \subseteq CH$.

Afterward, conversion function g is defined over the target spectrum and common time parameter T. In general, the way of defining the target spectrum and function g depends on the target PPI, as various PPIs require different information to be computed. Nevertheless, the following guidelines can be used by the analyst.

1. Channels CH_t and segments SEG_t should be defined according to Occam's razor principle to minimize the amount of information to be predicted, i.e., the minimal number of segments and channels with simplest classifiers are to be defined. For example, if just one segment and one channel with a trivial (single-class) classifier are sufficient to compute the target PPI, no extra segments or channels should be included in the target spectrum.

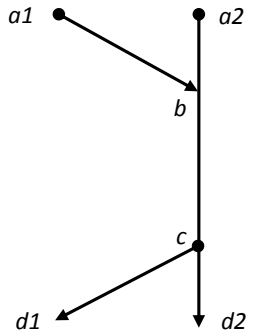


Figure 9.12: BHS material flow diagram.

2. The duration of the bin interval $[s_t, e_t]$ is to be chosen according to the system dynamics in the target segments. For example, if a load peak can be reliably detected within a time window of a particular length l, the length of $[s_t, e_t]$ should not be less than l. The exploitative analysis of PS_{mc} can help discover such characteristics if they are unknown beforehand.
3. Error ϵ should reflect the potentially achievable accuracy. That is, the information needed to compute PPI_t is either directly extraditable from the aggregate performance spectrum, or ϵ should be large enough to take, for example, possible noise into account. For instance, if only the part of the bags passing through some target segment seg_t is relevant to PPI_t, the other bag occurrences in the target spectrum (i.e., noise) can impede the accuracy of the PPI estimation. The given PQR-system is referred to for exploring bag flows through the system segments.

9.3.3 Origins of the Performance on Target Segments

In this section, we provide some intuition behind our approach for historic spectrum identification (Step 2). It is based on our observations about high load and blockage propagation in Chapter 8. In the following, we introduce a running example and consider how knowledge about the propagation of load and blockages allows for estimating performance along their propagation paths.

Running Example. Let us briefly recap the running example of Section 8.3, which we use in this section as well. A toy BHS, moving baggage from check-in counters $a1, a2$ via merge unit b and diverting unit c toward exits $d1, d2$ is shown in Figure 9.12. Despite the simplicity, this BHS has two "main" building blocks of MHSs — merge and diverting units – and can demonstrate typical MHS behaviors well.

To compute its multi-channel performance spectrum, we introduce

- a bin size $p \in \mathbb{T}$,
- a trivial performance classifier $\mathbb{C}_1$ that maps occurrences to the same performance class *normal speed*,
- a performance classifier $\mathbb{C}_2$ that maps occurrences to either *normal speed* or *slow speed*,
- a channel $ch_1 = (\mathbb{C}_1, start, p)$, and
- a channel $ch_2 = (\mathbb{C}_2, pending, p)$.

The performance spectra for ch_1 and ch_2 are shown in Figure 9.13(a) and Figure 9.13(b) respectively. They show the lines of occurrences for clarity but do not show bins that we do not consider in this section for simplicity. Additionally, these figures show detected high load instances $hl_0 - hl_5$ and blockage instances $bl_1 - bl_5$. We refer to Section 8.3.1 and Section 8.3.4 (Figure 8.16) respectively for details about the scenario which this spectrum is computed for, and the propagation chain discovered in this spectrum.

To discuss how a historic spectrum can be identified, we choose a PPI showing *how many bags entered conveyor* (b_s, c_s) *during time window* p. We call this PPI *segment load*. As the BHS is small, we choose prediction horizon $t_{\text{ph}} = 0$, i.e., we want to predict the PPI for a time window that starts at time *now*.

Then, we define a target spectrum. We choose target segments $SEG_{\text{t}} = \{(b_s, c_s)\}$, and target channel $CH_{\text{t}} = \{ch_1\}$ because the PPI can be derived from the total number of occurrences in bins of (b_s, c_s) in ch_1 of the given multi-channel performance spectrum. The indices are $s_{\text{t}} = e_{\text{t}} = 1$ because (1) $t_{\text{ph}} = p$, and (2) the PPI time window is p.

In the following, we consider sliding windows $sw_1 - sw_3$ over the spectrum in Figure 9.13 to discuss where the load on SEG_{t} comes from, and include its origins in the historic spectrum.

Identifying Historic Segments Where Load Propagates From. We start by considering sliding window sw_3 in Figure 9.13(a) first because it shows how high load instances hl_1 and hl_2 propagate to the target segment as hl_3, i.e., high load propagation that is already discussed in Section 8.3.4 in detail. In sw_3, bins with indices $-1, 0$ correspond to the observed spectrum till time *now* with respect to the sliding window, and a bin with index 1 corresponds to the future spectrum on the given prediction horizon. Bin 1 of (b_s, c_s) in ch_1 is the target spectrum. It shows that two occurrences o_8 and o_9 start in this bin, so the PPI value is *two* bags.

However, which spectrum "zone" in past does this load originate from? We already know that:

- high load propagates forward in the control-flow direction, and
- hl_3 is the result of propagation of hl_1 and hl_2 (see propagation links $link_5$ and $link_6$ in Figure 8.16).

In the spectrum, occurrences o_8 and o_9 in the target spectrum are of the same cases (bags) as o_6 and o_7 in bin 0 of the observed spectrum in segments $(a1_s, b_s)$ and

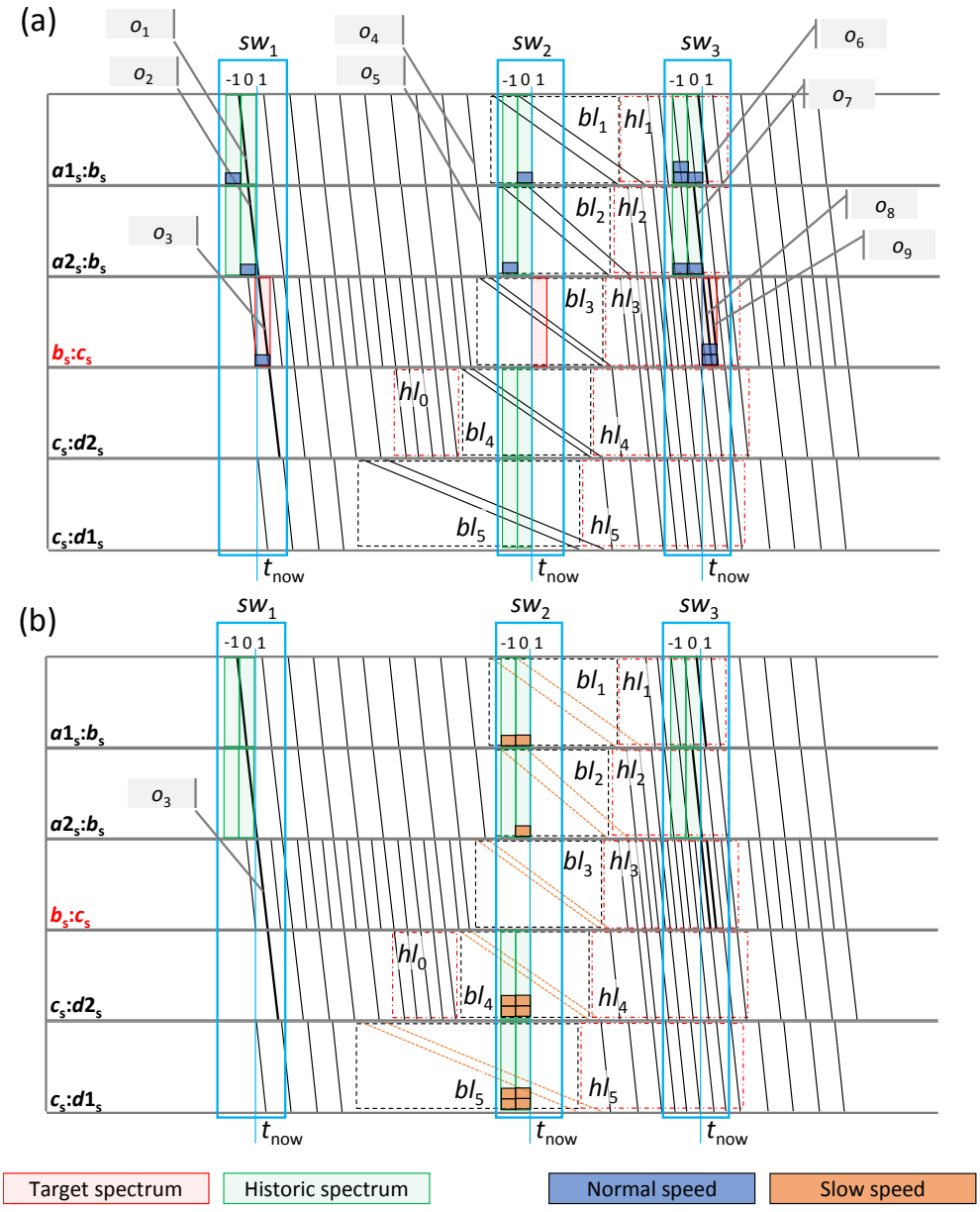


Figure 9.13: Sliding windows with target and historic spectra over channels ch_1 (a) and ch_2 (b).

$(a2_s, b_s)$. That is, some "pieces" of hl_1 and hl_2 comprise a "piece" of hl_3 because they propagated in the control-flow direction via merge unit b to the target segment. We conclude, that in case of high load on a target segment, high load propagation rules can be used to determine the sources of high load in the observed spectrum, to determine the corresponding historic segments. In our example, because load to (b_s, c_s) comes from $(a1_s, b_s)$ and $(a2_s, b_s)$, we define the historic spectrum parameters as:

- $SEG_{\text{h}} = \{(a1_s, b_s), (a2_s, b_s)\}$,
- $s_{\text{h}} = e_{\text{h}} = 0$,

The time interval $[s_{\text{h}}, e_{\text{h}}]$ includes just a single bin. We add one more bin to capture the dynamics in case of delays, i.e., we define:

- $s_{\text{h}} = -1$,
- $e_{\text{h}} = 0$.

Note, we do not consider ch_2 so far because it is not needed yet (i.e., in sw_3) to estimate the target spectrum.

Now, let us also consider the load that is not high load. Does it propagate in the same way as high load? To understand it, we consider sliding window sw_1 in Figure 9.13(a), where no high load instances are detected. The target and performance spectra are defined according to the parameters above. In the target spectrum bin, just one occurrence o_3 starts, so the PPI value is *one*. It is the result of the propagation of o_2 from the historic spectrum bin 0 of segment $(a2_s, b_s)$. At the same time, bin 0 of another historic segment $(a1_s, b_s)$ has no occurrences that start there.

So, the PPI value is two times less than in sw_3 because the load on the historic segment is also two times less. We conclude that non-high load propagates in the same way as high load. As a result, our approach for high load propagation tracking can be used for identifying where the load on a segment originates from.

Identifying Historic Segments Where Blockages Propagate From. Now, we consider blockage instances in the target segment. What historic segments are needed for predicting load when blockage instances occur on target segments? For that, we consider sw_2 in Figure 9.13(a), when blockage instances $bl_1 - bl_5$ are observed on all the segments. The target bin has zero occurrences starting there, so the PPI value is *zero*. However, the historic spectrum segments $(a1_s, b_s)$ and $(a2_s, b_s)$ have such occurrences. If a prediction were made using just information from these two segments, the load would be most probably *two* since two occurrences start in bin 0 of these segments. However, in reality $bl_1 - bl_3$ prevent this load from propagation to (b_s, c_s). So, blockage propagation should be additionally considered to enable more accurate predictions.

As we discussed in Chapter 8, blockage propagates backward in the control flow direction. That is, in our example, it propagates from the outgoing segments of (b_s, c_s).

These are $(c_s, d1_s)$ and $(c_s, d2_s)$, where instances bl_4 and bl_5 are observed for bins $-1, 0$ and 1 (see propagation links $link_3$ and $link_4$ in Figure 8.16). Thus, we additionally include these segments in the historic spectrum:

- $SEG_{\text{h}} = \{(a1_s, b_s), (a2_s, b_s), (c_s, d1_s), (c_s, d2_s)\}$.

However, zero occurrences start in the historic spectrum bins of these segments because occurrences comprising bl_4 and bl_5 intersect these bins but do not start within them. To capture blockage-related information, we need another channel. In ch_2 with grouping *pending*, we count how many occurrences *intersect* a bin, while $\mathbb{C}_2$ captures their speed class. As a result, bins $-1, 0$ of $(c_s, d1_s)$ and $(c_s, d2_s)$ capture information about bl_4 and bl_5. A predictive model can potentially use this information to infer that a blockage is observed on the target segment as well, and predict the target spectrum accurately.

Combining all the information in the historic spectrum of both channels, we capture both load and blockage propagation sources. However, in large systems, load and blockages can propagate through many system segments. How to determine what segments on the propagation paths are to be included in the historic spectrum? In the following, we discuss how the given prediction horizon and PQR-system help determine them.

Considering Prediction Horizon for Load Propagation. Over time, load and blockages can propagate through many segments. The longer is t_{ph}, the farther segments affect the target spectrum. How to identify which ones should comprise the historic spectrum? For that, the distance of load and blockage propagation within the prediction horizon interval is needed to be estimated.

To have a longer propagation path in our example, let us choose a father (from check-in counters $a1, a2$) segment $(c_s, d2_s)$ as a target one (see Figure 9.12), and a longer prediction horizon $t'_{\text{ph}} = 2p$.

In a PQR-system, the minimum load propagation time through a particular control-flow path in the P-proclet can be estimated using the minimum waiting and service time of the queues and resources along this path, known from the Q- and R-proclets. For example, let us consider how much time is required for the load comprised of bag $pid1$ to reach c_s from $a2_s$. For that, we follow its fastest path throughout the PQR-system of the BHS, shown in Figure 8.3.

1. Initially, $pid1$ is served by $rid2$ in the minimum service time $t_{\text{sR}}^{\{rid1, rid2\}}$.
2. Afterward, $pid1$ waits in $qid2$ for the minimum waiting time t_{wQ}^{qid2}.
3. When waiting is completed, $rid3$ picks $pid1$ up and serves it in t_{sR}^{rid3}.
4. Finally, waiting in $qid3$ for t_{wQ}^{qid3} is required to reach c_s.

Indeed, the fastest total time $t_f = t_{\text{sR}}^{\{rid1, rid2\}} + t_{\text{wQ}}^{qid2} + t_{\text{sR}}^{rid3} + t_{\text{wQ}}^{qid3}$ is comprised by the minimum service and waiting times of the resources and queues on the way.

In this equation, time $t_{\text{sR}}^{\{rid1,rid2\}} + t_{\text{wQ}}^{qid2}$ corresponds to the minimal occurrence duration of $(a2_s, b_s)$, and $t_{\text{sR}}^{rid3} + t_{\text{wQ}}^{qid3}$ corresponds to the minimal occurrence duration of (b_s, c_s). Let us assume that $t_f = t'_{\text{ph}}$. In this case, load from $(a2_s, b_s)$ can reach the target segment in t'_{ph}, so the historic spectrum should include $(a2_s, b_s)$. Similarly, we include $(a1_s, b_s)$ because $t_{\text{wQ}}^{qid1} = t_{\text{wQ}}^{qid2}$, so load from $(a1_s, b_s)$ reaches (b_s, c_s) in the same time t'_{ph}.

Note, that only the fastest possible time can be estimated in this way. In reality, it can take longer time in case of delays, i.e., a longer waiting and/or service time. To consider "delayed load" as well, extra bins for earlier time periods, that are farther in time from *now*, can be included in the historic spectrum.

An example of the resulting historic spectrum is shown in Figure 9.14 in sliding window sw_4, where:

- the historic spectrum includes two bins -1 and 0 of segments $(a1_s, b_s)$ and $(a2_s, b_s)$,
- the target spectrum in segment $(c_s, d2_s)$ is in bins 2, according to $t'_{\text{ph}} = 2p$.

So far, we discussed how to determine segments whose load affects the target spectrum within the prediction horizon. Next, we consider how to estimate blockage propagation time.

Considering Prediction Horizon for Blockage Propagation.

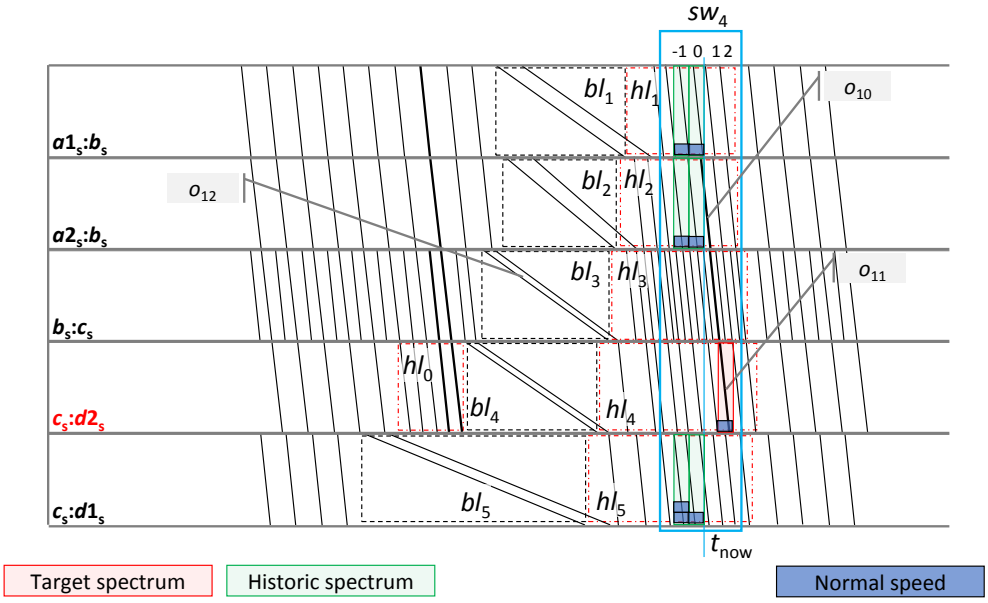


Figure 9.14: Target and performance spectra in sliding windows sw_4 of channel ch_1.

Let us finally discuss the blockage propagation time. While load propagates as the corresponding cases progress within the process or materials (bags) move through the system, blockage propagates differently. In an MHS, when a conveyor gets blocked, this applies to all the TSU on this conveyor *instantly*. Afterward, this blockage can propagate to incoming conveyors at a moment when a TSU fails to be handed over onto this conveyor. As a result,

- blockage propagation can be much faster than load propagation,
- propagation time depends on the presence of cases along the propagation path.

For example, in Figure 9.14 blockage instance bl_3 starts when occurrence o_{12} could not be handed over to already blocked $(c_s, d2_s)$.

To estimate blockage propagation time via a PS-P segment corresponding to a resource with rid_i and queue qid_j, we introduce a coefficient $k, 0 < k \leq 1$. Using k, we obtain the blockage propagation time as $k(t_{\text{sR}}^{rid_i} + t_{\text{wQ}}^{qid_j})$. This time is always less than or equal to the minimum load propagation time for the same segment.

So far, we provided some intuition of how to determine what segments and bins comprise the historic segment. Next, we formulate sub-steps to determine them for the given input.

9.3.4 Step 2. Identify Historic Spectrum Parameters

In this step, we identify a historic spectrum in a sliding window. It is defined by:

- historic channels CH_{h},
- bin interval $[s_{\text{h}}, e_{\text{h}}]$,
- and historic segments SEG_{h}.

Step 2.1. Identify Historic Channels and Bin Intervals. For our method, we use all channels of the given spectrum, i.e.,

- $CH_{\text{h}} = CH$.

Then, we compute the right border as an integer number of bin duration p in the prediction horizon, i.e.,:

- $e_{\text{h}} = \lceil \frac{t_{\text{ph}}}{p} \rceil$.

To capture dynamics in case of delays, we add an extra bin from the past, i.e.,

- $s_{\text{h}} = e_{\text{h}} - 2$.

It allows for capturing information about occurrences if load propagation takes a longer time. However, a smaller or greater number of bins can be used as well if needed.

Finally, we identify what segments are to be in SEG_{h}, using the intuition provided in the previous section. For that, we iterate over each target segment $seg_{\text{t}} \in SEG_{\text{t}}$, and identify historic segments where

1. load originates from, and
2. blockages originate from.

All the identified segments form the final historic segment set. In the following, we consider these two sub-steps in more detail.

Step 2.2. Identify Segments Where Load Originates From. In this sub-step, we identify a set of PS-P segments such that load propagates from each one of them to SEG_{t} within t_{ph}. For that, we consider possible control-flow paths in the P-proclet that end at seg_{t}, and choose paths where seg_{t} is reachable within t_{ph}. To estimate the time, we use the service and waiting time of the R- and Q-proclets that synchronize with the transitions comprising the path. The first segment of such a path is a historic segment. The definition of this set reads as follows.

Definition 9.1 (Set of load-related historic segments)**.** *Let*

- *PQR-system* $S = (\mathcal{N}, m_v, C, \ell^C)$ *(Definition 6.7),*
- *its P-proclet* $N_0 = (P_0, T_0, F_0, \ell_0, Var_0, colSet_0, m_0, arcExp_0) \in \mathcal{N}$ *(Definition 6.2),*
- *function* $pqrStr\colon T_0 \times T_0 \to \mathbb{T}$ *that maps transitions* $t_i, t_j \in T_0$ *to*
  - *the minimum service time* t_{sR} *of a R-proclet* $N_r \in \mathcal{N}$ *(Definition 6.6) if the* start *and* complete *transitions* $t_{start}, t_{complete} \in T_r$ *synchronize with* t_i *and* t_j *respectively, i.e.,* $(t_i, t_{start}), (t_j, t_{complete}) \in C$*,*
  - *and to zero otherwise,*
- *function* $pqrWtq\colon T_0 \times T_0 \to \mathbb{T}$ *that maps transitions* $t_i, t_j \in T_0$ *to*
  - *the minimum waiting time* t_{wQ} *of a Q-proclet* $N_q \in \mathcal{N}$ *(Definition 6.5) if the* enqueue *and* dequeue *transitions* $t_{enq}, t_{deq} \in T_q$ *synchronize with* t_i *and* t_j *respectively, i.e.,* $(t_i, t_{enq}), (t_j, t_{deq}) \in C$*,*
  - *and to zero otherwise.*
- *target segment* SEG_{t}*, comprised by two transitions in* T_0*,*
- *and prediction horizon* $t_{\text{ph}} \in \mathbb{T}$*.*

The set of load-related historic segments *is*

- $\{(\ell_0(t_0), \ell_0(t_2)) \mid t_0, t_2 \in T_0$*, and exists sequence* $\sigma = \langle t_0, t_1, t_2, t_3, \ldots, t_{2n} \rangle$ *of transitions in* T_0 *such that:*
  1. *it includes at least five transitions required to comprise one historic and one target segment, i.e.,* $n \geq 2$
  2. *it describes a valid path in the P-proclet in the control-flow direction, i.e.,* $\forall 0 \leq j < 2n, \exists p \in P_0, (t_j, p), (p, t_{j+1}) \in F_0$*,*

3. *no loops are possible in* σ*, i.e.,* $\forall 0 \le i < j \le 2n, t_i \neq t_j$,
4. *the last and third transitions from the end form the target segment, i.e.,* $(\ell_0(t_{2n-2}), \ell_0(t_{2n})) = SEG_t$,
5. *the total duration* $pathDur \in \mathbb{T}$ *of the resource service time (Definition 6.6) and queue waiting time (Definition 6.5) on the path described by* σ*, excluding its tail corresponding to the target segment, is on or beyond the prediction horizon, i.e.,*
   $pathDur = \sum_{i=0}^{n-2}(pqrStr(t_{2i}, t_{2i+1}) + pqrWtq(t_{2i+1}, t_{2i+2})) \ge t_{\text{ph}}$,
6. *and pathDur, decreased by excluding the beginning of the path, corresponding to segment* $(\ell_0(t_0), \ell_0(t_2))$*, is before the prediction horizon, i.e.,* $pathDur - pqrStr(t_0, t_1) - pqrWtq(t_1, t_2) < t_{\text{ph}}\}$.

We write ${SEG_{\text{h}}}^{l}$ for the set of load-related historic segments.

Step 2.3. Identify Segments Where Blockage Originates From. Now, we apply the same reasoning as above, additionally considering that:

- blockages propagate *backward* in the control-flow direction,
- blockages can propagate faster than load, so we use some coefficient k to take it into account.

As a result, we consider paths going backward in the control-flow direction, and adjust the service and waiting time of the R- and Q-proclets using k. The definition of such a set reads as follows.

Definition 9.2 (Set of blockage-related historic segments)**.** *Let*

- *PQR-system* $S = (\mathcal{N}, m_v, C, \ell^C)$ *(Definition 6.7),*
- *its P-proclet* $N_0 = (P_0, T_0, F_0, \ell_0, Var_0, colSet_0, m_0, arcExp_0) \in \mathcal{N}$ *(Definition 6.2),*
- *target segment* SEG_t*, comprised by two transitions in* T_0*,*
- *coefficient* $k \in \mathbb{T}, 0 < k \le 1$*, describing blockage propagation speed in S,*
- *and prediction horizon* $t_{\text{ph}} \in \mathbb{T}$.

The set of blockage-related historic segments *is*

- $\{(\ell_0(t_2), \ell_0(t_0)) \mid t_0, t_2 \in T_0$*, and exists sequence* $\sigma = \langle t_0, t_1, t_2, t_3, \ldots, t_{2n}\rangle$ *of transitions in* T_0 *such that:*

  1. *it includes at least five transitions required to comprise one historic and one target segment, i.e.,* $n \ge 2$
  2. *it describes a valid path in the P-proclet* backward *in the control-flow direction, i.e.,* $\forall 0 < j \le 2n, \exists p \in P_0, (t_j, p), (p, t_{j-1}) \in F_0$,
  3. *no loops are possible in* σ*, i.e.,* $\forall 0 \le i < j \le 2n, t_i \neq t_j$,
  4. *the last and third transitions from the end form the target segment, i.e.,* $(\ell_0(t_{2n}), \ell_0(t_{2n-2})) = SEG_t$,

5. *the total duration* $pathDur \in \mathbb{T}$ *of the resource service time and queue waiting time on the path described by* σ*, excluding its tail corresponding to the target segment, is on or beyond the prediction horizon with respect to coefficient* k*, i.e.,* $pathDur = k \sum_{i=1}^{n-1} (pqrStr(t_{2i}, t_{2i-1}) + pqrWtq(t_{2i-1}, t_{2i-2})) \geq t_{\text{ph}}$,
6. *and pathDur, decreased by excluding the beginning of the path, corresponding to segment* $(\ell_0(t_2), \ell_0(t_0))$*, is before the prediction horizon, i.e.,* $pathDur - pqrStr(t_2, t_1) - pqrWtq(t_1, t_0) < t_{\text{ph}}\}$.

We write $SEG_{\text{h}}{}^{b}$ for the set of blockage-related historic segments. The final target segment set is $SEG_{\text{h}} = SEG_{\text{h}}{}^{l} \cup SEG_{\text{h}}{}^{b}$.

Next, we describe feature extraction.

9.3.5 Step 3. Extract Feature and Dependent Variable Values

In this step, the standard ML pipeline is used for model training. The identified parameters of the target and historic spectrum are used to instantiate these spectra from the given multi-channel performance spectrum for various values of time t_{now}, using the sliding window technique [55]. Figure 9.15 shows how the sliding window goes from the spectrum beginning (bin s) toward its end (bin e).

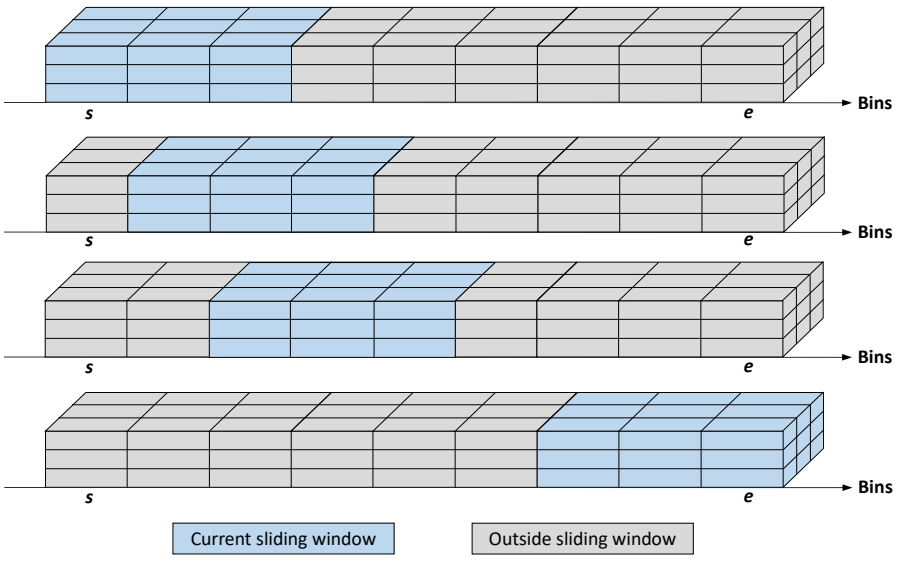


Figure 9.15: Sliding window technique over a multi-channel performance spectrum.

For each sliding window and t_{now}, the values of these spectra instances are flattened into a vector such that

- the values of the historic spectrum are feature values, and
- the values of the target spectrum are dependent variable values.

Let us consider a sliding window in Figure 9.16, where the historic spectrum parameters are:

- $CH_{\mathrm{h}} = ch_1, ch_2, ch_3$,
- $SEG_{\mathrm{h}} = seg_1, seg_3$,
- $s_{\mathrm{h}} = -1$,
- $e_{\mathrm{h}} = 0$,

and the target spectrum parameters are:

- $CH_{\mathrm{h}} = ch_1$,
- $SEG_{\mathrm{h}} = seg_2$,
- $s_{\mathrm{t}} = 1$,
- $e_{\mathrm{t}} = 1$.

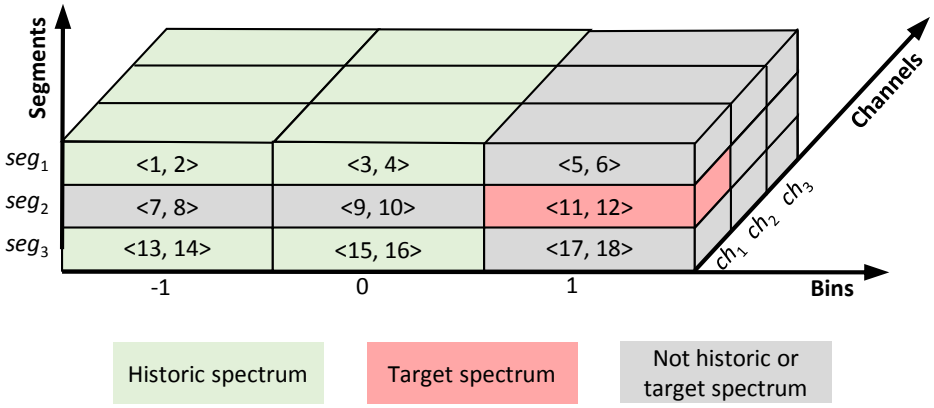


Figure 9.16: Target and historic spectrum values in a sliding window.

To generate a vector of a training or test set, we iterate over the target spectrum bins of each channel in CH_{h}. The order does not matter, but it must be the same for generating each vector of the test and training set. For ch_1 whose bin values are presented in the figure, the corresponding vector is

- $\langle 1, 2, 3, 4, 13, 14, 15, 16 \rangle$.

In the same way, we add values from the other historic channels. Finally, we add the values of the target spectrum. The resulting vector is

- $\langle 1, 2, 3, 4, 13, 14, 15, 16, \ldots, 23 \rangle$,

where … stand for the values from channels ch_2 and ch_3, and $11,12$ are the values of the target spectrum. Next, the resulting vector set V is used for model training.

9.3.6 Step 4. Train Predictive Model

In this step, vector set V is used for learning a model. Note, our method does not specify the model architecture and hyper-parameters, their selection is up to the analyst. When a model M is obtained, a decision on the model accuracy is made. If it is lower than required, more iterations of Steps 1-4 can be done to change the target/historic spectrum parameters, model architecture, or model hyper-parameters. When the required accuracy is obtained, the model is ready for deployment.

Next, we discuss the evaluation of this method.

9.4 Evaluation

In this section, we discuss the evaluated our method on problem instances PrIn_1 and PrIn_2 (see Section 9.2.4), using BHS synthetic and BHS data.

9.4.1 Experimental Setup

In the following, we give a brief overview of the scenarios and pipeline we used for our experiments first and discuss the chosen baseline and tools for the pipeline setup afterward.

Scenario Overview. Our approach allows us to extract a feature set and learn an ML model for predicting a given PPI, using a multi-dimensional performance spectrum, target PPI, and prediction horizon as input. In our pipeline for the evaluation, we implemented the following steps.

- A preliminary Step 0 to compute a multi-channel performance spectrum from a given event table (in the form of a CVS file). It was required because our method takes a spectrum (not an event table) as input.
- Step 1 to define a target spectrum.
- Step 2 to define a historic spectrum.
- Step 3 to extract a feature set.
- Step 4 to learn a predictive ML model. We chose Logistic Regression (LR) and Feed Forward (FF) Neural Network (NN) models for the evaluation with linear and non-linear models.

The model itself, as well as its metrics, were the pipeline outcome.

Last but not least, we also took into account the problem of event data incompleteness, considered in Chapter 7. Since the real BHS dataset had this problem, we

included an optional log repair sub-step in Step 0 to evaluate the effect of log repair on the accuracy of the resulting models.

Baselines. For a baseline approach, we chose the state-of-the-art approach in [35], allowing us to extract intra- and inter-case related features. As discussed in Section 9.1.2, it only predicts case-level PPIs, so we adapted it for predicting aggregate PPIs as follows.

1. We trained a model for predicting the time between *last events of trace prefixes* that end with the occurrences of historic spectrum segments, and the *starts of target segments*, using our LR and FF NN models of choice.
2. As a result, the obtained model predicted when each case would progress far enough to reach the target segment.
3. Finally, we estimated future load by counting how many cases are predicted to reach the target bin.

The latter could be done only for cases that eventually reached the target segments, i.e., it assumed beforehand knowledge about the future paths of cases (bags). This assumption held for historic data on the model training stage and for the mandatory screening step (PrIn_1) but did not hold for the prediction stage of the re-circulation problem (PrIn_2). As a result, this baseline was not applicable to problem instance PrIn_2. We integrated the source code, provided by the authors of [35], into our pipeline, to be able to use the baseline and our method interchangeably.

Additionally, as a naive baseline, we chose the average value of dependent variables, observed in a time interval $[s_h, e_h]$, corresponding to the historic spectrum. It was needed to evaluate our results for PrIn_2, where [35] could not be used.

In the following, we refer to the aforementioned methods as shown in Table 9.2.

Approach	Identifier in text
Performance spectrum- and PQR-system-based	(A1)
Data-driven inter-case feature encoding [35]	(A2)
Naive (average value of the dependent variable)	(A3)
Performance spectrum- and PQR-system-based with log repair	(A4)

Table 9.2: Mapping PPM methods to their identifiers in text.

Implementation. To implement this pipeline, we

- extended the interactive ProM plug-in "Performance Spectrum Miner" (PSM) [46] to support:
  - multi-channel performance spectra, and
  - the sliding window technique for feature extraction,

- implemented the sliding window technique for feature extraction,
- integrated the implementation of [35] into our source code, and
- implemented a Python script for training models using the PyTorch ML framework[1].

The extended PSM:

1. computes a multi-channel performance spectrum according to a provided configuration,
2. accepts the configuration of a target and historic spectra, and
3. exports a feature set into a CSV file, where each column corresponds to a bin value of the historic or target spectrum, and each row corresponds to a feature vector for a sliding window.

Additionally, used our implementation of the log repair method of Chapter 7.

The resulting dataset set could be imported by the Pythons script to train a model. To measure the errors, we computed *Root Mean Squared Error* (RMSE), *Mean Absolute Error* (MAE), and *R squared* (which is meaningful for linear models only). Additionally, we did meticulous residual diagnostics of predictions for test sets. To run the experiments, we used a laptop for work with the PSM, and a dedicated server with 40 CPUs, six GPUs, and 400 GB RAM for model training.

Next, we discuss our evaluation results.

9.4.2 Experimental Results

In this section, we first consider results obtained using the synthetic logs recorded by the BHS simulation model and then consider results obtained from the real dataset.

9.4.2.1 Evaluation Using Synthetic Data

To generate an event log for addressing PrIn_1, we re-used our simulation model of a simple BHS (see Section 6.7), comprising a typical BHS architecture and layout: conveyors, a sorting loop, a baggage screening machine, diverting and merge units. As an arrival process, a check-in scenario with normally distributed distances between bags was replayed to generate an event log with 134.000 events and 11.518 cases for 84 consecutive operating hours. The events were recorded when bags passed through various locations in the system. The resulting feature set had 15 feature variables and 15.000 samples[1], on which we applied approaches (A1), (A2), and (A3). Note, (A4) was not applied since no data incompleteness was presented in the synthetic logs. The first row of Table 9.3 (a) shows the resulting measures, where:

[1] the synthetic event log, ProM plugin, PyTorch script and source code implementing the baseline approach of [35] are available at `https://github.com/processmining-in-logistics/psm/tree/ppm`

- the LR and FF NN models trained using (A1) had two times smaller errors RMSE and MAE than the baseline method (A2),
- the LR model trained using (A1) had a greater and closer to 1.0 R squared measure than the LR model of the baseline approach (A2), i.e., it explained significantly more variable variations, and
- the naive baseline method had large RMSE and MAE.

Experiment	Approach	Model	R squared	RMSE	MAE
(a) BHS simulation model $PrIn_1$	(A1)	LR	0.82	12.1	8.2
	(A1)	FF NN	0.84*	11.6	7.7
	(A2)	LR	0.35	23.2	15.3
	(A2)	FF NN	0.46*	21.2	16.2
	(A3)	-	-	35.8	23.2
(b) Real BHS $PrIn_1$	(A1)	LR	0.74	7.0	5.0
	(A1)	FF NN	0.75*	6.9	4.8
	(A2)	LR	0.38	10.8	7.9
	(A2)	FF NN	0.69*	7.6	5.3
	(A3)	-	-	11.0	7.9
	(A4)	LR	0.75*	6.2	4.1
(c) Real BHS $PrIn_2$	(A1)	LR	0.05	3.0	1.8
	(A1)	FF NN	0.45*	2.4	1.3
	(A3)	-	-	3.0	1.7
	(A4)	FF NN	0.45*	2.4	1.3

Table 9.3: Model error measures, where RMSE and MAE are in % of max. load for (a) and (b), and re-circulation (c). **R squared values for FF NN are provided for the sake of completeness.*

9.4.2.2 Evaluation Using Real Datasets

In this experiment, we addressed both problem instances $PrIn_1$ and $PrIn_2$ for a Vanderlande-built BHS of a major European airport. In the event log, each case corresponded to one bag, events were recorded when bags passed sensors on conveyors, and the activity names described the sensor locations in the system. Note that a sensor event was recorded only if a bag was *diverted to another conveyor*, and was not recorded otherwise. As a result, the information about bag locations was significantly incomplete. For this system, the event log of one day of operations contained 850 activities, 25.000-50.000 cases (bags), and 1-2 million events on average. The entire log contained 148 million events for 120 consecutive days. Sparse events recorded during non-operating night hours were excluded from the log.

First, we addressed problem instance PrIn_1. For methods (A1) and (A4), the resulting feature set had 68 features and 108.000 vectors. As shown in Table 9.3 (b),

- the LR and FF NN models of (A1) had a smaller RMSE and MAE than the baseline models,
- the LR model R squared measure of (A1) was close to 1,
- training after log repair (A4) did not improve accuracy.

Figure 9.17 shows how the LR model correctly predicted peaks and dips in the load of the screening machines compared to the recorded data, proving that the model was adequate for the workload prediction. While the LR and FF NN models showed similar metric values, the former is preferred as a simpler solution. Log repair in (A4) improved accuracy insignificantly, we discuss the reasons for that at the end of this section.

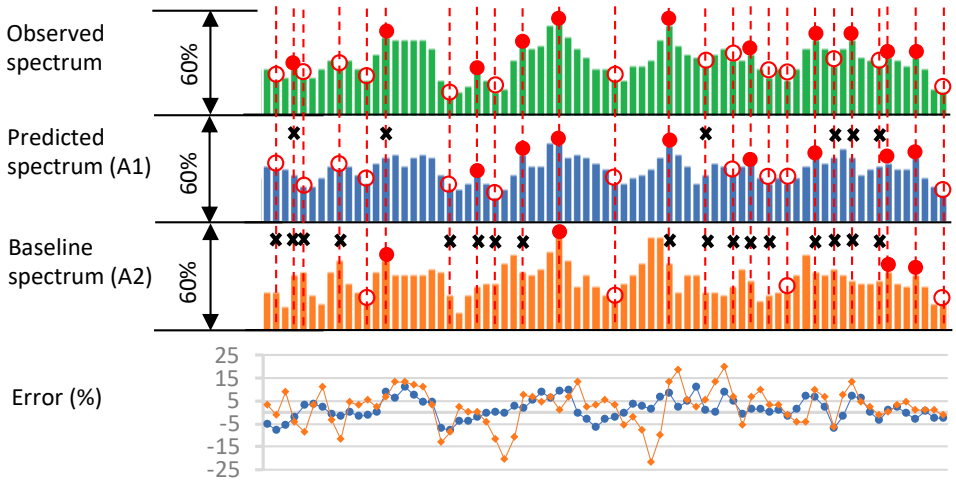


Figure 9.17: Real, predicted by (A1), and predicted by (A2) load of the baggage screening machines, in % of the maximal load (top): each bin represents the load for one minute, filled and blank circles show matched peaks and dips, while the X's show mismatches. The residuals of the predicted load in % of max. load per bin (bottom): the baseline (orange) shows greater deviations than the performance spectrum-based model (blue).

Finally, we addressed problem instance PrIn_2. We trained an LR model, and a four-layer FF NN on a dataset with 148 features and 216.000 samples using (A1) and (A4), and used approach (A3) as a baseline (since [35] was not applicable to PrIn_2). Note, the sample number is two times greater than for problem instance PrIn_1 because of the two times shorter duration of bins. Table 9.3 (c) shows:

- a close to zero R squared metric of the LR model of (A1), showing its incapability to explain variable variations, i.e., the model could not predict infrequent re-circulation peaks,
- smaller values of RMSE and MAE of the FF NN model, comparing with the baseline metrics.

Figure 9.18 sheds more light on the model's performance:

- the FF NN model correctly predicted moments of peaks in re-circulation, but consistently underestimated its actual amount,
- while the baseline demonstrated auto-correlation.

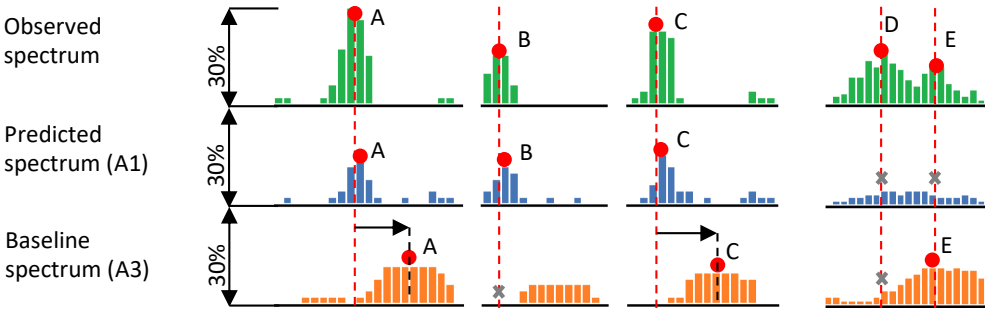


Figure 9.18: Peaks of re-circulation within 30-second bins (in % of the max. load): the FF NN model (A1) predicted peaks A, B, C in the correct bins, whiles the baseline (A3) predicted them with a significant delay as a result of auto-correlation. However, several peaks (e.g., D and E) were not predicted by the model (A1).

Last but not least, applying (A4) did not lead to any improvements. We discuss it next in more detail.

Using Repaired Event Logs For Feature Extraction. The real dataset used for evaluation was incomplete due to the system logging architecture. We applied our log repair approach (see Section 7.3) to reconstruct missing events and attributes. Due to the project time and resource limitations, we did not manage to restore the whole event log but re-used the log repair pipeline we already built for the log repair approach evaluation (see Section 7.3). That is, we restored the events between check-in locations $I_1 - I_n$, and pre-sorter locations E_1, E_2. To derive exact timestamps from the timestamp intervals, we used the minimal values, i.e., the left interval border. The resulting metrics are shown in Table 9.3, approach (A4).

For problem instance PrIn_1, the experiments showed just slightly better metrics. We explain this insignificant improvement by the fact of relatively precise event logging in the check-in area, where all the bags were always registered (and logged) in the system. Moreover, most historic spectrum segments were outside the areas, which

we repaired the log for. As a result, log repair could not influence the prediction results significantly.

For problem instance PrIn_2, the experiments showed zero improvements. This result was expected because links $A_1^1-D_1,\ldots,A_4^1-D_4, A_1^2-D_1,\ldots,A_4^2-D_4$ to final sorters $F1$ and $F2$ (see Figure9.10), whose state mainly affected the re-circulation amount, were outside the log repair scope.

To summarize, despite event log incompleteness, the sound predictive models, trained using (A1), demonstrated the feasibility of the proposed method for PPM of systems with shared resources and queues. Next, we discuss its limitations and future work.

9.5 Chapter Summary

In this chapter, we studied the problem of PPM in systems with shared resources and queues. We addressed the problem of predicting aggregate PPIs using historic event data and a process model — the PQR-system— that describes the process, queue, and resource dimensions (**RQ-7**). We showed how the channels of multi-channel performance spectra are capable of capturing different aspects of inter-case dynamics. We formulated the problem for **RQ-7** directly over a multi-channel performance spectrum and PQR-system such that an observed historic spectrum is used to estimate a future target spectrum. The target PPI is derived from the predicted target spectrum. This problem formulation allows for the formulation of a large class of PPM as a regression problem. We proposed a method for solving this problem by training an ML model on features extracted from the target spectrum to predict the target spectrum. This method is built on our method for descriptive performance analysis (see Chapter 8), and exploits our approach for tracking load and blockage propagation in systems with shared resources and queues.

We provided examples of real-life problem instances for BHSs and the evaluation of our approach by training sound models for solving these problem instances on the event log of a major European airport BHS. We demonstrated the feasibility of our approach and compared it to the current state-of-the-art approach [35]. The experiments showed that a linear model trained using our method outperformed the baseline method for the simpler problem instance, while a non-linear model, also trained using our method, outperformed the baseline method for the more complicated problem instance.

However, our method has the following limitations.

- Our approach has a too simplistic way to estimate blockage propagation time. We plan to improve it by considering the real propagation durations observed in historic data per segment.

- Although our models are technically sound, they still require validation in practice. We expect the need for higher accuracy, especially for predictions requiring a longer prediction horizon.
- Due to the time and resource limitations, our log repair approach was applied only to a part of the given incomplete event log, leaving the most events, relevant for computing historic spectra, unrepaired. As a result, the impact of log incompleteness on prediction accuracy is not measured yet. We aim to do more experiments to measure it.
- We only demonstrated feasibility for the MHS domain. For adopting our approach for business processes of other domains, we aim to design corresponding process models based on the same concepts and design choices as the PQR-system.

Last but not least, our method is orthogonal to the ML model of choice and does not provide any guidance for its selection. However, different models yield results of different accuracy, as the resulting metrics for PrIn_2 showed. Let us consider the following ML models, ordered by complexity: a linear regression model, a feedforward neural network, an LSTM model [174], and an encoder-decoder network [175]. Generally, a linear model can be trained on smaller datasets to solve simpler tasks, such as predicting accumulated load (e.g., PrIn_1). A feedforward neural network can consider non-linearity but perhaps requires larger datasets. A recurrent LSTM network can consider sequential data, such as the bins of the historic spectrum. Finally, the Recurrent Neural Network (RNN) of the encoder (for encoder-decoder networks) can produce a *thought vector* that captures the current state of an MHS such that the decoder infers the target spectrum based on the "understanding" of this current state. Additionally, *attention* can be used to take into account earlier system states. The use of RNN seems to be promising. However, it would probably require large datasets, which may not be available for many MHSs. Having said that, we did not conduct any experiments with RNNs due to time constraints, so coming up with recommendations for selecting an ML model is the subject of future work.

Chapter 10

Conclusion

This chapter concludes the thesis. For that, the key contributions to process mining of systems with shared resources and queues are summarized in Section 10.1, and the implemented software tools are discussed in Section 10.2. The main limitations of our methods and still open questions are discussed in Section 10.3. Finally, our ideas on solving the most actual open questions are considered in Section 10.4, including our vision for the further development of the tools.

10.1 Contribution

In this section, we first consider two central techniques we proposed — the performance spectrum (Chapter 3), and the PQR-system (Chapter 6). They bridge the gap in process mining techniques for performance description and modeling of systems with shared resources and queues. Then, we discuss how we built on them to create methods implementing the process mining tasks actual for answering our **AQs**, formulated in Section 4.2. Finally, we discuss how our contributions are valuable to MHS vendors like Vanderlande.

Performance Spectra. In Chapter 3, the performance spectrum, a technique for fine-grained performance description of processes, is proposed. The performance spectrum

- is a generic technique because it requires only a "classical" event log as input,
- provides an unbiased performance description because no model that would most probably introduce a bias is used for computing,
- describes the performance for *all cases together*, thereby capturing how cases interact in systems with shared resources and queues,
- can be tailored for the analyst's needs by defining a performance classifier over any information available from event attributes,

- can provide the performance description in a quantified form by computing an aggregate performance spectrum,
- can be visualized in a way that enables visual analytics using human (or artificial) intellect.

Additionally, the multi-channel performance spectrum, comprising multiple performance spectra computed using different performance classifiers, is proposed. It can capture various aspects of the process/system performance simultaneously.

We showed how the performance spectrum revealed insights about the performance of processes of different domains that no existing process mining technique could provide. We evaluated it on challenging problems of analysis of systems with shared resources and queues and showed how it allowed for obtaining correct results quickly.

These properties make the performance spectrum the number one choice for using in process mining tasks where the process performance must be considered, such as

- descriptive performance analysis, including performance pattern detection, and
- feature engineering for predictive performance monitoring, for example, to train an ML model predicting the remaining case time.

PQR-Systems. The PQR-system— a process model for systems with shared resources and queues— is suggested in Chapter 6. It enables model-based process mining techniques for systems with shared resources and queues and captures domain knowledge valuable for their performance analysis. Our contribution here is two-folded:

1. the PQR-system itself, and
2. the way we approached its design.

For the latter, we investigated existing approaches for performance analysis of MHSs in Chapter 4. As queueing theory has been studied such systems for decades, we started by exploring the queueing theory-based state-of-the-art approach. We identified the basic building blocks of MHSs and considered their (relatively simple) queue models. Crucially, we used performance spectra, showing typical behaviors observed in MHSs, to validate the assumptions used in that approach. Finally, we considered queueing networks with blocking, assembled from the basic building block models, for modeling the entire MHS.

As a result, we showed that the assumptions used in the mathematical apparatus behind the state-of-the-art approach did not hold for the behaviors we observed in the performance spectra. Moreover, their complexity showed that designing an accurate queueing theory-based model is infeasible. However, we learned the key elements capable of explaining MHS behaviors at least conceptually:

- finite-capacity FIFO queues with pre-defined time characteristics,
- resources (servers) with pre-defined time characteristics, and

- a routing function that defines the handover of jobs (TSUs, cases) between them.

In Chapter 6, we used these elements as input for designing the PQR-system. For that, we chose a synchronous proclet system, describing interactions of multiple proclets (models) as a foundation. Then, we mapped the queues, resources, and routing function to dedicated queues, resources, and process proclets, respectively, using channels to synchronize their interactions. We used a small subset of the CPN syntax to model these proclets. We also defined the PQR-system replay semantics, i.e., a way to check whether a given event log could be generated by the modeled system.

The PQR-system, together with the way we came up with its design, showed how key elements affecting process performance can be identified and modeled using a relatively simple syntax (in contrast to, for example, full-scale CPNs). In Chapter 7, we showed how this simplicity, together with the module structure (multiple proclets), allows for using (or easily adapting) existing techniques for PQR-system-based conformance checking. Consequently, it allowed us to adapt generalized conformance checking for systems with shared resources and queues, which combines the model and log repair tasks. For log repair, we proposed a method for inferring unobserved events with timestamp information, also based on the reasoning over the PQR-system.

PQR-System-Based Descriptive Performance Analysis and Predictive Performance Monitoring. In practice, using a process model is often limited to providing insights into the process, projecting descriptive statistics on the model elements, and conformance checking. In this thesis, we showed how a process model is actually an invaluable source of domain knowledge for analysis methods, decreasing, to a certain extent, the need for the analyst in the loop and potentially allowing for fully automated process analysis.

Thus, in Chapter 8, we first related performance spectra to the PQR-system. That allowed us to overcome the main drawback of performance spectra — a lack of any process-related structure, and see the performance for process steps described in the model. Additionally, we introduced performance spectra of queues and resources. That allowed us to develop a method for root-cause performance analysis by

- inferring the origins of propagation of undesirable performance pattern instances in the performance spectrum along the control flow (using the process proclet),
- identifying root causes of performance problems in these origins using the spectra of the related queues and resources.

Both the performance spectrum and PQR-system *together* are extensively used in the method steps to obtain results.

Finally, in Chapter 9, we addressed the problem of predictive performance monitoring of processes. We showed how the performance spectrum can capture the

system/process dynamics over time. It allows using it as a source of rich performance-related features. Our method identifies relevant features in the performance spectrum to train a predictive ML model. For that, similarly to the method in Chapter 8, the PQR-system is used for identifying relevant "pieces" of the performance spectrum for feature extraction.

Contribution to Vanderlande. Now, we summarize our contribution to Vanderlande and other organizations where process mining of MHSs is an actual problem. We structure it into the following parts.

- **Descriptive performance analysis.** We showed that in MHSs, cases interact with each other on shared resources and queues, and this interaction usually causes situations of interest for analysis. As a result, "classical" process mining techniques and tools, considering only cases in isolation and only the control-flow perspective, are not helpful. Instead,
  - event data capturing the behavior of the process, queues, and resources,
  - a process model describing them (such as PQR-system), and
  - a performance description capturing their interactions in all three dimensions (such as performance spectrum)

  are to be used as input for methods solving various process mining tasks. It is also beneficial to use domain knowledge from the model, so model-based methods are prime candidates for these tasks.
- **Modeling MHSs.** Although the use of the PQR-system is beneficial for many process mining tasks, its manual design for each MHS is time-consuming, and no algorithms have been proposed for PQR-system discovery from event data yet. However, in industry, each MHS usually has a complete set of documentation. As a result, there is no need to discover the PQR-system. Instead, it can be automatically generated if the required information is described according to some standard. The PQR-system formal definition, proposed in Chapter 6, is the input for designing such a standard. It can be used either at Vanderlande or industry-wide for creating the standard and generating PQR-systems without extra costs.
- **Event data logging and pre-processing.** We showed how event data incompleteness impedes the analysis and how complicated and computationally complex log repair algorithms are. In case of incomplete data, analysis costs increase while analysis accuracy drops. Additionally, the lack of standardization for the event logging architecture causes extra effort for event log extraction. MHS analysis would benefit from *collecting complete* multi-case notion event data represented in some common standard for event data exchange, such as OCEL [64]. For MHS vendors, it is possible to design and implement an event data logging architecture that achieves this goal. We recommended introducing

a standard for collecting and representing MHS event data for designing new and updating existing systems.

- **Predictive performance monitoring.** Although we obtained models for predicting aggregate process performance indicators, we consider them feasible, admitting thereby that the problem is still open. It is overwhelmingly difficult due to complicated system behavior and a large number of "participating" entities (TSUs, queues, resources, etc.) Moreover, no results have been published so far on the problem of predicting individual process performance indicators *per TSU*. We believe, one of the reasons is the limited amount of information available as input for this problem. For example, additional information on the system routing decisions over time, flight schedule (for BHSs), system error description, and so on, can be additionally collected and provided for process mining solutions.
- **Digital twins of MHSs.** The problem of creating an MHS digital twin was not in the project scope but nevertheless was regularly discussed. Although nowadays it is a broad open research problem, we believe that creating a digital twin of an MHS can be a reality in a decade, and process mining can significantly contribute to it. However, what is our contribution? Let us summarize it as follows.
  - We showed how incomplete data and the lack of standardization blocked approaching analysis problems or affected the outcome. Undoubtedly, the availability of complete and standardized data is also a crucial "prerequisite" for designing a digital twin. However, such data cannot be "obtained" instantly but requires systematic efforts and significant time for collection. Our project helped to create awareness of this problem.
  - We consider this thesis as the first step toward object-centric process mining of MHSs. That is, the PQR-system describes three process dimensions and their synchronization, while the performance spectrum describes the interactions of all cases together. We believe that full-scale object-centric process mining, empowered by techniques of other fields like AI, is essential for creating digital twins. Thereby, our research contributes to the problem and opens doors for further research in this direction.
  - Last but not least, we showed how a process model in the MHS domain is not mainly an analog of an MFD for the analyst, but a central input artifact for various algorithms solving different process mining tasks. We think it will help intensify further research targeting the creation of more advanced models of material handling processes, to be eventually used as a rich source of knowledge about the system for its digital twin.

10.2 Implemented Tools

We developed several software tools to analyze MHSs using event data and evaluate new methods. In this section, we discuss how these tools evolved, why they got particular features, and how they provided an efficient environment for conducting research in this thesis.

The Performance Spectrum Miner (PSM). We started our research with an exploratory analysis of available MHS datasets with existing process mining tools. One of the challenges was detecting and explaining the outlier behavior of TSUs in particular system areas during specific periods. For that, the most promising approach was to explore TSUs' behavior using event log replay and animation over a process model or map. However, it did not work because of the overwhelming amount of tokens in visualization, the inability to see event attributes (e.g., tasks and error codes), misleading approximation of tokens' progress in zoomed-out models, and so on.

As a result, we decided to implement a tool for the visual comparison of various performance metrics over time for multiple time intervals. It was the first version of the PSM, which could show aggregate performance spectra for *two* periods *simultaneously*. However, the amount of information was still large, so we started showing just one period. Then, we expanded the aggregate information, thereby obtaining non-aggregate performance spectra capable of revealing fascinating patterns in event data of processes. We introduced performance classifiers for segment occurrences, and color coding for visualizing them. It made the visualized information similar to the spectra of electromagnetic or audio waves. This observation helped to come up with the name. Participating in the Business process intelligence challenge 2018 [151] helped adjust the PSM features initially designed for MHS processes.

Figure 10.1 shows an MHS system, the data it records, the tools we developed, the PQR-system, and other artifacts. We started research using (incomplete) event logs extracted by Vanderlande, that the PSM could import to generate performance spectra.

Event Log Extraction Approaches. MHSs record large amounts of event data. However, their storage representation is usually very different from XES or OCEL event logs. For example, it can be a database with many hundreds of tables. Moreover, MHS datasets are usually larger than ones recorded by classical business processes for the same period. As a result, the algorithms of the event log and event table extraction can be quite complicated, and the datasets themselves do not fit into the memory of a "good" laptop (e.g., 32GB). As a result, the former can block applying any process mining technique on "raw" MHS datasets, and the latter can cause the memory explosion problem for open-source and commercial tools when data for a relatively large period of time must be analyzed in one piece.

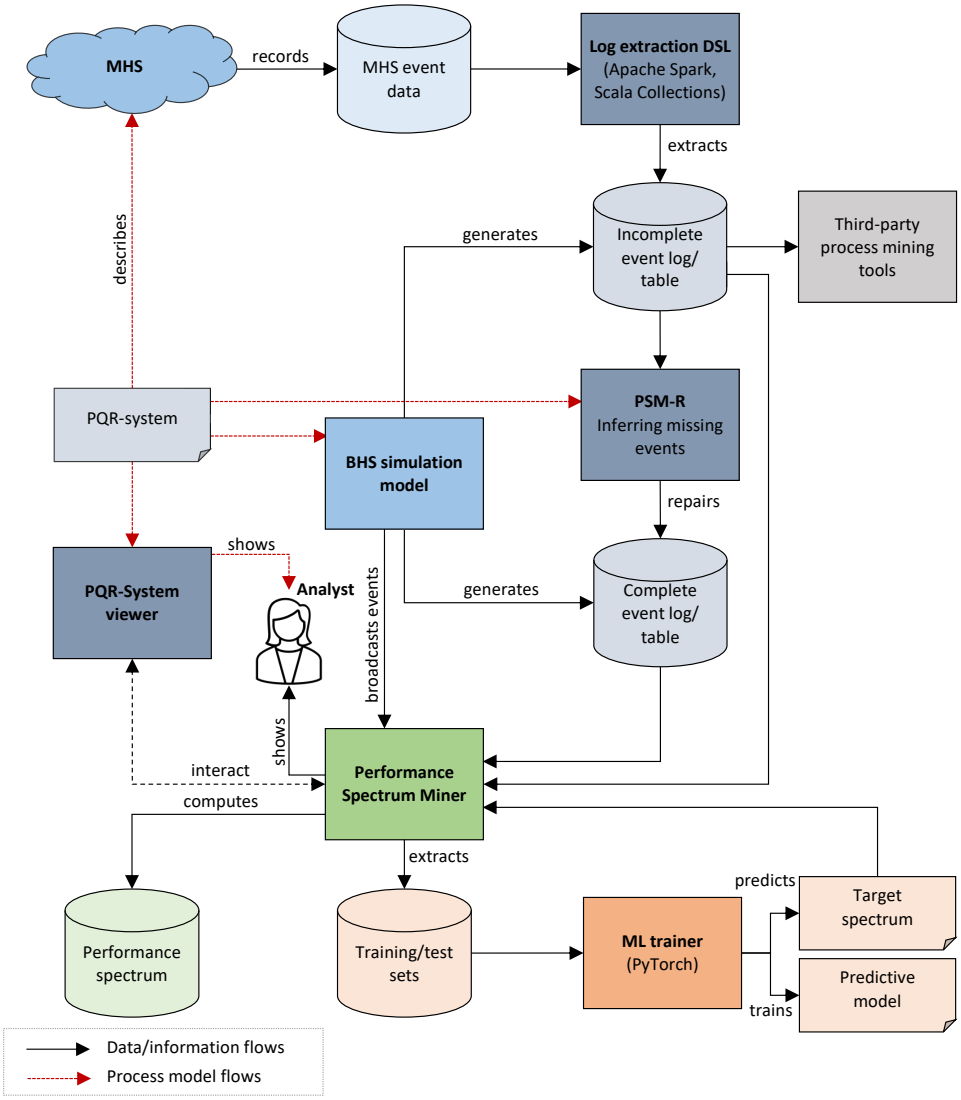


Figure 10.1: Overview of the developed software tools.

To address the former, i.e., event log extraction and transformation, we implemented a Domain Specific Language (DSL) in Scala on top of Apache Spark, a high-level framework for distributed data processing implemented in Scala. The corresponding box is shown in Figure 10.1, where it consumes "raw" MHS event data, and extracts event tables or logs. Besides Apache Spark, the DSL was backed by the standard Scala parallel collections, to be used in a single-server configuration when the

use of a Spark cluster is impossible or infeasible. Although we could not open-source it or include it in this thesis, we still conclude here that this approach is sound if analysts with moderate programming skills for using the DSL are available, and the infrastructure is in place.

To address the latter, i.e., the memory explosion problem, we made the PSM capable of working with large datasets that did not fit in the memory of a typical laptop. For that, we implemented partitioning of input logs for computing performance spectra and saving results on disk, to load required pieces of spectra *on demand*.

The Conveyor "Digital Twin", BHS Simulation Models, and Log Repair. The MHS datasets that we extracted and analyzed were incomplete. Because it impeded all kinds of analysis, we decided to address the problem of log repair. However, the ground truth, i.e., complete event data for evaluating possible log repair approaches, was missing. To obtain it, we decided to use a simulation model.

Initially, we started with an attempt of creating a "digital twin" of a single MHS conveyor in the CPN Tools. The goal was twofold: to study its behavior under different load conditions, and use it later as a building block for simulating larger MHS areas. However, after designing 80% of the model, we started experiencing long delays after each edit in the CPN Tools' editor. They were caused by automatic validation of the entire model after each change. As a result, we could not even complete the model design. Eventually, we had to give up because the CPN Tools did not work for that, and no other tools for modeling CPNs were available.

Later, we also attempted to simulate a small BHS in the AutoMod[1] — software for simulation of production and logistics systems. The additional goal was to understand whether it was a feasible approach for *what-if* analysis of BHSs. For that, we had a six-week "internship" in the Simulation and Emulation group of Vanderlande, trying to design a BHS simulation model that would fit our needs. The screenshot of the animation of the resulting model is shown in Figure 10.2. However, the overhead due to the oversimplified yet low-level programming language of AutoMod, requiring time-consuming coding, and other limitations we encountered made this approach infeasible, at least under the project time constraints.

Instead, we implemented a simple simulation model (see Section 6.7) capable of modeling the key aspects of the system behavior we were interested in. It is capable of recording both complete and incomplete event logs and broadcasting events about TSUs' movement in real time over the network. As Figure 10.1 shows, it takes a PQR-system description as input and generates both incomplete and complete event logs according to pre-configured scenarios.

Afterward, we implemented our log repair approach. For its evaluation, we incorporated our method into a dedicated version of the PSM, called PSM-R, capable of computing and showing performance spectra with regions. In Figure 10.1, the PSM-R

[1] `https://www.simul8.com/products/automod/`

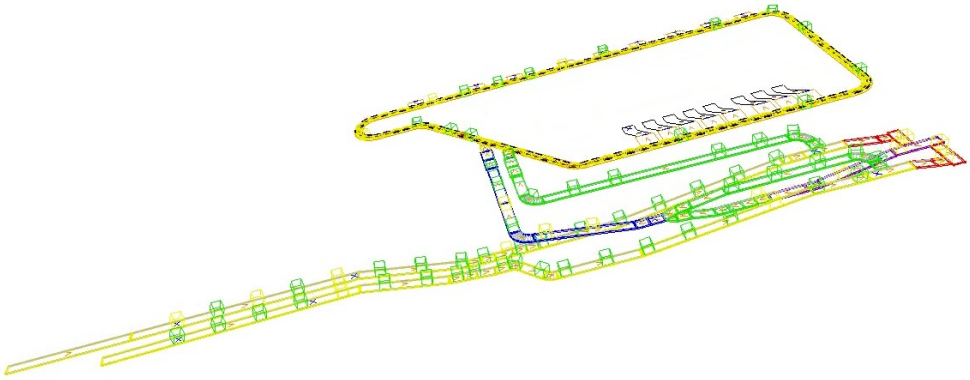


Figure 10.2: Screenshot of animation of a BHS simulation model in AutoMod: two links from the check-in area go via hold baggage scanners at the bottom right corner and continue to the sorting loop with laterals. The cubes represent bags on the conveyors.

takes an incomplete event table and produces a complete one, additionally taking the PQR-system as input. We kept the PSM-R as a branch of the "regular" PSM because converting it into a general-purpose software tool would require too much time for coding.

Integrated PQR-System and Performance Spectrum. The integrated process model and performance spectrum empower descriptive performance analysis and predictive performance monitoring. Vanderlande used this idea for implementing an internal process mining tool. However, implementing a full-scale general-purpose *open-source* software would require a significant amount of project time. Instead, we implemented a proof-of-concept tool, based on the PSM, showing the key benefits of such integration. As Figure 10.1 shows, a PQR-system description is used to visualize it in the PQR-system viewer, a UI application. This PSM modification either loads logs from disk as usual or receives events over the network in real-time from the simulation model for evaluating scenarios of process performance monitoring. The PQR-system viewer and the PSM interact for navigating throughout the PQR-system and the spectra of the process dimensions, filtering/sorting the segments, and so on. As with PSM-R, this implementation is kept as a separate branch of the regular PSM. Additionally, a PSM modification integrated with "classical" Petri nets was implemented as a ProM plugin [176] during a separate project.

Feature Extraction and Model Training. For our PPM method evaluation, we added to the PSM the ability to extract and export training/test sets from performance spectra. Thus, as Figure 10.1 shows, a performance spectrum can be used for feature extraction using parameters defined in the PSM configuration. For that, the ability

of the PSM to work with large datasets is crucial for handling large datasets. The PyTorch-based python script uses the obtained training and test sets for training various ML models. Note that the predicted target performance spectrum can be loaded back to the PSM to compare it with the observed spectrum.

10.3 Limitations and Open Issues

This section discusses the main limitations and open questions of the methods in this thesis.

Modeling Non-Material Handling Systems with Shared Resources and Queues The proposed PQR-system represents only a sub-class of systems with shared resources and queues. It means MHSs with batching, i.e., systems where TSUs can be consolidated using containers, trays, and pallets, cannot be modeled. "Classical" business processes that have parallelism, interchangeable resources, and different queue disciplines with priorities are not supported as well. Their support requires revising the PQR-system to capture these aspects. We consider it as an open and very actual research question.

Implementing Conformance Checking. We systematically explored how existing techniques can be used or extended for PQR-system-based conformance checking. However, we did not provide any implementation and evaluation. Moreover, these techniques provide little diagnostics in case of outliers. So, the problem of designing a method for relating event data to PQR-systems with extended diagnostics is open.

Instantiating Exact Timestamp for Inferred Events. The proposed log repair approach (see Chapter 7) reconstructs events, missing in the given event data, and possible intervals $[a, b]$ of their timestamps. Then, either interval border (a or b) can be used to instantiate a correct and complete event table with exact timestamps. However, this way of timestamp instantiation does not always result in the closest (to real timestamp) values. Choosing values from the entire intervals $[a, b]$ can lead to a more accurate reconstruction of the timestamps. How to choose these values is an open research question, which has been partially addressed in [125, 177].

Automating Descriptive Performance Analysis. We proposed the method for multi-dimensional root cause performance analysis that extensively uses information about detected blockage and high load instances in performance spectra. However, it assumes their manual detection by the analyst. The manual way of detection is slow and error-prone and requires painstaking exploration of large "areas" of performance spectra. Undoubtedly, the use of algorithms capable of detecting these pattern instances automatically can dramatically speed up the analysis. Moreover, currently, propagation discovery is also made manually by the analyst. Complete automation of this method is an open problem.

Increasing Prediction Accuracy and Horizon. We proposed a method for predictive performance monitoring for aggregate process performance indicators and showed its feasibility using two problem instances. We showed accurate predictions for a practically meaningful prediction horizon for one of them, while for another, both accuracy and prediction horizon length were fair. Thus, we consider the problem of predictive performance monitoring for systems with shared resources and queues an open problem because better accuracy and longer prediction horizon are required in practice. Moreover, we did not consider the problem of predicting process performance indicators for individual cases, which remains actual and open.

10.4 Future Work

In this section, we discuss our ideas for overcoming the identified limitations and solving still open questions of Section 10.3. Additionally, we share our vision for further development of the software tools discussed in Section 10.2.

Modeling MHSs with Batching. The PQR-system does not model batching, as we discussed previously. However, there is a way to support it. The idea is to have a P-proclet per each transportation unit type, for example, per bag/cartoon, container, palette. For each process type, the process is similar to TSU handling in the PQR-system, e.g., containers still must follow each other on conveyors (FIFO ordering) and be merged/diverted at the corresponding units. The main challenge is to model the consolidating and breaking down processes, which can be modeled, for example, through introducing one-to-many or many-to-many relations between the system entities. The replay semantics of this extended PQR-system would have additional constraints related to the consolidating and breaking down processes (proclets).

Modeling Manufacturing Processes. This thesis primarily focuses on material handling processes. However, its contributions may also be valuable for process mining of *manufacturing processes* because they have much in common. That is, in both domains

- the process/system entities are machines, workers, the software controlling the machines, the environment, and objects (materials) to be handled, e.g., TSUs and parts to be assembled,
- the entities interact and affect each other,
- batching can take place,
- machines have queues of entities to be handled,
- most process steps are usually automated, and
- humans play roles of either observers or actors [4].

As a result, we expect that the key concepts for modeling manufacturing processes are the same as for material handling. In this case, a dedicated synchronous proclet system, similar to the PQR-system, can be designed. We expect that the support

of batching may be the most important extension for such a model, and perhaps more advanced models for human actors, whose role in MHSs is relatively simple and therefore modeled as a very simple lifecycle.

Further, generalized conformance checking may be helpful to either detect and diagnose outliers or infer missing events, for example, not recorded because of *digital gaps* [4] of human task execution. Then, the performance spectrum can be used for root cause performance analysis and predictive performance monitoring.

We consider extending the methods of this thesis to manufacturing processes as promising future work.

Modeling "Classical" Non-Material Handling Business Processes. Support for "classical" processes can be achieved by

- introducing a full-scale Petri-net with parallelism for describing the control flow in the P-proclets,
- modeling case priorities and different ordering disciplines in queues in the Q-proclets, and
- describing resource management in the R-proclets.

Additionally, pools of resources (e.g., assignment groups) dedicated to solving particular tasks can be modeled as a new type of proclets — *group* proclets. Then, resources should not be "wired" to particular process steps, as in the PQR-system. Last but not least, many-to-many relations between entities will be required to represent complex processes.

Timestamp Instantiating for Inferred Unobserved Events. The still unsolved problem of instantiating exact timestamps from time intervals is discussed in Section 10.3. While this problem is complex, there is an observation that can help solve it. Analyzing the performance spectra of the ground truth event data, we noticed that the minimum timestamp values were usually correct except for the events recorded during blockage instances, when the actual timestamp values tended to have the maximum values, besides just one or two events with values from the middle of the interval. The idea is to learn an ML model estimating the exact timestamps for events comprising blockage instances on simulation models or systems with better logging, and apply transfer learning to obtain models for concrete systems/processes.

Improving Predictive Performance Monitoring Accuracy. As we already said, richer information can help improve prediction accuracy. Additionally, recurrent neural networks, such as LSTM [174], can potentially help capture the time dimension and achieve better results. However, we believe that the problem and the corresponding analysis questions should be revised and formulated differently to approach reliable predictive performance monitoring of MHSs.

In MHSs, processes are mainly automated, so software controllers "decide" how to route TSUs, switch equipment modes, and so on. However, they do not predict

undesirable performance scenarios. Predictive monitoring models can predict them and affect the controllers' decision-making. However, there is no guarantee that the resulting (future) performance will be improved and that no undesirable side effects will be introduced.

Alternatively, the digital twin of the system with what-if capabilities, based on (among other things) both predictive performance monitoring and exact knowledge about the algorithms of the controllers, may solve this problem. Process mining, in general, and the methods and techniques of this thesis, in particular, can help design and validate digital twins.

Further Development of the Software Tools. To conclude this section, we share our vision for further development of the software tools implemented for this thesis. They provide a "framework" designed around the performance spectrum and PQR-system. While the former is a general-purpose process mining technique implemented as a process mining tool with a rich set of features, the latter is designed primarily for MHSs and implemented as a proof-of-concept. Moreover, the log extraction DSL was not open-sourced. As a result, re-using the entire framework to analyze non-MHS processes is only possible with further software extension.

Having said that, we do not mean extensive software development. Instead, we see this extension as integration with existing techniques aimed at supporting object-centric process mining [73]. The architecture of a revised framework is shown in Figure 10.3.

In this figure, a business process, described by a process model, records event data in some custom format. The log extraction and repair module exports the data as an object-centric event log [64, 73], repairing the data if needed. In the case of MHSs, the implemented DSL and proposed log repair method can be used. Assuming that a process model (e.g., *directly-follows multigraph* [73]) is provided in a standardized format, the Process model and event log import module can read the model and log to import them into a graph database, e.g., Neo4j[2], to be represented as a model graph and event graph respectively [143, 145, 178].

A graph database provides access to the model graph and allows various queries against the event graph, including extracting traces for a particular entity (case notion) [145]. The PSM can be extended to query a graph database directly to obtain the model graph and an event log for a required process dimension. As a result, the process model and the performance spectra for cases of any case notion, presented in the event graph and described by the process model, can be computed, visualized, and used for feature extraction if needed. Such an architecture will be a powerful and flexible tool for facilitating further research in object-centric process mining and related fields.

[2] `https://neo4j.com/`

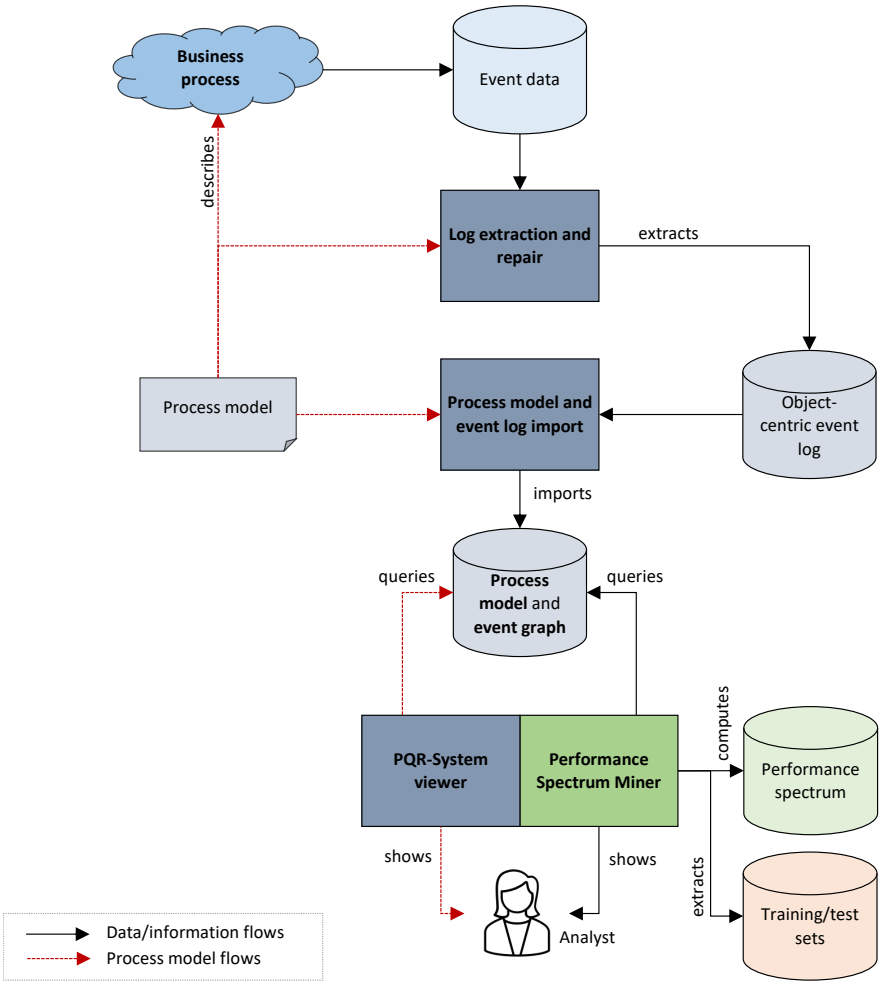


Figure 10.3: Further development of the PSM and PQR-system viewer based on representing the graph of a process model, and an object-centric event log in the form of an event graph in a graph database.

Bibliography

[1] Wil M. P. van der Aalst. *Process Mining - Data Science in Action, Second Edition*. Springer, 2016. (Cited on pages 1, 16, 21, 43, 47, 53, 67, 152, 154, 155, 156, 197, 205, 211, 216, and 221.)

[2] Geoff Black and Valeriy Vyatkin. Intelligent component-based automation of baggage handling systems with IEC 61499. *IEEE Transactions on Automation Science and Engineering*, 7(2):337–351, 2010. (Cited on page 1.)

[3] Can Saygin and Balaji Natarajan. RFID-based baggage-handling system design. *Sensor Review*, 2010. (Cited on page 1.)

[4] Stefanie Rinderle-Ma and Juergen Mangler. Process automation and process mining in manufacturing. In *International Conference on Business Process Management*, pages 3–14. Springer, 2021. (Cited on pages 1, 381, and 382.)

[5] Onur Dogan, Antonio Martinez-Millana, Eric Rojas, Marcos Sepúlveda, Jorge Munoz-Gama, Vicente Traver, and Carlos Fernandez-Llatas. Individual behavior modeling with sensors using process mining. *Electronics*, 8(7):766, 2019. (Cited on page 1.)

[6] Prasannjeet Singh, Mehdi Saman Azari, Francesco Vitale, Francesco Flammini, Nicola Mazzocca, Mauro Caporuscio, and Johan Thornadtsson. Using log analytics and process mining to enable self-healing in the internet of things. *Environment Systems and Decisions*, pages 1–17, 2022. (Cited on page 1.)

[7] Marlon Dumas, Fabiana Fournier, Lior Limonad, Andrea Marrella, Marco Montali, Jana-Rebecca Rehse, Rafael Accorsi, Diego Calvanese, Giuseppe De Giacomo, Dirk Fahland, et al. AI-augmented business process management systems: A research manifesto. *ACM Transactions on Management Information Systems*, 2022. (Cited on page 2.)

[8] Wil M. P. van der Aalst, Oliver Hinz, and Christof Weinhardt. Resilient digital twins: organizations need to prepare for the unexpected, 2021. (Cited on page 2.)

[9] Wil M.P. van der Aalst. Concurrency and objects matter! Disentangling the fabric of real operational processes to create digital twins. In *Theoretical Aspects of Computing–ICTAC 2021: 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8–10, 2021, Proceedings 18*, pages 3–17. Springer, 2021. (Cited on page 2.)

[10] Iris Beerepoot, Claudio Di Ciccio, Hajo A Reijers, Stefanie Rinderle-Ma, Wasana Bandara, Andrea Burattin, Diego Calvanese, Tianwa Chen, Izack Cohen, Benoît Depaire, et al. The biggest business process management problems to solve before we die. *Computers in Industry*, 146:103837, 2023. (Cited on page 2.)

[11] Florian Stertz, Juergen Mangler, and Stefanie Rinderle-Ma. The role of time and data: Online conformance checking in the manufacturing domain. *arXiv preprint arXiv:2105.01454*, 2021. (Cited on page 2.)

[12] Vanderlande on the Wikipedia. `https://en.wikipedia.org/wiki/Vanderlande`. Accessed: 2023-01-21. (Cited on page 3.)

[13] Vanderlande's profile. `https://www.vanderlande.com/about-vanderlande/company-profile/`. Accessed: 2023-01-21. (Cited on page 3.)

[14] Vanderlande's facts and figures. `https://www.vanderlande.com/about-vanderlande/facts-and-figures/`. Accessed: 2023-01-21. (Cited on page 7.)

[15] Work at Albert Heijn Zaandam. `https://careers.vanderlande.com/albertheijn-zaandam/`. Accessed: 2023-01-21. (Cited on page 7.)

[16] Hong Chen and David D. Yao. *Fundamentals of queueing networks: Performance, asymptotics, and optimization*, volume 46. Springer Science & Business Media, 2013. (Cited on pages 15 and 145.)

[17] S. Balsamo, V. de Nitto Persone, and R. Onvural. *Analysis of Queueing Networks with Blocking*. International Series in Operations Research & Management Science. Springer US, 2001. (Cited on pages 15, 19, 131, 133, 134, 150, and 153.)

[18] James MacGregor Smith. Robustness of state-dependent queues and material handling systems. *International Journal of Production Research*, 48(16):4631–4663, 2010. (Cited on pages 15, 19, 29, 41, 121, 122, 129, 130, 131, 133, 138, 140, 141, 143, 146, and 151.)

[19] Prince Bedell and James MacGregor Smith. Topological arrangements of M/G/c/K, M/G/c/c queues in transportation and material handling systems. *Computers & operations research*, 39(11):2800–2819, 2012. (Cited on pages 15, 19, 122, 141, 143, and 151.)

[20] James MacGregor Smith and Laoucine Kerbache. State-dependent models of material handling systems in closed queueing networks. *International Journal of Production Research*, 50(2):461–484, 2012. (Cited on pages 15, 19, 29, 41, 122, 141, 143, and 151.)

[21] Rajat Jain and James MacGregor Smith. Modeling vehicular traffic flow using M/G/c/c state dependent queueing models. *Transportation Science*, 31(4):324–336, 1997. (Cited on pages 15, 19, 122, 129, 130, and 151.)

[22] Caterina Malandri, Marco Briccoli, Luca Mantecchini, and Filippo Paganelli. A discrete event simulation model for inbound baggage handling. *Transportation Research Procedia*, 35:295 – 304, 2018. INAIR 2018. (Cited on pages 16 and 158.)

[23] Saeid Nahavandi, Bruce Gunn, Michael Johnstone, and Douglas Creighton. Modelling and simulation of large and complex systems for airport baggage handling security. In Kohei Arai, Supriya Kapoor, and Rahul Bhatia, editors, *Intelligent Computing*, pages 1055–1067, Cham, 2019. Springer International Publishing. (Cited on pages 16 and 158.)

[24] Josep Carmona, Boudewijn F. van Dongen, Andreas Solti, and Matthias Weidlich. *Conformance Checking - Relating Processes and Models*. Springer, 2018. (Cited on pages 16, 17, 32, 154, 155, 177, 206, 216, and 228.)

[25] Wil M.P. van der Aalst, Martin Bichler, and Armin Heinzl. Robotic process automation. *Business and Information Systems Engineering*, 60, 05 2018. (Cited on page 16.)

[26] Maikel L. van Eck, Xixi Lu, Sander J.J. Leemans, and Wil M.P. van der Aalst. PM^2: a process mining project methodology. In *International conference on advanced information systems engineering*, pages 297–313. Springer, 2015. (Cited on page 17.)

[27] Jen Yeng Cheah and James MacGregor Smith. Generalized M/G/c/c state dependent queueing models and pedestrian traffic flows. *Queueing Systems*, 15(1-4):365–386, 1994. (Cited on pages 19, 122, 129, 130, and 151.)

[28] K. Mani Chandy, Ulrich Herzog, and Lin Woo. Parametric analysis of queuing networks. *IBM Journal of Research and Development*, 19(1):36–42, 1975. (Cited on page 19.)

[29] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984. (Cited on page 19.)

[30] Søren Asmussen, Soren Asmussen, and Sren Asmussen. *Applied probability and queues*, volume 2. Springer, 2003. (Cited on page 19.)

[31] Douglas R. Miller. Computation of steady-state probabilities for M/M/1 priority queues. *Operations Research*, 29(5):945–958, 1981. (Cited on page 19.)

[32] Alfonso E. Márquez-Chamorro, Manuel Resinas, and Antonio Ruiz-Cortés. Predictive monitoring of business processes: A survey. *IEEE Transactions on Services Computing*, 11(6):962–977, Nov 2018. (Cited on pages 20, 155, 228, 319, and 328.)

[33] Claudio Di Ciccio, Giovanni Meroni, and Pierluigi Plebani. On the adoption of blockchain for business process monitoring. *Software and Systems Modeling*, 21(3):915–937, 2022. (Cited on page 20.)

[34] Arik Senderovich, Matthias Weidlich, Avigdor Gal, and Avishai Mandelbaum. Queue mining for delay prediction in multi-class service processes. *Information Systems*, 53:278–295, 2015. (Cited on pages 20, 144, and 151.)

[35] Aric Senderovich, Chiara Di Francescomarino, and Fabrizio Maggi. From knowledge-driven to data-driven inter-case feature encoding in predictive process monitoring. *Information Systems*, 2019. (Cited on pages 20, 44, 158, 331, 334, 335, 363, 364, 366, and 368.)

[36] Mirko Polato, Alessandro Sperduti, Andrea Burattin, and Massimiliano de Leoni. Time and activity sequence prediction of business process instances. *Computing*, pages 1–27, 2018. (Cited on pages 20, 68, 156, and 158.)

[37] Mahsa Pourbafrani and Wil M. P. van der Aalst. Extracting process features from event logs to learn coarse-grained simulation models. In *Advanced Information Systems Engineering: 33rd International Conference, CAiSE 2021, Melbourne, VIC, Australia, June 28–July 2, 2021, Proceedings*, pages 125–140. Springer, 2021. (Cited on page 20.)

[38] Björn Rafn Gunnarsson, Jochen De Weerdt, and Seppe vanden Broucke. A framework for encoding the multi-location load state of a business process. 2022. (Cited on page 20.)

[39] Niek Tax, Ilya Verenich, Marcello La Rosa, and Marlon Dumas. Predictive business process monitoring with LSTM neural networks. In *CAiSE 2017*, volume 10253 of *LNCS*, pages 477–492. Springer, 2017. (Cited on pages 21, 158, and 319.)

[40] Eva L. Klijn and Dirk Fahland. Identifying and reducing errors in remaining time prediction due to inter-case dynamics. In *2020 2nd International Conference on Process Mining (ICPM)*, pages 25–32. IEEE, 2020. (Cited on page 21.)

[41] Eva L. Klijn and Dirk Fahland. Performance mining for batch processing using the performance spectrum. In *International Conference on Business Process Management*, pages 172–185. Springer, 2019. (Cited on pages 21 and 95.)

[42] Wolfgang Reisig. *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013. (Cited on pages 21, 47, and 50.)

[43] Wil M.P. van der Aalst, Arya Adriansyah, and Boudewijn F. van Dongen. Replaying history on process models for conformance checking and performance analysis. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(2):182–192, 2012. (Cited on pages 21, 32, 154, 156, 177, 206, 209, 211, 216, 217, 223, 228, and 246.)

[44] Andreas Rogge-Solti, Arik Senderovich, Matthias Weidlich, Jan Mendling, and Avigdor Gal. In log and model we trust? A generalized conformance checking framework. In *International Conference on Business Process Management*, pages 179–196. Springer, 2016. (Cited on pages 25, 32, 41, 154, 156, 205, 208, 209, and 254.)

[45] Vadim Denisov, Dirk Fahland, and Wil M. P. van der Aalst. Unbiased, fine-grained description of processes performance from event data. In Mathias Weske, Marco Montali, Ingo Weber, and Jan vom Brocke, editors, *Business Process Management - 16th International Conference, BPM 2018, Sydney, NSW, Australia, September 9-14, 2018, Proceedings*, volume 11080 of *Lecture Notes in Computer Science*, pages 139–157. Springer, 2018. (Cited on pages 28, 39, 40, 93, 94, 95, 157, 228, 230, 259, and 262.)

[46] Vadim Denisov, Elena Belkina, Dirk Fahland, and Wil M. P. van der Aalst. The performance spectrum miner: Visual analytics for fine-grained performance analysis of processes. In Wil M. P. van der Aalst, Fabio Casati, Raffaele Conforti, Massimiliano de Leoni, Marlon Dumas, Akhil Kumar, Jan Mendling, Surya Nepal, Brian T. Pentland, and Barbara Weber, editors, *Proceedings of the Dissertation Award, Demonstration, and Industrial Track at BPM 2018 co-located with 16th International Conference on Business Process Management (BPM 2018), Sydney, Australia, September 9-14, 2018*, volume 2196 of *CEUR Workshop Proceedings*, pages 96–100. CEUR-WS.org, 2018. (Cited on pages 28, 39, 69, 85, 93, 95, 157, 313, 320, and 363.)

[47] Vadim Denisov, Dirk Fahland, and Wil M. P. van der Aalst. Predictive performance monitoring of material handling systems using the performance spec-

trum. In *International Conference on Process Mining, ICPM 2019, Aachen, Germany, June 24-26, 2019*, pages 137–144. IEEE, 2019. (Cited on pages 28, 36, 38, and 230.)

[48] Dirk Fahland. Describing behavior of processes with many-to-many interactions. In Susanna Donatelli and Stefan Haar, editors, *Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings*, volume 11522 of *Lecture Notes in Computer Science*, pages 3–24. Springer, 2019. (Cited on pages 30, 41, 62, 152, 153, 156, 159, 162, 166, 174, 196, and 202.)

[49] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009. (Cited on pages 30, 55, 60, and 153.)

[50] Vadim Denisov, Dirk Fahland, and Wil M. P. van der Aalst. Repairing event logs with missing events to support performance analysis of systems with shared resources. In Ryszard Janicki, Natalia Sidorova, and Thomas Chatain, editors, *Application and Theory of Petri Nets and Concurrency*, pages 239–259, Cham, 2020. Springer International Publishing. (Cited on pages 31, 33, 220, and 221.)

[51] Dirk Fahland, Vadim Denisov, and Wil M.P. van der Aalst. Inferring unobserved events in systems with shared resources and queues. *Fundamenta Informaticae*, 183(3-4):203–242, 2021. (Cited on pages 31 and 33.)

[52] Massimiliano De Leoni and Wil M.P. van der Aalst. Aligning event logs and process models for multi-perspective conformance checking: An approach based on integer linear programming. In *Business Process Management*, pages 113–129. Springer, 2013. (Cited on pages 32, 154, 155, and 221.)

[53] Elham Ramezani Taghiabadi, Dirk Fahland, Boudewijn F. van Dongen, and Wil M.P. van der Aalst. Diagnostic information for compliance checking of temporal compliance requirements. In *International Conference on Advanced Information Systems Engineering*, pages 304–320. Springer, 2013. (Cited on pages 32, 154, 221, 222, 223, and 224.)

[54] Vadim Denisov, Dirk Fahland, and Wil M. P. van der Aalst. Multi-dimensional performance analysis and monitoring using integrated performance spectra. In *2020 ICPM Doctoral Consortium and Tool Demonstration Track, ICPM-D 2020*, pages 27–30. CEUR-WS. org, 2020. (Cited on pages 36, 43, 313, and 322.)

[55] Thomas G. Dietterich. Machine learning for sequential data: A review. In Terry Caelli, Adnan Amin, Robert P. W. Duin, Dick de Ridder, and Mohamed Kamel,

editors, *Structural, Syntactic, and Statistical Pattern Recognition*, pages 15–30, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. (Cited on pages 37, 344, 345, and 360.)

[56] Dirk Fahland, R.A.J.J. van Delft, S. Esser, S. Habets, M. Heinrich, M. Hrytsenia, E.L. Klijn, O. Koroglu, A. Turu Pi, L. Vugs, A. Karetnikov, J. Leander, J.W.H. Nooyen, I. Vagionitis, S.J. Ruiz Sainz, F. Shafiee, and R. Uku. Reproduction of Experimental Evaluations in Process Mining, September 2019. (Cited on pages 39, 93, 94, and 95.)

[57] Walter Vogler. *Modular Construction and Partial Order Semantics of Petri Nets*, volume 625 of *Lecture Notes in Computer Science*. Springer, 1992. (Cited on page 50.)

[58] José Meseguer, Ugo Montanari, and Vladimiro Sassone. On the semantics of Petri nets. In *International Conference on Concurrency Theory*, pages 286–301. Springer, 1992. (Cited on page 50.)

[59] Luca Castellano, Giorgio De Michelis, and Lucia Pomello. Concurrency versus interleaving: an instructuve example. *Bulletin of the EATCS*, 31:12–14, 1987. (Cited on page 50.)

[60] CPN Tools. A tool for editing, simulating, and analyzing Colored Petri nets. `http://cpntools.org/`. Accessed: 2020-04-24. (Cited on page 56.)

[61] CPN Tools IDE. `https://github.com/cpn-io`. Accessed: 2020-04-24. (Cited on page 56.)

[62] Standard ML of New Jersey. `https://www.smlnj.org/`. Accessed: 2020-04-24. (Cited on page 56.)

[63] Kurt Jensen and Lars M. Kristensen. Colored Petri nets: a graphical language for formal modeling and validation of concurrent systems. *Communications of the ACM*, 58(6):61–70, 2015. (Cited on pages 61 and 152.)

[64] Anahita Farhang Ghahfarokhi, Gyunam Park, Alessandro Berti, and Wil M. P. van der Aalst. OCEL: A standard for object-centric event logs. In *European Conference on Advances in Databases and Information Systems*, pages 169–175. Springer, 2021. (Cited on pages 63, 96, 374, and 383.)

[65] IEEE standard for eXtensible Event Stream (XES) for achieving interoperability in event logs and event streams. *IEEE Std 1849-2016*, pages 1–50, 2016. (Cited on page 63.)

[66] Laura Maruster and Nick R. T. P. van Beest. Redesigning business processes: a methodology based on simulation and process mining techniques. *Knowl. Inf. Syst.*, 21(3):267–297, 2009. (Cited on pages 67, 228, and 257.)

[67] Wil M. P. van der Aalst, Maja Pesic, and Minseok Song. Beyond process mining: From the past to present and future. In *CAiSE*, 2010. (Cited on pages 67 and 156.)

[68] Moe Thandar Wynn, Erik Poppe, J. Xu, Arthur H. M. ter Hofstede, Ross Brown, Azzurra Pini, and Wil M. P. van der Aalst. Processprofiler3d: A visualisation framework for log-based process performance comparison. *Decision Support Systems*, 100:93–108, 2017. (Cited on pages 67 and 156.)

[69] Wil M. P. van der Aalst, Helen Schonenberg, and Minseok Song. Time prediction based on process mining. *Inf. Syst.*, 36:450–475, 2011. (Cited on pages 68 and 158.)

[70] Boudewijn F. van Dongen, Ronald A. Crooy, and Wil M. P. van der Aalst. Cycle time prediction: When will this case finally be finished? In *OTM Conferences*, 2008. (Cited on pages 68 and 158.)

[71] Andreas Rogge-Solti and Mathias Weske. Prediction of business process durations using non-Markovian stochastic Petri nets. *Inf. Syst.*, 54:1–14, 2015. (Cited on pages 68 and 158.)

[72] Daniel A. Keim, Gennady L. Andrienko, Jean-Daniel Fekete, Carsten Görg, Jörn Kohlhammer, and Guy Melançon. Visual analytics: Definition, process, and challenges. In *Information Visualization*, volume 4950 of *LNCS*, pages 154–175. Springer, 2008. (Cited on pages 68 and 157.)

[73] Wil M. P. van der Aalst. Object-centric process mining: Dealing with divergence and convergence in event data. In *International Conference on Software Engineering and Formal Methods*, pages 3–25. Springer, 2019. (Cited on pages 96, 155, and 383.)

[74] Koen Verhaegh. Process mining for systems with automated batching: an exploratory study on new process mining grounds. Master's thesis, Eindhoven University of Technology, 2018. (Cited on pages 116 and 117.)

[75] Sanjay Bose. *An Introduction to Queuing Systems*. 01 2002. (Cited on pages 121, 131, and 144.)

[76] James MacGregor Smith and Barış Tan. *Handbook of stochastic models and analysis of manufacturing system operations*, volume 20013. Springer, 2013. (Cited on pages 130 and 141.)

[77] Zahra Toosinezhad, Dirk Fahland, Ozge Koroglu, and Wil M.P. van der Aalst. Detecting system-level behavior leading to dynamic bottlenecks. In *ICPM2020*, 2020. (Cited on pages 133, 157, 258, 259, 260, 261, 262, and 323.)

[78] Bianka Bakullari and Wil M. P. van der Aalst. High-level event mining: A framework. In *2022 4th International Conference on Process Mining (ICPM)*, pages 136–143. IEEE, 2022. (Cited on pages 133, 157, 258, 260, 262, and 323.)

[79] Arik Senderovich. Queue mining: Service perspectives in process mining. In *BPM (Demos)*, 2017. (Cited on pages 143, 151, and 153.)

[80] Peter Whittle. *Systems in stochastic equilibrium*. John Wiley & Sons, Inc., 1986. (Cited on page 143.)

[81] Gerardo Rubino. Transient analysis of Markovian queueing systems: a survey with focus on closed-forms and uniformization, 2020. (Cited on page 143.)

[82] A.H. Manggala Putri, Retno Subekti, and Nikenasih Binatari. The completion of non-steady-state queue model on the queue system in Dr. Yap Eye Hospital Yogyakarta. In *Journal of Physics.: Conference Series*, volume 855, 2017. (Cited on page 143.)

[83] Ward Whitt. The queueing network analyzer. *The bell system technical journal*, 62(9):2779–2815, 1983. (Cited on page 144.)

[84] Hema Tahilramani, D Manjunath, and Sanjay K Bose. Approximate analysis of open network of ge/ge/m/n queues with transfer blocking. In *MASCOTS'99. Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 164–171. IEEE, 1999. (Cited on page 144.)

[85] Arik Senderovich, Matthias Weidlich, Avigdor Gal, and Avishai Mandelbaum. Queue mining - predicting delays in service processes. In *CAiSE*, 2014. (Cited on pages 144 and 155.)

[86] Arik Senderovich, Sander J.J. Leemans, Shahar Harel, Avigdor Gal, Avishai Mandelbaum, and Wil M.P. van der Aalst. Discovering queues from event logs with varying levels of information. In *International Conference on Business Process Management*, pages 154–166. Springer, 2016. (Cited on page 144.)

[87] Arik Senderovich, J. Christopher Beck, Avigdor Gal, and Matthias Weidlich. Congestion graphs for automated time predictions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4854–4861, 2019. (Cited on pages 144, 145, 146, 151, 153, and 155.)

[88] Avigdor Gal, Arik Senderovich, and Matthias Weidlich. Challenge paper: data quality issues in queue mining. *Journal of Data and Information Quality (JDIQ)*, 9(4):1–5, 2018. (Cited on page 144.)

[89] David Gamarnik, Assaf Zeevi, et al. Validity of heavy traffic steady-state approximations in generalized Jackson networks. *The Annals of Applied Probability*, 16(1):56–90, 2006. (Cited on page 145.)

[90] T.T. Kwo. A theory of conveyors. *Management science*, 5(1):51–71, 1958. (Cited on page 150.)

[91] Eginhard J. Muth. A model of a closed-loop conveyor with random material flow. *AIIE Transactions*, 9(4):345–351, 1977. (Cited on page 150.)

[92] Eginhard J. Muth. Analysis of closed-loop conveyor systems, the discrete flow case. *AIIE Transactions*, 6(1):73–83, 1974. (Cited on page 150.)

[93] Eginhard J. Muth and John A. White. Conveyor theory: a survey. *Aiie Transactions*, 11(4):270–277, 1979. (Cited on page 150.)

[94] F Benson and G Gregory. Closed queueing systems: a generalization of the machine interference model. *Journal of the Royal Statistical Society: Series B (Methodological)*, 23(2):385–393, 1961. (Cited on page 150.)

[95] M. Posner and B. Bernholtz. Two-stage closed queueing systems with time lags. *Journal of the Canadian Operational Research Society*, 5:82–91967, 1967. (Cited on page 150.)

[96] M. Posner and B. Bernholtz. Closed finite queuing networks with time lags. *Operations Research*, 16(5):962–976, 1968. (Cited on page 150.)

[97] M. Posner and B. Bernholtz. Closed finite queuing networks with time lags and with several classes of units. *Operations Research*, 16(5):977–985, 1968. (Cited on page 150.)

[98] H. Mayer. Introduction to conveyor theory. *Western Electric Engineer*, 4(1):43–47, 1960. (Cited on page 150.)

[99] William T. Morris. Analysis for materials handling management. Homewood, Illinois: Richard D. Irwin. 1962. (Cited on page 150.)

[100] Ralph L. Disney. Some multichannel queueing problems with ordered entry. *Journal of Industrial Engineering*, 13(1):46–48, 1962. (Cited on page 150.)

[101] G. Gregory and C.D. Litton. A Markovian analysis of a single conveyor system. *Management Science*, 22(3):371–375, 1975. (Cited on page 150.)

[102] Linda C Schmidt and John Jackman. Modeling recirculating conveyors with blocking. *European Journal of Operational Research*, 124(2):422–436, 2000. (Cited on page 150.)

[103] David Sonderman. An analytical model for recirculating conveyors with stochastic inputs and outputs. *The International Journal Of Production Research*, 20(5):591–605, 1982. (Cited on page 150.)

[104] Ivo Jean-Baptiste François Adan, Onno J. Boxma, and Jacobus Adrianus Cornelis Resing. Queueing models with multiple waiting lines. *Queueing Systems*, 37(1):65–98, 2001. (Cited on page 150.)

[105] Ruben Bossier, Maria Vlasiou, and Ivo Adan. Analytic properties of two-carousel systems. *Probability in the Engineering and Informational Sciences*, 27(1):57–84, 2013. (Cited on page 150.)

[106] Jelmer P van der Gaast, MBM De Koster, and Ivo J.B.F. Adan. Conveyor merges in zone picking systems: a tractable and accurate approximate model. *Transportation Science*, 52(6):1428–1443, 2018. (Cited on page 150.)

[107] Jelmer P. van der Gaast, Rene M.B.M. De Koster, Ivo J.B.F. Adan, and J.A.C. Resing. Modeling and performance analysis of sequential zone picking systems. *Eurandom Preprint Series*, 2012. (Cited on page 150.)

[108] Remco Dijkman, Ivo Adan, and Sander Peters. Advanced queueing models for quantitative business process analysis. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 260–267. IEEE, 2018. (Cited on page 151.)

[109] Sander Peters, Yoav Kerner, Remco Dijkman, Ivo Adan, and Paul Grefen. Fast and accurate quantitative business process analysis using feature complete queueing models. *Information Systems*, 104:101892, 2022. (Cited on page 151.)

[110] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The refined process structure tree. In *International Conference on Business Process Management*, pages 100–115. Springer, 2008. (Cited on page 151.)

[111] Artem Polyvyanyy, Jussi Vanhatalo, and Hagen Völzer. Simplified computation and generalization of the refined process structure tree. In *International Workshop on Web Services and Formal Methods*, pages 25–41. Springer, 2010. (Cited on page 151.)

[112] Bernardo D'Auria, Ivo J.B.F. Adan, René Bekker, and Vidyadhar Kulkarni. An M/M/c queue with queueing-time dependent service rates. *European Journal of Operational Research*, 299(2):566–579, 2022. (Cited on page 151.)

[113] Erjie Ang, Sara Kwasnick, Mohsen Bayati, Erica L. Plambeck, and Michael Aratow. Accurate emergency department wait time prediction. *Manufacturing & Service Operations Management*, 18(1):141–156, 2016. (Cited on page 151.)

[114] Fernando Rosa-Velardo and David de Frutos-Escrig. Name creation vs. replication in Petri net systems. *Fundam. Inform.*, 88(3):329–356, 2008. (Cited on pages 152, 159, 162, and 163.)

[115] Marco Montali and Andrey Rivkin. DB-nets: On the marriage of Colored Petri nets and relational databases. In *Transactions on Petri Nets and Other Models of Concurrency XII*, pages 91–118. Springer, 2017. (Cited on pages 152 and 153.)

[116] Marco Montali and Andrey Rivkin. From DB-nets to Coloured Petri nets with priorities (extended version). *arXiv preprint arXiv:1904.00058*, 2019. (Cited on pages 152 and 153.)

[117] Silvio Ghilardi, Alessandro Gianola, Marco Montali, and Andrey Rivkin. Petri nets with parameterised data: modelling and verification (extended version). *arXiv preprint arXiv:2006.06630*, 2020. (Cited on page 152.)

[118] Sebastian Steinau, Kevin Andrews, and Manfred Reichert. Coordinating large distributed process structures. In *Enterprise, Business-Process and Information Systems Modeling*, pages 19–34. Springer, 2019. (Cited on page 152.)

[119] Wil M. P. van der Aalst, Paulo Barthelmess, Clarence A. Ellis, and Jacques Wainer. Proclets: A framework for lightweight interacting workflow processes. *International Journal of Cooperative Information Systems*, 10(04):443–481, 2001. (Cited on pages 152, 153, and 159.)

[120] Wil M. P. van der Aalst and Alessandro Berti. Discovering object-centric Petri nets. *Fundamenta informaticae*, 175(1-4):1–40, 2020. (Cited on pages 152 and 197.)

[121] Karolin Winter, Florian Stertz, and Stefanie Rinderle-Ma. Discovering instance and process spanning constraints from process execution logs. *Information Systems*, 89:101484, 2020. (Cited on pages 152 and 153.)

[122] Karolin Winter and Stefanie Rinderle-Ma. Defining instance spanning constraint patterns for business processes based on proclets. In *Conceptual Modeling: 39th International Conference, ER 2020, Vienna, Austria, November 3–6, 2020, Proceedings 39*, pages 149–163. Springer, 2020. (Cited on page 153.)

[123] Florian Stertz, Karolin Winter, and Stefanie Rinderle-Ma. Discovering instance spanning exceptions from process execution logs. In *2022 IEEE 24th Conference on Business Informatics (CBI)*, volume 2, pages 49–56. IEEE, 2022. (Cited on page 153.)

[124] RP Jagadeesh Chandra Bose and Wil M. P. van der Aalst. Process diagnostics using trace alignment: opportunities, issues, and challenges. *Information Systems*, 37(2):117–141, 2012. (Cited on page 154.)

[125] Paolo Felli, Alessandro Gianola, Marco Montali, Andrey Rivkin, and Sarah Winkler. Conformance checking with uncertainty via SMT (extended version). *arXiv preprint arXiv:2206.07461*, 2022. (Cited on pages 154 and 380.)

[126] Marco Pegoraro and Wil M.P. van der Aalst. Mining uncertain event data in process mining. In *2019 International Conference on Process Mining (ICPM)*, pages 89–96. IEEE, 2019. (Cited on page 154.)

[127] Marco Pegoraro, Merih Seran Uysal, and Wil M. P. van der Aalst. Discovering process models from uncertain event data. In *International Conference on Business Process Management*, pages 238–249. Springer, 2019. (Cited on page 154.)

[128] Marco Pegoraro, Merih Seran Uysal, and Wil M. P. van der Aalst. Conformance checking over uncertain event data. *Information Systems*, 102:101810, 2021. (Cited on page 154.)

[129] Linh Thao Ly, Stefanie Rinderle-Ma, David Knuplesch, and Peter Dadam. Monitoring business process compliance using compliance rule graphs. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 82–99. Springer, 2011. (Cited on page 154.)

[130] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. Verification of gsm-based artifact-centric systems through finite abstraction. In *International Conference on Service-Oriented Computing*, pages 17–31. Springer, 2012. (Cited on page 154.)

[131] Felix Mannhardt, Massimiliano De Leoni, Hajo A Reijers, and Wil M.P. van der Aalst. Balanced multi-perspective checking of process conformance. *Computing*, 98(4):407–437, 2016. (Cited on pages 154 and 210.)

[132] Khalil Mecheraoui, Julio C. Carrasquel, and I. Lomazova. Compositional conformance checking of nested Petri nets and event logs of multi-agent systems. *ArXiv*, abs/2003.07291, 2020. (Cited on page 154.)

[133] Dirk Fahland, Massimiliano De Leoni, Boudewijn F. Van Dongen, and Wil M.P. van der Aalst. Behavioral conformance of artifact-centric process models. In *International Conference on Business Information Systems*, pages 37–49. Springer, 2011. (Cited on pages 154, 155, 212, and 213.)

[134] Dirk Fahland, Massimiliano De Leoni, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Conformance checking of interacting processes with overlapping instances. In *International Conference on Business Process Management*, pages 345–361. Springer, 2011. (Cited on pages 154 and 155.)

[135] Noah Gans, Ger Koole, and Avishai Mandelbaum. Telephone call centers: Tutorial, review, and research prospects. *Manufacturing & Service Operations Management*, 5:79–141, 03 2003. (Cited on page 155.)

[136] Lawrence Brown, Noah Gans, Avishai Mandelbaum, Anat Sakov, Haipeng Shen, Sergey Zeltyn, and Linda Zhao. Statistical analysis of a telephone call center. *Journal of the American Statistical Association*, 100(469):36–50, 2005. (Cited on page 155.)

[137] Suriadi Suriadi, Robert Andrews, Arthur H.M. ter Hofstede, and Moe Thandar Kyaw Wynn. Event log imperfection patterns for process mining: Towards a systematic approach to cleaning event logs. *Information Systems*, 64:132 – 150, 2017. (Cited on pages 155 and 220.)

[138] Raffaele Conforti, Marcello La Rosa, and Arthur ter Hofstede. Timestamp repair for business process event logs. Technical report, 2018/04/05 2018. (Cited on pages 155 and 220.)

[139] Niels Martin, Benoît Depaire, An Caris, and Dimitri Schepers. Retrieving the resource availability calendars of a process from an event log. *Information Systems*, 88:101463, 2020. (Cited on page 155.)

[140] Viara Popova, Dirk Fahland, and Marlon Dumas. Artifact lifecycle discovery. *International Journal of Cooperative Information Systems*, 24(01):1550001, 2015. (Cited on page 155.)

[141] Xixi Lu, Marijn Nagelkerke, Dennis van de Wiel, and Dirk Fahland. Discovering interacting artifacts from ERP systems. *IEEE Transactions on Services Computing*, 8(6):861–873, 2015. (Cited on page 155.)

[142] Michael Werner and Nick Gehrke. Multilevel process mining for financial audits. *IEEE Transactions on Services Computing*, 8(6):820–832, 2015. (Cited on page 156.)

[143] Stefan Esser and Dirk Fahland. Storing and querying multi-dimensional process event logs using graph databases. In *International Conference on Business Process Management*, pages 632–644. Springer, 2019. (Cited on pages 156 and 383.)

[144] Alessandro Berti and Wil M.P. van der Aalst. Extracting multiple viewpoint models from relational databases. In *Data-Driven Process Discovery and Analysis*, pages 24–51. Springer, 2018. (Cited on page 156.)

[145] Stefan Esser and Dirk Fahland. Multi-dimensional event data in graph databases. *Journal on Data Semantics*, pages 1–33, 2021. (Cited on pages 156 and 383.)

[146] John C. Nash. The (Dantzig) simplex method for linear programming. *Computing in Science & Engineering*, 2(1):29–31, 2000. (Cited on pages 156 and 251.)

[147] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Using life cycle information in process discovery. In *BPM Workshops 2015*, volume 256 of *LNBIP*, pages 204–217. Springer, 2015. (Cited on page 156.)

[148] Arik Senderovich, Matthias Weidlich, and Avigdor Gal. Temporal network representation of event logs for improved performance modelling in business processes. In *BPM 2017*, volume 10445 of *LNCS*, pages 3–21. Springer, 2017. (Cited on page 156.)

[149] Minseok Song and Wil M. P. van der Aalst. Supporting process mining by showing events at a glance. In *Proceedings of the 17th Annual Workshop on Information Technologies and Systems (WITS)*, pages 139–145, 2007. (Cited on page 157.)

[150] Ayush Shrestha, Ben Miller, Ying Zhu, and Yi Zhao. Storygraph: Extracting patterns from spatio-temporal data. In *ACM SIGKDD Workshop IDEA'13*, pages 95–103. ACM, 2013. (Cited on page 157.)

[151] Vadim Denisov, Elena Belkina, and Dirk Fahland. BPIC'2018: Mining concept drift in performance spectra of processes. In *8th International Business Process Intelligence Challenge*, 2018. (Cited on pages 157 and 376.)

[152] Francesco Folino, Massimo Guarascio, and Luigi Pontieri. Discovering high-level performance models for ticket resolution processes. In Robert Meersman, Hervé Panetto, Tharam Dillon, Johann Eder, Zohra Bellahsene, Norbert Ritter, Pieter De Leenheer, and Deijing Dou, editors, *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*, pages 275–282, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. (Cited on page 158.)

[153] Geetika T. Lakshmanan, Davood Shamsi, Yurdaer N. Doganata, Merve Unuvar, and Rania Khalaf. A Markov prediction model for data-driven semi-structured business processes. *Knowledge and Information Systems*, 42(1):97–126, Jan 2015. (Cited on page 158.)

[154] Andreas Rogge-Solti, Wil M.P. van der Aalst, and Mathias Weske. Discovering stochastic Petri nets with arbitrary delay distributions from event logs. In *BPM Workshops 2013*, volume 171 of *LNBIP*, pages 15–27. Springer, 2014. (Cited on page 158.)

[155] Arik Senderovich, Andreas Rogge-Solti, Avigdor Gal, Jan Mendling, Avishai Mandelbaum, Sarah Kadish, and Craig A. Bunnell. Data-driven performance analysis of scheduled processes. In *BPM 2015*, volume 9253 of *LNCS*, pages 35–52. Springer, 2015. (Cited on page 158.)

[156] Arik Senderovich, Matthias Weidlich, Avigdor Gal, and Avishai Mandelbaum. Queue mining for delay prediction in multi-class service processes. *Inf. Syst.*, 53:278–295, 2015. (Cited on page 158.)

[157] Alfredo Cuzzocrea, Francesco Folino, Massimo Guarascio, and Luigi Pontieri. Predictive monitoring of temporally-aggregated performance indicators of business processes against low-level streaming events. *Information Systems*, 81:236 – 266, 2019. (Cited on page 158.)

[158] Matthias Ehrendorfer, Juergen Mangler, and Stefanie Rinderle-Ma. Assessing the impact of context data on process outcomes during runtime. In *International Conference on Service-Oriented Computing*, pages 3–18. Springer, 2021. (Cited on page 158.)

[159] Abbas Khosravi, Saeid Nahavandi, and Doug Creighton. Estimating performance indexes of a baggage handling system using metamodels. *Proceedings of the IEEE International Conference on Industrial Technology*, pages 1 – 6, 03 2009. (Cited on page 158.)

[160] Tanvir Ahmed, Torben Bach Pedersen, Toon Calders, and Hua Lu. Online risk prediction for indoor moving objects. In *2016 17th IEEE International Conference on Mobile Data Management (MDM)*, volume 1, pages 102–111, June 2016. (Cited on pages 158 and 229.)

[161] Jacek Skorupski, Piotr Uchroński, and Adrian Łach. A method of hold baggage security screening system throughput analysis with an application for a medium-sized airport. *Transportation Research Part C: Emerging Technologies*, 88:52 – 73, 2018. (Cited on page 158.)

[162] Fernando Rosa-Velardo, Olga Marroquín Alonso, and David de Frutos-Escrig. Mobile synchronizing Petri nets: A choreographic approach for coordination in ubiquitous systems. *Electr. Notes Theor. Comput. Sci.*, 150(1):103–126, 2006. (Cited on pages 159, 162, and 163.)

[163] Kees M. van Hee, Natalia Sidorova, Marc Voorhoeve, and Jan Martijn E. M. van der Werf. Generation of database transactions with Petri nets. *Fundam. Inform.*, 93(1-3):171–184, 2009. (Cited on pages 159, 162, and 163.)

[164] Wil M. P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998. (Cited on page 163.)

[165] Vladimir A Bashkin and Irina A Lomazova. Decidability of-soundness for workflow nets with an unbounded resource. In *Transactions on Petri Nets and Other Models of Concurrency IX*, pages 1–18. Springer, 2014. (Cited on page 197.)

[166] Kees Van Hee, Alexander Serebrenik, Natalia Sidorova, and Marc Voorhoeve. Soundness of resource-constrained workflow nets. In *Applications and Theory of Petri Nets 2005: 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005. Proceedings 26*, pages 250–267. Springer, 2005. (Cited on page 197.)

[167] Natalia Sidorova and Christian Stahl. Soundness for resource-constrained workflow nets is decidable. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 43(3):724–729, 2012. (Cited on page 197.)

[168] Guanjun Liu, Changjun Jiang, and Mengchu Zhou. Time-soundness of time petri nets modeling time-critical systems. *ACM Transactions on Cyber-Physical Systems*, 2(2):1–27, 2018. (Cited on pages 197 and 198.)

[169] Irina A Lomazova, Alexey A Mitsyuk, and Andrey Rivkin. Soundness in object-centric workflow petri nets. *arXiv preprint arXiv:2112.14994*, 2021. (Cited on pages 197 and 198.)

[170] Anne Rozinat and Wil M. P. van der Aalst. Decision mining in ProM. In *International Conference on Business Process Management*, pages 420–425. Springer, 2006. (Cited on page 206.)

[171] George B. Dantzig. Origins of the simplex method. In *A history of scientific computing*, pages 141–151. 1990. (Cited on page 251.)

[172] Ozge Köroglu. *Outlier Detection in Event Logs of Material Handling System*. PhD thesis, Master's thesis, Eindhoven University of Technology, 2019. (Cited on page 260.)

[173] Arik Senderovich, Chiara Di Francescomarino, Chiara Ghidini, Kerwin Jorbina, and Fabrizio Maria Maggi. Intra and inter-case features in predictive process monitoring: A tale of two dimensions. In *International Conference on Business Process Management*, pages 306–323. Springer, 2017. (Cited on pages 331 and 334.)

[174] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. (Cited on pages 369 and 382.)

[175] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014. (Cited on page 369.)

[176] Wil M. P. van der Aalst, Daniel Tacke Genannt Unterberg, Vadim Denisov, and Dirk Fahland. Visualizing token flows using interactive performance spectra. In Ryszard Janicki, Natalia Sidorova, and Thomas Chatain, editors, *Application and Theory of Petri Nets and Concurrency*, pages 369–380, Cham, 2020. Springer International Publishing. (Cited on page 379.)

[177] Eli Bogdanov, Izack Cohen, and Avigdor Gal. Conformance checking over stochastically known logs. In *Business Process Management Forum: BPM 2022 Forum, Münster, Germany, September 11–16, 2022, Proceedings*, pages 105–119. Springer, 2022. (Cited on page 380.)

[178] Wil M.P. van der Aalst and Josep Carmona. *Process mining handbook*. Springer Nature, 2022. (Cited on page 383.)

Index

Summary

Process Mining for Systems with Shared Resources and Queues— Process Modeling, Conformance Checking, and Performance Analysis

Organizations transform digitally to improve their processes, implement new business models, and develop new capabilities. Information systems execute their processes and store detailed data about the execution progress and outcome for various needs. Recently, the vast amount of such data, being intensively collected due to cheap storage solution availability, triggered extensive developments in data science, including the emergence of *process mining*. Process mining is a field of data science that exploits data about the execution of business processes, typically referred to as *event logs*, for identifying process improvements and providing operational support. It is achieved through such *tasks* as the *data-driven discovery* of *process models*, *conformance* and *compliance checking*, and *performance analysis* and *monitoring*.

Historically, most process mining techniques address the analysis of *process instances*, or *cases*, in *isolation*, i.e., assuming that various cases do not affect each other. However, this assumption does not hold for many business processes, for example, when cases *interact* on *limited shared resources*. If this is the case, applying many existing process mining techniques is infeasible as it would lead to poor or even inaccurate results.

In this dissertation, we study *material handling processes* of Material Handling Systems (MHSs) in logistics, such as Baggage Handling Systems (BHSs) of airports, or warehouse solutions. In MHSs, cases are not isolated. For instance, passenger bags in BHSs interact on conveyors of finite capacity while competing for shared machines. The primary concern of MHS operators is to keep the MHS performance at the desired level. It makes improving material handling processes and providing operational support an actual problem. However, existing process mining techniques fail to capture interactions between cases. This dissertation aims to bridge this gap by adapting existing techniques and creating new ones, primarily targeting MHSs.

We start with proposing the *performance spectrum* in Chapter 3. It is a generic technique for process performance description, capable of revealing case interactions and various performance phenomena, which we describe in a taxonomy of *performance patterns*. Then, we investigate core aspects affecting the behavior of MHSs in Chapter 4. For that, we explore state-of-the-art queueing theory models for MHS performance analysis, consider their fundamental assumptions, and validate them using the performance spectrum. We show why they do not hold for the MHSs we study but also identify the key concepts for modeling MHSs: queues, resources, and routing functions. Further, we design a *Process-Queue-Resource system* (PQR-system) by materializing these concepts in a *modular* process model in Chapter 6. This model is a dedicated *synchronous proclet system*, whose modules (proclets) represent the process, queues, and resources of an MHS, and whose synchronization channels describe the proclets interactions.

Next, we build on the performance spectrum and PQR-system to extend existing techniques and create new ones. Thus, we adopt the concept of *generalized conformance checking* in Chapter 7. We consider how the problem of PQR-system-based conformance checking can be decomposed into simpler tasks for which existing approaches can be used. Then, we propose a novel method for inferring missing events with timestamps for the log repair task of generalized conformance checking to address the common problem of the incompleteness of MHS event data.

Further, we propose a way to align performance spectra to PQR-systems in Chapter 8. As a result, we obtain the performance description of the queue and resource dimensions (besides the "classical" control flow dimension). Exploiting information about performance patterns in performance spectra, and possible ways of their *propagation* in the system along the PQR-system paths, we propose a method for root-cause performance analysis. It detects problems in the performance spectrum of the control-flow dimension and identifies their root causes in the spectra of the queue and resource dimensions.

Finally, we address the problem of Predictive Performance Monitoring (PPM) in Chapter 9. We exploit the ability of the performance spectrum to capture the system dynamics to formulate a large class of PPM problems as a generic regression problem over the spectrum. Furthermore, we suggest a PQR-system-based method for selecting features relevant to learning the corresponding regression models.

The proposed techniques have been evaluated in controlled experiments using synthetic event logs, generated by a simulation model, and the real data of MHSs built by Vanderlande, an MHS manufacturer. The evaluation of performance spectrum-based analysis allowed us to identify the root causes of a severe performance incident in a major European airport BHS significantly quicker than the existing techniques used by the domain experts. As a result, the corresponding tool was implemented internally by Vanderlande and successfully evaluated on other MHSs. Additionally, an empirical exploration of performance spectra of event logs, recorded by processes

outside the MHS domain, showed that untrained analysts were able to identify the performance patterns unambiguously.

Evaluation of our method for inferring missing events showed accurate results with synthetic data for which the ground truth was available, and a small error in the estimated load using the real data. Finally, the ML models for PPM, trained on the feature sets extracted with our method, showed feasible results for predicting load on critical areas of BHSs, and peaks of undesirable re-circulation on the sorting loops. Open-source implementations for all the methods have been made available as a ProM plugin and several stand-alone tools.

Curriculum Vitæ

Vadim Vladimirovich Denisov was born on 29-11-1977 in Leningrad, USSR. He studied Computer Science at Saint-Petersburg State Electrotechnical University in Saint-Petersburg, Russia. In 2000 he graduated with an M.S. in Computer Science. From 2000 to 2016, he worked in the software development industry.

From 2016 on, he started a Ph.D. project in the Department of Mathematics and Computer Science at Eindhoven University of Technology in Eindhoven, the Netherlands, under the supervision of dr. Dirk Fahland and prof.dr.ir. Wil M.P. van der Aalst, of which the results are presented in this dissertation. Since 2020 he has been a process scientist and software engineer with Process Optimization, Future Products at ServiceNow, Amsterdam, the Netherlands.

List of Publications

Vadim Denisov has the following publications:

Proceedings and Workshop Contributions

- Vadim Denisov, Dirk Fahland, and Wil M. P. van der Aalst. Multi-dimensional performance analysis and monitoring using integrated performance spectra. In *2020 ICPM Doctoral Consortium and Tool Demonstration Track, ICPM-D 2020*, pages 27–30. CEUR-WS. org, 2020

- Vadim Denisov, Dirk Fahland, and Wil M. P. van der Aalst. Repairing event logs with missing events to support performance analysis of systems with shared resources. In Ryszard Janicki, Natalia Sidorova, and Thomas Chatain, editors, *Application and Theory of Petri Nets and Concurrency*, pages 239–259, Cham, 2020. Springer International Publishing

- Vadim Denisov, Dirk Fahland, and Wil M. P. van der Aalst. Predictive performance monitoring of material handling systems using the performance spectrum. In *International Conference on Process Mining, ICPM 2019, Aachen, Germany, June 24-26, 2019*, pages 137–144. IEEE, 2019

- Wil M. P. van der Aalst, Daniel Tacke Genannt Unterberg, Vadim Denisov, and Dirk Fahland. Visualizing token flows using interactive performance spectra. In Ryszard Janicki, Natalia Sidorova, and Thomas Chatain, editors, *Application and Theory of Petri Nets and Concurrency*, pages 369–380, Cham, 2020. Springer International Publishing

- Vadim Denisov, Dirk Fahland, and Wil M. P. van der Aalst. Unbiased, fine-grained description of processes performance from event data. In Mathias Weske, Marco Montali, Ingo Weber, and Jan vom Brocke, editors, *Business Process Management - 16th International Conference, BPM 2018, Sydney, NSW, Australia, September 9-14, 2018, Proceedings*, volume 11080 of *Lecture Notes in Computer Science*, pages 139–157. Springer, 2018

- Vadim Denisov, Elena Belkina, Dirk Fahland, and Wil M. P. van der Aalst. The performance spectrum miner: Visual analytics for fine-grained performance analysis of processes. In Wil M. P. van der Aalst, Fabio Casati, Raffaele Conforti, Massimiliano de Leoni, Marlon Dumas, Akhil Kumar, Jan Mendling, Surya Nepal, Brian T. Pentland, and Barbara Weber, editors, *Proceedings of the Dissertation Award, Demonstration, and Industrial Track at BPM 2018 co-located with 16th International Conference on Business Process Management (BPM 2018), Sydney,*

Australia, September 9-14, 2018, volume 2196 of *CEUR Workshop Proceedings*, pages 96–100. CEUR-WS.org, 2018

Journals

- Dirk Fahland, Vadim Denisov, and Wil M.P. van der Aalst. Inferring unobserved events in systems with shared resources and queues. *Fundamenta Informaticae*, 183(3-4):203–242, 2021

Technical Reports (Non-Refereed)

- Vadim Denisov, Elena Belkina, and Dirk Fahland. BPIC'2018: Mining concept drift in performance spectra of processes. In *8th International Business Process Intelligence Challenge*, 2018