

Inferring Unobserved Events in Systems with Shared Resources and Queues

Dirk Fahland*, Vadim Denisov

Eindhoven University of Technology

Eindhoven, the Netherlands

{d.fahland, v.denisov}@tue.nl

Wil. M.P. van der Aalst

Process and Data Science (Informatik 9)

RWTH Aachen University, Aachen, Germany

wvdaalst@pads.rwth-aachen.de

Abstract. To identify the causes of performance problems or to predict process behavior, it is essential to have correct and complete event data. This is particularly important for distributed systems with shared resources, e.g., one case can block another case competing for the same machine, leading to inter-case dependencies in performance. However, due to a variety of reasons, real-life systems often record only a subset of all events taking place. To understand and analyze the behavior and performance of processes with shared resources, we aim to reconstruct bounds for timestamps of events in a case that must have happened but were not recorded by inference over events in other cases in the system. We formulate and solve the problem by systematically introducing multi-entity concepts in event logs and process models. We introduce a partial-order based model of a multi-entity event log and a corresponding compositional model for multi-entity processes. We define PQR-systems as a special class of multi-entity processes with shared resources and queues. We then study the problem of inferring from an incomplete event log unobserved events and their timestamps that are globally consistent with a PQR-system. We solve the problem by reconstructing unobserved traces of resources and queues according to the PQR-model and derive bounds for their timestamps using a linear program. While the problem is illustrated for material handling systems like baggage handling systems in airports, the approach

*Address for correspondence: TU Eindhoven, PO Box 513, 5600MB Eindhoven, NL

can be applied to other settings where recording is incomplete. The ideas have been implemented in ProM and were evaluated using both synthetic and real-life event logs.

Keywords: Log repair, Process mining, Performance analysis, Multi-entity modeling, Multi-entity event logs, Conformance checking, Material handling systems

1. Introduction

Precise knowledge about actual process behavior and performance is required for identifying causes of performance issues [1], as well as for predictive process monitoring of important process performance indicators [2]. For Material Handling Systems (MHS), such as Baggage Handling Systems (BHS) of airports, performance incidents are usually investigated offline, using recorded event data for finding root causes of problems [3], while online event streams are used as input for predictive performance models [4]. Both analysis and monitoring heavily rely on the completeness and accuracy of 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, an MHS tracks the location of an item, e.g., a bag or box, via hardware sensors placed throughout the system, generating tracking events for 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 onto another. Moreover, even when a sensor is installed, an event still can be discarded to save storage space. As a result, the recorded event data of an MHS are typically incomplete, hampering analysis based on such incomplete data. Therefore, it is essential to repair the event data before analysis. Fig. 1 shows a simple MHS where events are not always recorded. The process model is given and for two cases the recorded incomplete sets of events are depicted using the so-called *Performance Spectrum* [3].

Fig. 1(b) shows item 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 information in which order both cases traversed the *segment* $m4 \rightarrow d1$. Naively interpolating the movement of both items, as shown in Fig. 1(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, Fig. 1(c) and Fig. 1(d) show two possible behaviors that 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

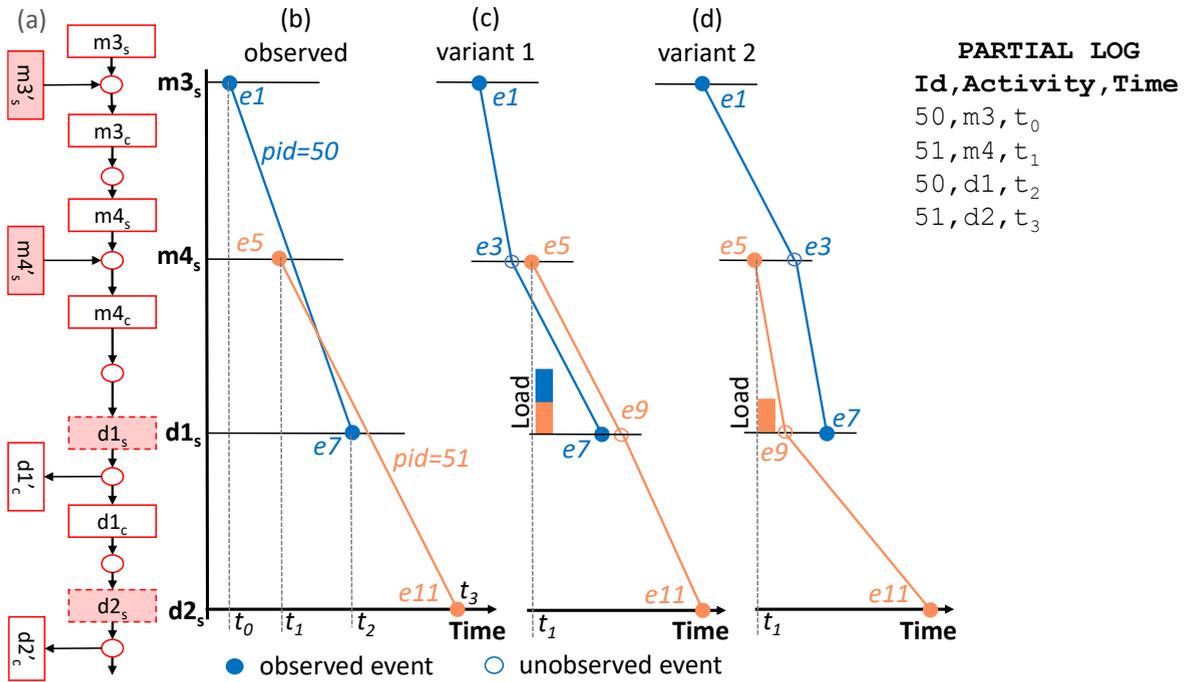


Figure 1. An MHS model example (a), observed imprecise behavior for two cases 50 and 51 (b), possible actual behaviors (c,d).

the potential error. Errors in load, for example, make performance outlier analysis [3] or short-term performance prediction [5] rather difficult. Errors in order impede root-cause analysis of performance outliers, e.g., finding the cases that caused or were affected by outlier behavior.

Problem. In this paper, we address a novel type of problem as illustrated in Fig. 1 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 *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. More specifically, we consider the following information as given: (1) an event log L_1 containing the case identifier, activity and time for recorded events where intermediate steps are not recorded (i.e., the event log may be incomplete), (2) a model of the process (i.e., possible paths for handling each individual case), and (3) a description (model) of the resources involved in each step (e.g., queues, single server resources and their performance parameters such as processing and waiting time). Based on the above input, we want to provide a complete event log L_2 that describes (1) for each case the exact sequence of process steps, (2) and for each unobserved event a time-window of earliest and latest occurrence of the event so that (3) either all earliest or all latest timestamps altogether describe a consistent execution of the entire process over all shared resources.

We elaborate the problem further in Sect. 2 where we discuss related work. Specifically, prior work either only considers the case or the resource perspective explicitly, making implicit assumptions about their complex interplay. The goal of this paper is to explicitly account for the interplay of control-flow, resources, and queues in the entire *system*. This requires us to first identify and develop suitable formal concepts that allow us to precisely state and automatically solve the above problem of inferring missing events and their time-stamps in a way that considers all perspectives jointly.

Contribution. We approach the problem under the conceptual lens of treating each process case, resource, or queue as a separate entity exhibiting its own behavior. System behavior then is the result of multiple entities synchronizing in joint steps, e.g., when a resource starts working on a case. Section 3 further develops the problem of inferring missing events under this conceptual lens. To solve the problem, the paper systematically introduces multi-entity concepts in formal models for event logs and in process models with the following four contributions.

(1) To ground the problem in existing types of event data, we propose in Sect. 4 an alternative definition of event logs that can handle multiple entity identifiers. The information is carried in an event table with multiple entity identifier columns. We then show that the information in this table can be viewed from two different but equivalent perspectives: (i) as a family of sequential event logs, one per entity type; and (ii) as a global strict partial order over all events that is typed with entity types and can be understood as a *system-level run*. This model allows us to conceptually decompose behavior (run of a system or event data) into individual *entity traces* of process cases, resources, and queues. Different entity traces synchronize when a resource or queue is involved in a case, allowing to explicitly describe their many-to-many relations in the run.

(2) To provide a well-defined problem of repairing incomplete event logs, we develop a novel conceptual model for processes with shared resources in Sect. 5. We extend the recently proposed synchronous proctet model [6] with concepts of coloured Petri nets [7] to precisely describe queueing and timed behavior in systems with multiple synchronizing entities, resulting in the model of *CPN proctet systems*. We provide a replay semantics for CPN proctet systems that defines when a model accepts a given event log. Our semantics is compositional: the system can replay the log iff each component can replay the part of the log it relates to. A side product of this work is that we also provide a semantics for replaying event logs on regular coloured Petri nets.

(3) We then formalize a special class of CPN proctet systems called *PQR systems* which are composed of one component for the process, and multiple components for shared resources and queues. PQR systems allow to model processes where each step is served by one single-server resource and resources are connected by strict FIFO queues only. These assumptions are reasonable for a large class of MHSs.

(4) We then provide an automated technique to solve the problem for PQR systems where the process is *acyclic* which suffices for many real-life problem instances. The central solution idea given in Sect. 6 is to decompose the behavioral information in the incomplete event log into entity traces. We gradually infer unobserved events and unobserved entity traces and their synchronization with other entities from the component-based structure of the PQR system. We then formulate a Linear Programming (LP) problem [8] to infer upper and lower bounds of timestamps of unobserved events based on bounds of timestamps along the different entity traces.

We evaluated the approach by comparing the restored event logs with the ground truth for synthetic logs and estimate errors for real-life event logs for which the ground truth is unavailable (Sect. 7). We discuss our findings and alleys for future work in Sect. 8.

2. Related work

In all operational processes (logistics, manufacturing, healthcare, education and so on) complete and precise event data, including information about workload and resource utilization, is highly valuable since it allows for process mining techniques uncovering compliance and performance problems. Event data can be used to replay processes on top of process models [9], to predict process behavior [10, 5], or to visualize detailed process behavior using performance spectra [3]. All of these techniques rely on complete and correct event data. Since this is often not the case, we aim to transform *incomplete* event data into *complete* event data.

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 [11], queueing theory models can be used for load predictions under assumptions about distributions of unobserved parameters, such as customer patience duration [12], while assuming high load snapshot principle predictors show better accuracy [13]. For time predictions in congested systems, the required features are extracted using congestion graphs [14] mined using queueing 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 [15]. The taxonomy specifically discusses how to detect and correct inadvertent time intervals (i.e., time stamps recorded later than the occurrence of the event) through domain knowledge; no automatic technique is provided. The timestamp repair technique in [16] automatically reconstructs the most likely order of wrongly recorded events and most likely intervals for timestamps based on other traces; the technique assumes all events were recorded and does not consider ordering constraints due to resources involved across traces. In [17] resource availability calendars are retrieved from event logs without the use of a process model, but assuming *start* and *complete* life-cycle transitions as well as a case arrival time present in a log. Using a process model, classical trace alignment algorithms [18] restore missing events but do not restore their timestamps. The authors conclude (see [18], 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. Recently, also techniques for process discovery and conformance checking over uncertain event data were presented [19, 20]. The output of our approach can provide the input needed for these techniques.

Multiple recent works address behavioral models for behavior over multiple different entities in one-to-many and many-to-many relations. The model of proplets thereby defines one behavioral model (a Petri net) per entity. Entities interact asynchronously via message exchange [21] or synchronously via dynamic transition synchronization [6]. Object-centric Petri nets [22] are a special class of coloured Petri nets [7] that are structured to model the flow and synchronization of different objects (or entities); they correspond to synchronous proplets [6] where the synchronization has been materialized in the model structure. Catalog nets [23] approach the problem from the side databases and model entity

behavior by describing database updates through transitions; entity synchronization is similar to synchronous proclats. Process structures [24] integrate relational modeling and behavioral modeling but are using dedicated behavioral model without existing analysis techniques. None of these works so far considered system-level entities such as queues and resources as part of the model to study how system-level entities impact process behavior. Further, none of these works has provided any techniques for reasoning about missing temporal and behavioral information across different entities.

Also data models for event data over multiple entities have been studied extensively in three forms. One type of event logs describe entities just as a sequence (or collection) of events [25, 26] where each event carries multiple entity identifier attributes, possibly even having multiple entity identifier values. Behavioral analysis requires to extract a trace per entity, thereby constructing a set of related sequential event logs [25, 27]. 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 [28, 29, 30, 31]. In this paper, we show that the three representations are essentially equivalent and just materialize the data in different forms; reasoning about incomplete behavior across multiple entities benefits from being able to switch between these perspectives arbitrarily. We thereby adopt a more classical partial-order model instead of a graph as it simplifies reasoning.

Our work contributes to the problem of reconstructing behavior of cases and limited shared resources for which the cases compete. We use the notion of *proclats* first introduced in [21] and adapted for process mining in [6] to approach the problem from control-flow and resource perspectives at once. We assume a system model given as a composition of a control-flow proclat (process) and resource/queue proclats. The given event log is a set of events with multiple entity identifiers. We restore missing events through classical trace alignments over control-flow proclats. The dynamic synchronization of proclats [6] allows us to infer how and when sequential traces of resource entities must have traversed over the control-flow steps, which we express as a linear programming problem to compute time stamp intervals for the restored events. For the construction of the linear program we make extensive use of the partial ordering of events. Event logs repaired in this way enable the use of analysis which assume event logs to be complete.

Compared to a prior version of this article [32], we here provide a complete formalization of the problem and all underlying concepts, including the definitions of multi-entity event logs, CPN proclat systems and their replay semantics, and a formal definition of PQR systems.

3. Modeling inter-case behavior via shared resources

Prior work (cf. Sect. 2) approaches the problem of analyzing the performance of systems with shared resources primarily either from the control-flow perspective [17, 19, 20, 10, 5] or the resource/queuing perspective [11, 12, 13, 14], leading to information loss about the other perspective. In the following, we show how to conceptualize the problem from both perspectives at once using *synchronous proclats* [6] extended with a few concepts of coloured Petri Nets [7]. This way we are able to capture both control-flow and resource dynamics and their interaction as synchronizing entity traces. We introduce the model in Sect 3.1 and use it to illustrate how incomplete logging incurs information loss for performance analysis in Sect. 3.2.

3.1. Processes-aware systems with shared resources

We explain the dynamics of process-aware systems over shared resources using a BHS handling luggage. The process control-flow takes a bag from a source (e.g., check-in or transfer from another flight), to a destination (e.g., the airplane, transfer) along intermediate process steps (e.g., baggage scanning, storage). BHS resources are primarily single-server machines (e.g., baggage scanners) connected via conveyor belts, i.e., FIFO queues. Fig. 2(a) shows a typical system design pattern involving the control-flow and resource perspective: four parallel check-in desks ($c1-c4$) merge into one *linear conveyor* through *merge points* ($m2-m4$). *Divert points* ($d1$ and $d2$) can route bags from the linear conveyor to *scanners* ($s1$ and $s2$). Each merge point and scanner is preceded by a FIFO queue for buffering incoming cases (bags) in case the corresponding resource is busy. Fig. 2(b) shows the plain control-flow of this BHS (also called Material Flow Diagram (MFD)). A real-life BHS may contain hundreds of process steps and resources, and conveyors may also form loops. Each processing step in a BHS is served by a limited number of resources (in case of machines exactly one) with a minimum *processing time* and often a minimum *waiting time* to ensure sufficient “operating space” p between two subsequent bags as shown in Fig. 2(a). Similarly, the conveyor belts realizing FIFO queues have certain operating speeds which determine a minimum *waiting time* to reach the end of queue.

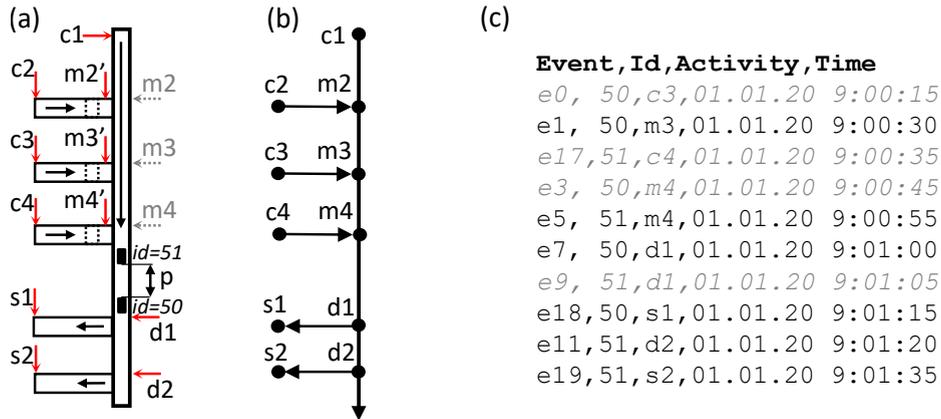


Figure 2. A baggage handling system fragment (a) and its material flow diagram (b). Conveyor belts of check-in counters $c1 - c4$ merge at points $m2 - m4$, further downstream bags can divert at $d1$ and $d2$ to X-Ray security scanners $s1$ and $s2$. Red arrows show sensor (logging) locations. An example of an incomplete event log of the system in (a) is shown in (c), where missing events are shown in the grey color.

Modeling with Coloured Petri Nets. Fig. 3 shows a coloured Petri Net (CPN) model for the segment from check-in $c1$ to merge step $m2$. In the model, transitions $c1_s$ and $c1_c$ describe *start* and *completion* of the check-in step $c1$. When $c1_s$ occurs, arc inscription νpid produces a new identifier value for a bag (also called coloured token) on place *busy* and the single token on place *capacity_{c1}* is removed, i.e., no more resource is available to start $c1$ for another pid . By annotation $@tsr_{c1}$, the new pid on *busy* can only be consumed by transition $c1_c$ (to complete step $c1$) after service-time tsr_{c1} has passed. When $c1_c$ occurs, a token is produced on *capacity_{c1}* and waiting time twr_{c1} has to pass before $c1_s$ can occur

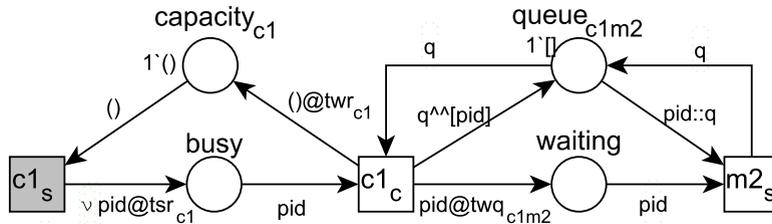


Figure 3. Coloured Petri net model of conveyor $c1 : m2$ of Fig 2.

again. Also, the bag identified by pid is removed from *busy* and inserted into a FIFO queue (modeling a conveyor belt) between the end of $c1$ and the start of $m2$. Arc annotation $q^{\wedge}[pid]$ specifies that bag pid is appended to the end of queue q . Producing pid with time annotation $@twq_{c1m2}$ on place *waiting* models the minimum time it takes for a bag to travel from $c1$ to $m2$. Only then a bag may leave the queue at transition $m2_s$ where arc annotation $id::q$ specifies that bag pid at the head is removed from the tail of the queue q .

The CPN model in Fig. 3 describes the impact of limited resource capacity and queues on the progress of a case, but does not model the resource and the queue as entities themselves. This makes it impossible to reason about resource and queue behavior explicitly. To alleviate this, we use proclets.

Modeling with Synchronous Proclets. A procllet is a Petri net that describes the behavior of a specific entity that can be distinguished through a unique identifier. Interactions between entities are described through synchronization channels between transitions of different procllets [6]. The synchronous procllet system in Fig. 4 describes the entire BHS of Fig. 2(a) by using three types of procllets.

1. The *process procllet* (red dotted border) is a Petri net describing the control-flow perspective of how bags, identified by variable pid may move through the system. It directly corresponds to the MFD of Fig. 2(b). It is transition-bordered and each occurrence of one of its initial transitions creates a new case identifier (a new value bound to variable pid) that was never seen before in the sense of ν -Petri nets [33], see [6] for details.
2. Each *resource procllet* (green dashed border) models a resource with cyclic behavior as its own entity identified by variable rid . For example, the *PassengerToSystemHandover* procllet (top left) identifies a concrete resource by token $rid = c1$; its life-cycle models that starting a task ($c1_s$) makes the resource *busy* and takes service time tsr_{c1} , after completing the task ($c1_c$) the resource has waiting time twr_{c1} before being *idle* again in the same way as Fig. 3. Which item the resource is busy with is recorded through variable pid in the pair (rid, pid) . In the classical CPN in Fig. 3, the pid is determined by the input and output transition of place *busy*. For the resource procllets in Fig. 4, pid is a free variable at $c1_s$ and $c1_c$ whose value is determined when synchronizing with the corresponding transition in the Process procllet (which we describe below). All other resource procllets follow the same pattern, though some resources such as *MergingUnit-m2* and *DivertingUnit-d1* may have two transitions to become *busy* or *idle*, respectively.
3. Each queue procllet (blue dash-dotted border) describes a FIFO queue as in Fig. 3 from the end transition of one task to the start transition of the subsequent task, e.g., from $c1_c$ (end of $c1$)

to $m2_s$ (start of $m2$). However, where Fig. 3 uses a distinct place $queue_{c1:m2}$ for the queue, a queue procelet maintains the queue state (the list) together with the queue identifier qid in place $queue$. Items entering the queue are remember by their pid . In the classical CPN in Fig. 3, the pid is determined by consuming from the input place of transition $c1_c$. For the queue procleets in Fig. 4, pid is a free variable at $c1_c$ whose value is determined when synchronizing with the corresponding transition in the Process procelet.

Where the model of Fig. 3 only uses identifiers for pid and distinguishes resource and queue through model structure, the resource and queue procleets explicitly model resource and queue identifiers through markings and variables. This will later allow us to relate event data over multiple identifiers to a procelet model and to decompose analysis problems along identifiers.

Transition Synchronization in Procleets. The procelet system synchronizes process, resources, and queues via *synchronous channels* between transitions. A transition linked to a synchronous channel may only occur when all linked transitions are enabled; when they occur, they occur in a single synchronized event. For example, transition $c1_s$ is always enabled in *Process*, generating a new bag id, e.g., $pid = 49$, but it may only occur together with $c1_s$ in *PassengerToSystemHandover*, i.e., when resource $c1$ is *idle*, thereby synchronizing the process case for bag $pid = 49$ with the resource with identifier $rid = c1$. By storing the pair $(rid, pid) = (c1, 49)$ on place *busy*, resource $c1$ is now *correlated* to case 49. Transition $c1_c$ of the process procelet can now only occur when synchronizing with $c1_c$ of the resource procelet, and thus only for $pid = 49$ and $rid = c1$. Moreover, both $c1_c$ transitions can only occur when synchronizing with the $c1_c$ transition of the queue procelet for $qid = c1 : m2$, thereby completing the work of $c1$ on $pid = 49$ and putting $pid = 49$ into the queue. Note that using CPN expressions (as used in queue and resource procleets) eliminates the need for dedicated correlation expressions used for the basic procelet model introduced in [6].

In the example, each resource is statically linked to one process step, but the model also allows for one resource to participate in multiple different process steps, and multiple resources to be required for one process step. In the following, we call a procelet system that defines procleets for processes, queues, and resource that are linked via synchronous channels as described above, a *PQR system*.

Procleets Describe Synchronizing Entity Traces. We now highlight how the partial-order semantics of synchronous procleets [6] preserves the identities of process, resources, and queues as “entity traces”. Figure 5(b) shows a partially-ordered run of the PQR system of Fig. 4 for two bags $id = 50$ and $id = 51$. The run in Fig. 5(b) can be understood as a synchronization of multiple runs or traces of the process, resource, and queue procleets, one for each case, resource, or queue involved as shown in Fig. 5(a).

Bag 50 gets inserted via input transition $c3_c$ (event e_0^* in Fig. 5(b)). This event is a *synchronization* of events e_0 ($c3_c$ occurs for bag 50 in the *Process* procelet) and e_0' ($c3_c$ occurs for the $c3:m3$ queue) in Fig. 5(a). The minimal waiting time twq_{c3m3} must pass before bag 50 reaches the end of the queue and process step $m3$ can start. The process step $m3$ merges bag 50 from the check-in conveyor $c3$ onto the main linear conveyor and may only start via transition $m3_s$ when *MergingUnit-m3* is *idle*. As this is the case, bag 50 leaves the queue ($e1''$ in $c3:m3$), $m3$ starts merging ($e1'$ in $m3$), the bag starts the merging step (event $e1$ in *Process*), resulting in the synchronized event $e1^*$ in Fig. 5(b).

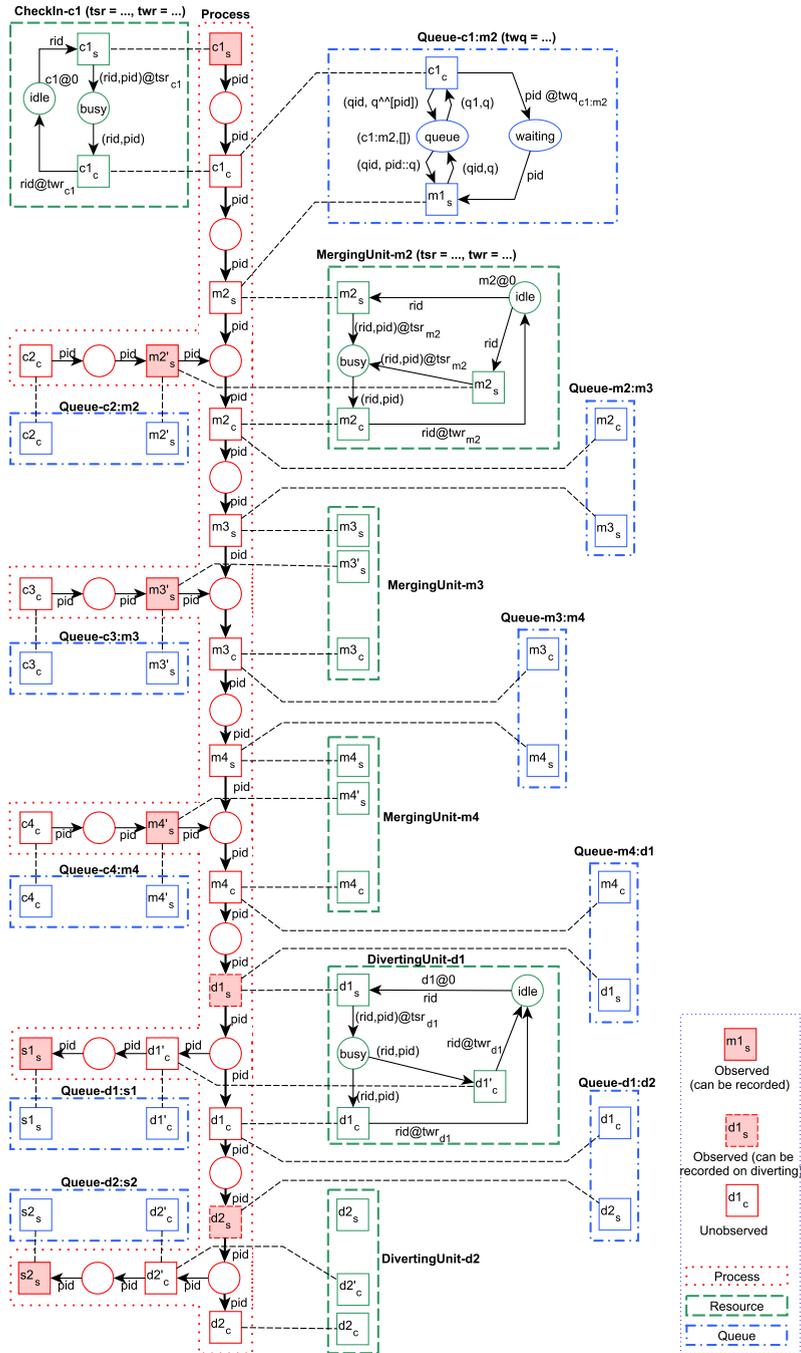


Figure 4. The synchronous procket model of the system shown in Fig. 2(a) consists of three types of prockets: *Process* for modeling a system layout and process control flow (red, dotted), *Resource* for modeling equipment performing tasks (green, dashed), and *Queue* for modeling conveyors transporting bags in the FIFO order (blue dash-dotted). Only filled transitions can be observed in an event log.

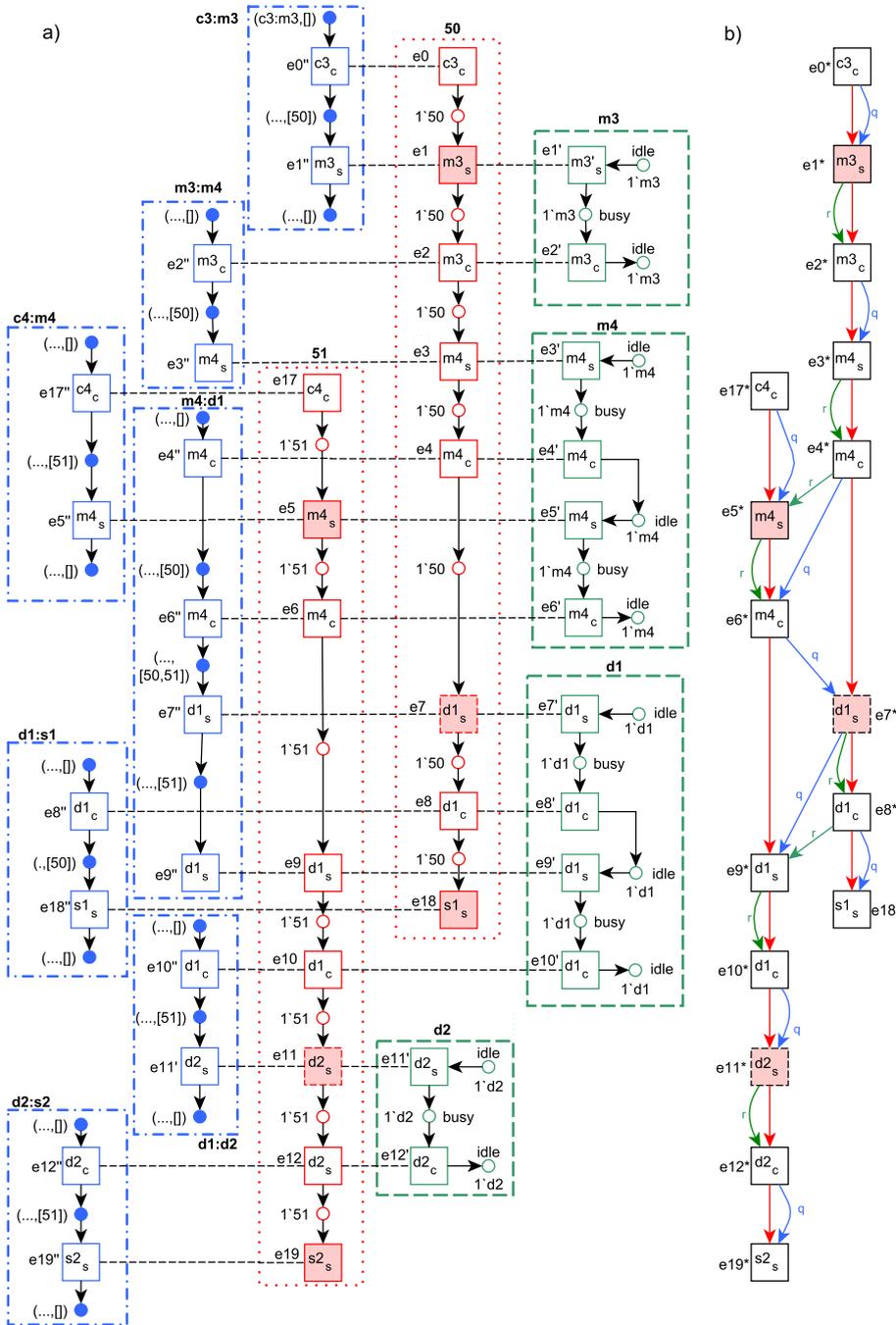


Figure 5. Synchronization of multiple sub-runs of the synchronous proclat system in Fig. 4 over shared resources and queues (a), and a global partial order obtained by the union of partial orders of each sub-run (b) for synchronized events, shown by red arrows, green arrows (with marker r) and blue arrows (with marker q) for partial orders \prec_{pid} , \prec_{rid} and \prec_{qid} respectively.

By e'_1 , resource $m3$ switches from *idle* to *busy* and takes time tsr_{m3} before it can complete the merge step with $m3_c$ (event $e2'$) on bag 50 (event $e2$); this merge step also inserts bag 50 into queue $m3:m4$ ($e2''$) resulting in synchronized event $e2^*$. Subsequently, bag 50 leaves queue $m3:m4$ ($e3^*$) is pushed by merge unit $m4$ into queue $m4:d1$ ($e4^*$).

Concurrently, bag 51 is inserted via input transition $c4_c$ (event $e17^*$), moves via queue $c4:m4$ also to merge unit $m4$ to enter queue $m4:d1$, i.e., both bags 50 and 51 now compete for merge unit $m4$ and the order of entering $m4:d1$. In the run in Fig. 5, $m4$ executes $m4_s$ and $m4_c$ for bag 51 ($e5^*$ and $e6^*$) after completing this step for bag 50 ($e3^*$ and $e4^*$). Thus, 51 enters the queue ($e6^*$) after 50 entered the queue ($e5^*$) but before 50 leaves the queue $e7^*$. Consequently, divert unit $d1$ first serves 50 ($e7^*$ and $e8^*$) to reach scanner $s1$ ($e18^*$) before serving 51 ($e9^*$ and $e10^*$) to reach scanner $s2$ ($e19^*$).

Fig. 5(b) shows how the process tokens of bag 50 and 51 synchronized with the resources and queue tokens along the run, forming sequences or *traces* of events where each of these tokens was involved. For example, bag 50 followed the trace $e0^*, e1^*, \dots, e8^*, e18^*$ and queue $m4:d1$ followed trace $e4^*, e6^*, e7^*, e9^*$ thereby synchronizing with both bag 50 and bag 51.

3.2. Information loss because of incomplete logging

Although event data on objects that are tracked can be used for various kinds of data analysis [4, 5], in practice sensors are placed only where it is absolutely necessary for correct operation of the system, e.g., for merge and divert operations, without considering data analysis needs. Applied to our example, only the transitions that are shaded in Fig. 4 would be logged, i.e., $c1_s, m2'_s, m3'_s, m4'_s, d1_s, d2_s, s1_s, s2_s$ would be logged from the *control-flow* perspective only. The run of Fig. 5 would result in a “typical” but highly incomplete event log as shown in Fig. 2(c).

According to this incomplete log, bag 50 silently passes $m4$ and is tracked again only at $d1$ ($e7$) and finally at $s1$ ($e18$) whereas 51 silently passes $d1$ (as it moves further on the main conveyor) and is tracked again only at $d2$ ($e11$). Based on this incomplete information the bags 50 and 51 may have traversed $m4:d1$ in different orders and at different speeds resulting also in different loads as illustrated in Fig. 1. As a result, in case of congestion, we cannot determine the ordering of cases [3], cannot compute the exact load on each conveyor part for (predictive) process monitoring [5, 10]. The longer an unobserved path (e.g., $c1 \rightarrow d2$), the higher the uncertainty about the actual behavior and the less accurate performance analysis outcome.

Although minimal (or even average) service and waiting times on conveyor belts and resource are known, we need to determine the *order* of all missing events and the possible *intervals of their timestamps* to reconstruct for how long resources were occupied by particular cases and in which order cases were handled, e.g., did 50 precede 51 on $m4:d1$ or vice versa?

The objective of this paper is to reconstruct from a subset of events logged from the control-flow perspective only the remaining events (including time information), so that the time order is consistent with a partially ordered run of the entire system, including resource and queue proclats. For example, from the recorded events of the event log in Fig. 2(c) we reconstruct the remaining events (Fig. 5(a)) with time information so that the resulting order (by time) is consistent with the partially ordered run in Fig. 5(b).

4. Modeling system-level runs from event data

Having introduced the problem in an informal way in Sect. 3, we now turn to formalizing it. We first discuss a behavioral model that describes, both, the behavior of all process executions in the system, and how these process executions interact via shared resource. To this end, we develop the notion *system-level runs* and how they canonically emerge from classical event logs that also log the shared resources involved in the process execution.

We first recall classical event logs in Sect. 4.1 and their traces. We then formalize *multi-entity event logs* in Sect. 4.2 where an event can be related to multiple different entities. This allows us to observe behavior along a specific shared resources in the same way as we observe behavior along a case identifiers.

To later be able to reason about behavior along multiple entities we then introduce two different but equivalent views on a multi-entity event log that differ in how they explicate behavior over multiple entities.

- Each multi-entity event log induces a family of classical event logs (one per entity type) that synchronize on shared events; we introduce this view in Sect. 4.3.
- Each multi-entity event log also induces a strict partial order over the events, where events are ordered over time along the same entity. We call this view a *system-level run* and introduce it in Sect. 4.4.

We show in Sect. 4.5 that all views contain the same information allowing us to switch perspectives on the behavior. We will use the different perspectives when formally stating the problem in Sect. 5 and solving the problem in Sect. 6.

4.1. Classical event logs

We first provide a definition of a classical event log, which we call *single entity event log*. We later generalize this definition to a *multi-entity event log*. With this aim of generalization in mind, we define a single entity event log just as a set of events with attributes. The cases and traces of an event log will then be derived from event attributes through canonical functions we provide afterwards.

From the usual event attributes of activity, time, and case identifier, only the activity name attribute *act* is mandatory. The *time* attribute is optional as we later want to study situations of incomplete logging. Also the case identifier attribute is optional for the same reason. When we later move to a multi-entity setting the term “case” is no longer adequate. We therefore call the case identifier attribute an *entity type* attribute *et*, referring to the type of entity on which the events are recorded (e.g., bags in baggage handling system)

Definition 4.1. (Single entity event log)

A *single entity event log* $L = (E, AN, et, \#)$ is a set E of events, a non-empty set AN of attribute names with $time, act \in AN$ and a designated *entity type attribute* $et \in AN$. The partial function $\# : E \times AN \rightarrow Val$ assigns events $e \in E$ and attribute names $a \in AN$ a value $\#_a(e) = v$, so that the activity name $\#_{act}(e)$ is defined for each event $e \in E$.

We write $\#_a(e) = \perp$ if event e has no value defined for attribute a . We call the value $\#_{et}(e) = id$ the entity identifier of e (for entity type et). Note that we do not require all events to be correlated to the designated entity type, i.e., $\#_{et}(e)$ can be undefined. Such events will later simply not be part of a case and trace. Events without time stamp $\#_{time}(e) = \perp$ are unordered to all other events. To distinguish event logs where all events are correlated to an entity and are ordered by time stamps, we introduce the following definitions.

Definition 4.2. (time-incomplete, time-monotone event log)

We call a single entity-event log $L = (E, AN, et, \#)$ *time-complete* iff for each $e \in E$ holds $\#_{time}(e) \neq \perp$ and $\#_{et}(e) \neq \perp$, i.e., each event has activity, time, and entity type. Otherwise L is called *time-incomplete*. We call a complete log L *time-monotone* iff for any two $e, e' \in E$ holds if $\#_{et}(e) = \#_{et}(e')$ then $\#_{time}(e) \neq \#_{time}(e')$.

Table 1. Event log with multiple entity types pid, rid, qid

event id	pid	activity	time	rid	qid
e_0	50	$c3_c$	01.01.20 9:00:15	\perp	c3:m3
e_1	50	$m3_s$	01.01.20 9:00:30	m3	c3:m3
e_2	50	$m3_c$	01.01.20 9:00:40	m3	m3:m4
e_3	50	$m4_s$	01.01.20 9:00:45	m4	m3:m4
e_4	50	$m4_c$	01.01.20 9:00:50	m4	m4:d1
e_7	50	$d1_s$	01.01.20 9:01:05	d1	m4:d1
e_8	50	$d1_c$	01.01.20 9:01:10	d1	d1:s1
e_{18}	50	$s1_s$	01.01.20 9:01:15	\perp	d1:s1
e_{17}	51	$c4_c$	01.01.20 9:00:35	\perp	c4:m4
e_5	51	$m4_s$	01.01.20 9:00:55	m4	c4:m4
e_6	51	$m4_c$	01.01.20 9:01:00	m4	m4:d1
e_9	51	$d1_s$	01.01.20 9:01:15	d1	m4:d1
e_{10}	51	$d1_c$	01.01.20 9:01:20	d1	d1:d2
e_{11}	51	$d2_s$	01.01.20 9:01:25	d2	d1:d2
e_{12}	51	$d2_c$	01.01.20 9:01:30	d2	d2:s2
e_{19}	51	$s2_s$	01.01.20 9:01:35	\perp	d2:s2

Table 1 shows a single entity event log for entity type *pid*. The log is time-complete and monotone: no two events for the same entity type carry the same time-stamp.

All events with the same value for *et* are correlated to the same entity or *case*. A *trace* is the sequence of all events in a case ordered by time (and events without time-stamp can be placed anywhere in the sequence).

Definition 4.3. (Case, trace, sequential event log)

Let $L = (E, AN, et, \#)$ be an event log with entity type attribute *et*.

The set of *cases* in L wrt. *et* is $et(L) = \{\#_{et}(e) \mid e \in E\}$, i.e., all entity (or case) identifier values in L .

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

A *trace* of case id is a sequence $\langle e_1, \dots, e_n \rangle$ of all events correlated to id that preserves time, i.e., $corr(L, et = id) = \{e_1, \dots, e_n\}$ and for all $1 \leq i < j \leq n$ hold if $\#_{time}(e_i) \neq \perp$ and $\#_{time}(e_j) \neq \perp$ then $\#_{time}(e_i) \leq \#_{time}(e_j)$. If two events have the same timestamp, a case id has more than one trace; we write $\sigma(L, et = id)$ for the set of traces of case id .

A *sequential event log* of L is a set $\sigma(L, et)$ which contains for each $id \in et(L)$ exactly one trace $\sigma \in \sigma(L, et = id)$.

For a time-incomplete event log L , the notion of trace and sequential event log are non-deterministic, i.e., an event without time stamp can be placed at an arbitrary position in the trace, allowing for multiple different traces for the same case. Only in a monotone event log L , each case id has a unique trace $\{\sigma_{et}^{id}\} = \sigma(L, et = id)$ and the log $\sigma(L)$ is uniquely defined. We then write $\sigma_{et}^{id} = \sigma(L, et = id)$.

Table 1 shows a time-complete, monotone single entity event log for entity type pid defining cases $\{50, 51\}$ and traces $\sigma(L, pid, 50) = \sigma_{pid}^{50} = \langle e_0, e_1, \dots, e_8, e_{18} \rangle$ and $\sigma_{pid}^{51} = \langle e_{17}, e_5, e_6, e_9, \dots, e_{12}, e_{19} \rangle$. This classical interpretation of the events in Table 1 describes how bags 50 and 51 travel through the baggage handling system of Fig. 2(a) from check-in $c3_c$ and $c4_c$ to scanners $s1_s$ and $s2_s$.

4.2. Event logs over multiple entities

In the classical single-entity event logs of Sect. 4.1, attributes rid and qid of Table 1 are considered so-called *event attributes* [9] which describe the event further, i.e., event e_6 of bag 51 at $m4_c$ was performed by resource $\#_{rid}(e_6) = m4$ (merge-unit 4) as the bag entered the conveyor belt $\#_{qid}(e_6) = m4 : d1$ from merge-unit 4 to divert-unit 1.

However, attributes rid and qid do refer to system entities in their own right: the machines that perform the various activities on the bags, and the conveyor belts that move bags between activities and machines. These machines and conveyor belts exist beyond individual cases and occur also in other cases, e.g., $\#_{rid}(e_3) = m4$, $\#_{rid}(e_4) = m4 : d1$ for bag 50. To study how these shared resources (e.g., machines, conveyor belts) relate and order bags over time, we introduce the notion of a *multi-entity event log* which designates multiple entity type attributes.

Definition 4.4. (Multi-entity event log)

An *event log with multiple entity types* $L = (E, AN, ET, \#)$ is a set E of events, a non-empty set AN of attribute names with $act \in AN$ and a subset $\emptyset \neq ET \subset AN$ is designated as *entity types*. Partial function $\# : E \times AN \rightsquigarrow Val$ assigns events $e \in E$ and attribute names $a \in AN$ a value $\#_a(e) = v$, so that the activity name $\#_{act}(e)$ is defined for each event $e \in E$.

Table 1 shows a monotone event log with multiple entity types $ET = \{pid, rid, qid\}$. In contrast to a single-entity log (Def. 4.1), an event in a multi-entity log (Def. 4.4) may carry more than one entity type $\#_{et}(e) \neq \perp$, $et \in ET$, e.g., $\#_{pid}(e_0) = 50$, $\#_{qid}(e_0) = c3 : m3$. As for Def. 4.1, an entity type may be undefined or one or multiple events, e.g., $\#_{rid}(e_0) = \perp$.

Note that a multi-entity event log $L = (E, AN, ET, \#)$ with a singleton set of identifiers $ET = \{et\}$ coincides with a classical event log (Def. 4.1). The notions of time-incomplete, complete, and monotone event log lift to multi-entity event logs by applying them on all entity types in ET .

We next introduce two views that materialize the behavioral information in a multi-entity event log: as sets of sequential event logs and as a partial order.

4.3. Sequential view on event logs over multiple entities

We use sequential traces (Def. 4.3) to describe the behavior stored in a single-entity event log L in an explicit form. Each trace in $\sigma(L, id)$ describes a possible sequences of activity executions over time for entity $id \in et(L)$; no two traces $\sigma(L, endid, id_1)$ and $\sigma(L, endid, id_2)$ share an event.

We now discuss how to materialize such sequential information from a multi-entity event log L for all entity types ET . First, we canonically derive a set of sequential event logs of L , one per entity type $et \in ET$. In Sect. 4.4 we discuss an alternative view based on partial orders.

Note that the functions $et(L)$, $corr(L, et, id)$, $\sigma(L, et, id)$ of Def. 4.3 are well-defined over multi-entity event logs.

Definition 4.5. (Sequential views on multi-entity event log)

Let $L = (E, AN, ET, \#)$ be a multi-entity event log.

A *sequential event log* of L for entity type $et \in ET$ is a set $\sigma(L, et)$ containing exactly one trace $\sigma \in \sigma(L, et, id)$ for each case $id \in et(L)$ of et (see Def. 4.3).

A *sequential view* on L is a family $\langle \sigma(L, et) \rangle_{et \in ET}$ of sequential event logs – one per entity type in L .

As for sequential event logs, if L is monotone, then the sequential event log $\sigma(L, et)$ of an entity is unique, and the sequential view on L is unique.

The sequential view on the monotone multi-entity event log in Tab. 1 has three sequential event logs $\sigma(L, pid)$, $\sigma(L, qid)$, $\sigma(L, rid)$. Thereby the *pid*-log $\sigma(L, pid)$ is the same as for the single-entity event log. It describes the behavior along the classical case identifier, i.e., one trace per bag in the system.

Log $\sigma(L, rid)$ has cases $m3, m4, d1, d2$ and, among others, traces $\sigma(L, rid, m4) = \sigma_{rid}^{m4} = \langle e_3, e_4, e_5, e_6 \rangle$ and $\sigma_{rid}^{d1} = \langle e_7, e_8, e_9, e_{10} \rangle$. These traces describe the order in which each machines was used. Note that $e_3, e_4, e_7, e_8 \in corr(L, pid, 50)$ while $e_5, e_6, e_9, e_{10} \in corr(L, pid, 51)$. That is, traces σ_{rid}^{m4} and σ_{rid}^{d1} for rid contain events from different bags, i.e., the *rid*-traces go “across” multiple different *pid*-traces.

Log $\sigma(L, qid)$ has cases $c3 : m3, m3 : m4, c4 : m4, m4 : d1, d1 : s1, d1 : d2, d2 : s2$ and, among others, trace $\sigma_{qid}^{m4:d1} = \langle e_4, e_6, e_7, e_9 \rangle$ while $e_4, e_7 \in corr(L, pid, 50)$ and $e_6, e_9 \in corr(L, pid, 51)$. These traces describe the order in which different *pid*-cases entered and left the queues.

Note that per sequential event log, each event occurs in only one trace, but the same event can be part of multiple different event logs (for different entity types), e.g., e_4 occurs in σ_{pid}^{50} , σ_{rid}^{m4} , and $\sigma_{qid}^{m4:d1}$. In this way, the *rid*- and *qid*-traces describe how different *pid*-traces are synchronized via shared machines (*rid*) and conveyor belts (*qid*). However, that synchronization of multiple traces is implicit in the sequential view. We therefore propose a partially-ordered view on a multi-entity event log next.

4.4. Partially ordered view on event logs over multiple entities

In Sect. 4.3, we used $\#_{et}(\cdot)$ to derive sequences of events related to the same entity ordered by $\#_{time}(\cdot)$. We next encode the same information in an ordering relation over events, which is a strict partial order due to the monotonicity of the $\#_{time}(\cdot)$ values. We thereby start by first ordering events $e_1 < e_2$ only if they are related to the same entity. The transitive closure then naturally extends this ordering across multiple different entities.

Definition 4.6. (Partial-order view, system-level run)

Let $L = (E, AN, ET, \#)$ be a monotone multi-entity event log.

Let $et \in ET$ and $id \in et(L)$. Event $e_1 \in E$ precedes event $e_2 \in E$ in entity id of type et , written $e_1 <_{et}^{id} e_2$ iff

1. $\perp \neq \#_{time}(e_1) < \#_{time}(e_2) \neq \perp$ (the time stamp of e_1 is before the time stamp of e_2), and
2. $\#_{et}(e_1) = \#_{et}(e_2) = id$ (both events are related to the same entity id).

e_1 directly precedes e_2 in entity id of type et , written $e_1 <_{et}^{id} e_2$, iff there exists no $e' \in E$ with $e_1 <_{et}^{id} e' <_{et}^{id} e_2$.

This ordering lifts to entity types and entire L :

- e_1 directly precedes e_2 in entity type et , written $e_1 <_{et} e_2$, iff $e_1 <_{et}^{id} e_2$ for some $id \in et(L)$; and
- e_1 directly precedes e_2 , written $e_1 < e_2$, iff $e_1 <_{et} e_2$ for some $et \in ET$.
- The transitive closures $(<_{et})^+ = <_{et}$ and $(<)^+ = <$ define (indirectly) precedes per entity type and for all events in L , respectively.

The partial-order view on L or *system-level run* of L (induced by τ) is $\pi = (E, <, AN, ET, \#)$.

Figure 5(b) visualizes the directly precedes relations $<_{pid}$, $<_{rid}$, $<_{qid}$ induced by $\#_{time}$ for the multi-entity event log in Tab. 1. This behavioral model shows that events of different process cases ($pid = 50$ and $pid = 51$) are independent under the classical control-flow perspective $<_{pid}$, e.g., $e4 \not<_{pid} e5 \not<_{pid} e7$ (see Def. 4.6), but mutually depend on each other under $<_{rid}$ and $<_{qid}$, e.g., $e4 <_{rid} e5 <_{rid} e6$ and $e6 <_{qid} e7$.

Note that events without defined time stamp are unordered to all other events, i.e., they can occur at any time. We will exploit this when inferring missing time stamps.

The order $<$ is indeed a strict partial order.

Lemma 4.7. Let $L = (E, AN, ET, \#)$ be a monotone multi-entity event log. Let $\pi = (E, <, AN, ET, \#)$ be the system-level run of L . Then $(E, <)$ is a strict partial order.

Proof:

We have to show that $<$ is transitive and irreflexive. $< = (<)^+$ is transitive by construction in Def. 4.6. Regarding irreflexivity: $e_1 < e_2$ holds only if $\#_{\tau}(e_1) < \#_{\tau}(e_2)$. As L is monotone, either $\#_{\tau}(e_1) < \#_{\tau}(e_2)$ or $\#_{\tau}(e_2) < \#_{\tau}(e_1)$ holds (Def. 4.2) but not both, hence $<$ is irreflexive. \square

4.5. Relation between sequential and partially-ordered view

To better define and solve the problem, we now establish a more explicit relation between the system-level run π of L and the traces in the sequential view of L .

Given a system-level run $\pi = (E, <, AN, ET, \#)$ we write π_{et} for the projection of π onto entity type $et \in ET$ where $\pi_{et} = (E_{et}, <_{et}, AN, \{et\}, \#)$ contains only the events $E|_{et} = \{e \in E \mid \#_{et}(e) \neq \perp\}$ related to et . Relation $<_{et}$ is already well-defined wrt. $E|_{et}$. We call π_{et} the entity-type level run of L for entity type et .

Correspondingly, given an identifier $id \in et(L)$, the projection $\pi_{et}^{id} = (E_{et}^{id}, <_{et}^{id}, AN, \{et\}, \#)$ contains only the events $E|_{et}^{id} = corr(L, et = id)$ of id . We call π_{et}^{id} the entity-level run of L of entity id of type et .

From the system-level run in Figure 5(b) we can obtain the entity-level runs π_{pid}^{50} and π_{pid}^{51} from the perspective of the process, π_{rid}^{m3} , π_{rid}^{m4} , π_{rid}^{d1} , π_{rid}^{d2} from the perspective of the resources, and $\pi_{qid}^{c3:m3}$, $\pi_{qid}^{m3:m4}$, $\pi_{qid}^{c4:m4}$, $\pi_{qid}^{m4:d1}$, $\pi_{qid}^{d1:d2}$, $\pi_{qid}^{d1:s1}$, $\pi_{qid}^{d2:s2}$ from the perspective of the conveyor belts (or queues).

Each entity-level run π_{et}^{id} corresponds to a sequential trace σ_{et}^{id} in the sequential view of L because either view derives the direct precedence/succession of events from the same principles.

Lemma 4.8. Let $L = (E, AN, ET, \#)$ be a monotone multi-entity event log. Let $\pi = (E, <, AN, ET, \#)$ be the system-level run of L .

For all $e_1, e_2 \in E$ holds: $e_1 < e_2$ iff there exists $et \in ET$ and $id \in et(L)$ so that $\langle \dots, e_1, e_2, \dots \rangle = \sigma(L, et = id)$ is a trace in the sequential view $\langle \sigma(L, et) \rangle_{et \in ET}$ of L .

Proof:

If $e_1 < e_2$ then by Def. 4.6, $e_1 <_{et}^{id} e_2$ for some $et \in ET$ and $id \in et(L)$. Thus, $\#_{et}(e_1) = \#_{et}(e_2)$ and $\#_{time}(e_1) < \#_{time}(e_2)$ (by Def. 4.6 and L being monotone). By Def. 4.3, $e_1, e_2 \in corr(L, et = id)$ (correlated into the same case id for et). Further, because there is no $e' \in corr(L, et = id)$ with $\#_{time}(e_1) < \#_{time}(e') < \#_{time}(e_2)$ (definition of $<$ in Def. 4.6), e_1 and e_2 are ordered next to each other in the sequential trace $\langle \dots, e_1, e_2, \dots \rangle = \sigma(L, et = id)$. The converse holds by the same arguments. \square

Corollary 4.9. Let $L = (E, AN, ET, \#)$ be a monotone multi-entity event log. Let $\pi = (E, <, AN, ET, \#)$ be the system-level run of L . For any π_{et}^{id} for $et \in ET, id \in et(L)$ holds $e_1 <_{et}^{id} e_2$ iff $\langle \dots, e_1, e_2, \dots \rangle = \sigma(L, et = id)$ and $e_i <_{et}^{id} e_j$ iff $\langle \dots, e_i, \dots, e_j, \dots \rangle = \sigma(L, et = id)$.

The above relation may not seem profound and be more a technical exercise. However, we benefit 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 $<_{pid}, <_{rid}, <_{qid}$ visualized in Figure 5(b) directly define the sequences of events we find in $\sigma(L, pid)$, $\sigma(L, rid)$, and $\sigma(L, qid)$.

5. Modeling multi-entity behavior with queueing and time

We introduced system-level runs and multi-entity event logs in Sect. 4. We now want to formally state the problem of inferring missing time-stamps from events logs which only recorded partial information about the system. Thereby the gap between “partial” and “complete” information depends on domain knowledge. We therefore first introduce a model for describing such domain knowledge about system-level behavior and state the formal problem afterwards.

Our model uses synchronous proplets [6] to describe a system as a composition of multiple smaller components (each called a proplet) that synchronize dynamically on transition occurrences. Which transitions synchronize is specified in channels. The original definition [6] is based on Petri nets with identifiers. To model queueing and time, we extend the synchronous proplet model with concepts of coloured Petri nets (CPN).

We first recall some basic syntax of colored Petri nets in Sect. 5.1 and then formulate a *replay semantics* to replay an event log over a CPN in Sect. 5.2. This replay semantics allows us to define conformance checking problems between a CPN and a multi-entity event log.

We then lift this definition of CPN replay semantics to CPN proplet systems where multiple proplets (each defined by a CPN) synchronize on joint transition firings in Sect. 5.3; the syntax defined there extend the basic synchronous proplet model [6] with concepts for data and time.

We then consider a specific class of CPN proplet systems which describe a single process with shared resources and queues. We call such a proplet system a *PQR system*. We introduce PQR systems in Sect. 5.4 and then formally state our research problem of repairing incomplete event logs in Sect. 5.5.

5.1. Background on coloured Petri nets

We here only recall the CPN notation and semantic concepts also needed in the remainder of this paper and do not introduce the entire formal model CPN; refer to [7] for an introduction and further details.

A *labeled coloured Petri net* (CPN) $N = (P, T, F, \Sigma, \ell, Var, Types, colSet, m_0, arcExp, arcTime)$ defines

- a *skeleton Petri net* (P, T, F) of places P , transitions T , and arcs F as usual; we write $\bullet t$ and $t\bullet$ for the pre- and post-places of transition t and $\bullet p$ and $p\bullet$ for the pre- and post-transitions of place p ;
- a *labelling function* $\ell : T \rightarrow \Sigma$ assigning each transition a name $a \in \Sigma$ from an alphabet Σ ;
- a set of *variable names* Var and a set of data $Types$;
- *color sets* (i.e., data types) $colSet : P \cup Var \rightarrow Types$ specifying for each place and variable which values it can hold;
- an *initial marking* $m_0 : P \rightarrow 2^{Values \times \mathbb{R}}$ assigning to each place a multiset of value-time pairs $(v, time)$ so that $v \in colSet(p), p \in P$;
- *arc expressions* $arcExp : F \rightarrow Exp$ defining for each arc an expression over Var and various operators, specifying which values to consume/produce; and

- a *time annotation* $arcTime : F \rightarrow \mathbb{R}$ defining for an output arc (t, p) how much time $arcTime(t, p)$ has to pass until a produced token becomes available.

Figure 4 shows multiple labeled coloured CPNs:

In *Process* (shown in red) each arc is annotated with the variable pid and each place p has the colorset $colSet(p) = Pid$ describing identifiers for different bags (cases) of the process. Thus, bags are described by their identifiers and transitions consume and produce these identifiers, thereby moving the bag forward through the process. Note that some transitions have no pre-places, e.g., $c1_s$. The pid value that these transitions produce can be chosen freely; our replay semantics in Sect. 5.2 will determine the pid value based on an event log.

In *CheckIn-c1*, places *idle* and *busy* have colorset Rid . Place *idle* carries a token $(c1, 0)$ meaning value $c1$ is available from time 0 onwards, i.e., resource $c1$ is idle at this time. All arcs carry variable rid as arc expression. Arc $(c1_s, busy)$ carries time annotation $arcTime(c1_s, busy) = tsr_{c1} > 0$ which specifies a delay of tsr_{c1} time units. When check-in starts ($c1_s$ fires) the resource $c1$ moves from *idle* to *busy* and only becomes available after tsr_{c1} time units. Then check-in can complete and $c1$ moves from *busy* to *idle* and only becomes available after twr_{c1} time units.

In *Queue-c1:m2*, place *queue* has a colorset of a pair (qid, q) where $qid \in Qid$ is a queue identifier and $q \in Pid^*$ is a list of bag identifiers. The initial token on *queue* is $(c1 : m2, \langle \rangle)$ (i.e., the empty queue). Transition $c1_c$ places a new bag (pid) on the start of conveyor belt by adding it to the end of the current queue, transition $m1_s$ removes a bag from the end of conveyor belt by removing its head from the queue. When $c1_c$ fires, the current queue (qid, q) , $q = \langle pid_1, \dots, pid_n \rangle$ is consumed from place *queue* and a bag identifier pid is appended to q , producing $(qid, q \hat{\sim} \langle pid \rangle) = \langle pid_1, \dots, pid_n, pid \rangle$ onto place *queue*. At the same time, token pid is produced on p_3 with a delay of $twq_{c1:m2}$, i.e., pid only becomes available after $twq_{c1:m2}$ time units. This allows to model that conveyor belt movement takes time. When this delay for pid has passed and pid is at the head of the queue, i.e., token (qid, q') with $q' = pid :: \langle pid_1, \dots, pid_n \rangle = \langle pid, pid_1, \dots, pid_n \rangle$ is on place *queue*, then $m1_s$ can fire. If $m1_s$ fires it consumes (qid, q') from *queue* and pid from p_3 , and produces (qid, q) with $q = \langle pid_1, \dots, pid_n \rangle$ on *queue*, thereby removing pid from the queue.

5.2. An event log replay semantics for colored Petri nets

We briefly recall the semantic concepts of CPNs. Let $N = (P, T, F, \Sigma, \ell, Var, Types, colSet, m_0, arcExp, arcTime)$ be a CPN in the following.

A *timed marking* m assigns each place p a multiset $m(p)$ of timed tokens $(val, time')$ where val is the value on p and $time'$ is the time after which v can be consumed. A *state* $s = (m, time)$ of a N has a timed marking m and a time-stamp $time$. The time-stamp $time$ is the global system time reached. The initial state of N is $(m_0, 0)$.

A *binding* $\beta : Var \rightarrow Values$ binds each variable v to some value $\beta(var)$. An arc expression exp can be evaluated under a binding β by replacing each variable var in exp with $\beta(var)$ and computing the result, we denote the result by $exp[\beta]$.

A transition t is *enabled* in a state $s = (m, time)$ for binding β iff for each input arc $(p, t) \in F$ holds there exists $(val, time') \in m(p)$ so that $arcExp(p, t)[\beta] = val$ and $time' \leq time$, i.e., evaluating the arc expression $arcExp(p, t)$ yields a value val which is already available at the current time.

If t is enabled in s for binding β , then t can fire resulting in the *transition-step* $s = (m, time) \xrightarrow{t, \beta} (m', time) = s'$ where

1. for each pre-place p of t , remove from $m(p)$ a timed token $(arcExp(p, t)[\beta], time')$ with $time' \leq time$, resulting in an intermediate marking m'' , and
2. for each post-place p of t , add to $m''(p)$ the token $(arcExp(t, p), time + arcTime(t, p))$.

Further N can make a *time-step* $s = (m, time) \xrightarrow{delay} (m, time + delay) = s', delay \geq 0$. In the original CPN semantics [7] time has to advance at most until the next transition becomes enabled. We drop this requirement and allow arbitrary time-progress to facilitate replaying event logs with their own time-stamps.

We can now define the semantics of replaying a multi-entity event log $L = (E, AN, ET, \#)$ over a CPN. The idea is that the activity name $\#_{act}(e)$ of an event e specifies the label $\ell(t)$ of the transition t that shall be fired, $\#_{time}(e)$ specifies the global time when t fires. We treat the attributes $AN \setminus \{time, act\}$ as variables and the attribute-value function $\#_a(e) = v$ defines the binding $\beta(a) = v$ for which t shall fire. For this definition we will ignore the entity types ET as the definition is general to any CPN.

Definition 5.1. (CPN Replay Semantics)

Let $N = (P, T, F, \Sigma, \ell, Var, Types, colSet, m_0, arcExp, arcTime)$ be a CPN with initial state $s_0 = (m_0, 0)$. Let $L = (E, AN, ET, \#)$ be a time-complete multi-entity event log, i.e., each event has a time-stamp.

Let $\sigma = \langle e_1, \dots, e_n \rangle$ be a sequence of all events in E such that $\#_{time}(e_i) \leq \#_{time}(e_{i+1})$ for $1 \leq i < n$. Each event e_i defines the *binding* β_i with $\beta_i(a) = \#_a(e_i)$ for all $a \in AN \setminus \{time, act\}$.

An event sequence σ of L can be *replayed* on N iff for $1 \leq i \leq n$ exist delay $0 \leq d_i \in \mathbb{R}$ and transition $t_i \in T$ with $\ell(t) = \#_{act}(e_i)$ so that $s_{i-1} = (m_{i-1}, time_{i-1}) \xrightarrow{d_i} (m_{i-1}, \#_{time}(e_i)) \xrightarrow{t_i, \beta_i} (m_i, \#_{time}(e_i)) = s_i$ are time- and transition-steps in N that advance to $\#_{time}(e_i)$ and fire $\#_{act}(e_i)$.

By $t(e_i, \sigma)$ we denote for each event e_i the transition $t_i \in T$ which performed the transition-step described by e_i when replaying σ .

Note that this definition requires that the transition $t = t(e_i, \sigma)$ is enabled at time $\#_{time}(e_i)$ (or was already enabled earlier).

For example, the sequential trace $\sigma_{qid}^{c3:m3} = \langle e_0, e_1 \rangle$ of the event log in Table 1 is replayed on the CPN *Queue-c3:m3* (identical to CPN *Queue-c3:m3* in Fig. 4) as follows:

- the initial marking is $m_0(queue) = [((c3 : m3, \langle \rangle), 0)]$, $m_0(p3) = []$;
- e_0 yields steps $(m_0, 0) \xrightarrow{9:00:15} (m_0, 9:00:15) \xrightarrow{c3c, \beta_1} (m_1, 9:00:15)$ with binding $\beta_1(qid) = c3 : m3$, $\beta_2(pid) = 50$ and resulting marking $m_1(queue) = [((c3 : m3, \langle 50 \rangle), 9:00:15)]$, $m_1(p3) = [(50, 9:00:15 + twqc3:m3)]$;
- e_1 yields steps $(m_1, 9:00:15) \xrightarrow{0:00:15} (m_2, 9:00:30) \xrightarrow{m3s, \beta_2} (m_2, 9 : 00 : 30)$ with binding $\beta_2(qid) = c3 : m3$, $\beta_2(pid) = 50$ and resulting marking $m_2(queue) = [((c3 : m3, \langle \rangle), 9:00:30)]$ and $m_2(p3) = []$.

5.3. Synchronous procelet systems with data and time

We can now define our model of a synchronous procelet system with data and time.

Definition 5.2. (CPN Procelet)

A CPN procelet $Proc = (N, et)$ is a CPN $N = (P, T, F, \Sigma, \ell, Var, Types, colSet, m_0, arcExp, arcTime)$ so that

- $et \in Var$ is a designated *entity type variable* that can be bound to entity identifier values $colSet(et) = IDval$,
- for each transition t with a pre-place, there exists an arc (p, t) with $arcExp(p, t)$ has the form et or (et, exp) where exp is some expression, and
- for each transition t with a post-place, there exists an arc (t, p) with $arcExp(t, p)$ has the form et or (et, exp) where exp is some expression.

All CPNs in Fig. 4 are CPN-procleets, e.g., *Queue-c1:m2* has $et = qid$ and $c1_c$ and $m1_s$ both have incoming and outgoing arcs of the form (qid, exp) ; the arcs to/from p_3 have a different form.

This structure ensures that each transition in a CPN-procelet $Proc = (N, et)$ occurs in relation to a specific entity instance identified by variable et , e.g., a specific bag, resource, or queue. A *procelet system* synchronizes multiple procleets via channels.

Definition 5.3. (CPN Procelet System)

A CPN procelet system $S = (\{Proc_1, \dots, Proc_k\}, C)$ is a set $\{Proc_1, \dots, Proc_k\}$ of procleets with disjoint sets of transitions and places, and a set of *synchronization channels* $C \subseteq 2^T$ being sets of transitions so for each channel $\{t_1, \dots, t_r\}$ holds $\ell(t_i) = \ell(t_j)$.

In Fig. 4 shows a CPN procelet system where the channels are indicated by dashed edges, e.g., all transitions labeled $c1_c$ in *CheckIn-c1*, *Process*, and *Queue-c1:m2* form a channel.

The intuition is that transitions connected via a channel (t_i, t_j) can only fire together, i.e., procleets $Proc_i$ and $Proc_j$ must each be in a marking where t_i and t_j are enabled for the same binding (i.e., variables occurring in both t_i and t_j must be bound to the same values). While the original procelet semantics [6] is an operational semantics, we now provide a replay semantics over a CPN procelet system.

We replay a multi-entity event log $L = (E, AN, ET, \#)$ over a CPN procelet system $S = (\{Proc_1, \dots, Proc_k\}, C)$ by decomposing L into its sequential event logs $L_{et}, et \in ET$ and replaying each sequential event log over the corresponding procelet in S . As multiple procleets may use the same entity type et (e.g., *CheckIn-c1* and *MergeUnit-m2* both use rid), we need to specify which case in L_{et} belongs to which procelet $Proc_i$ in S .

Definition 5.4. (Replaying a Log over a CPN Procelet System)

Let $L = (E, AN, ET, \#)$ be a multi-entity event log (Def. 4.4) that is time-complete (Def. 4.2). Let $S = (\{Proc_1, \dots, Proc_k\}, C)$ be a procelet system.

Let $f : Val \rightarrow \{1, \dots, k\}$ be a mapping so that for each $et \in ET$ and each $id \in et(L)$, $f(id) = i$ maps to a procelet $Proc_i$ with $et_i = et$.

The entire log L can be *replayed* over S (for a given mapping f) iff the following conditions hold:

1. For each $et \in ET$ and sequential event log $\sigma(L, et)$ of L (Def. 4.5) holds: each trace $\sigma_{et}^i d \in \sigma(L, et)$ can be replayed on the CPN N_i of proctlet $Proc_i$ with $i = f(id)$.
2. for each event $e \in E$ and all traces $\sigma_{et_1}^{id_1}, \dots, \sigma_{et_k}^{id_k}$ that contain e , the set $c = \{t(e, \sigma_{et_1}^{id_1}), \dots, t(e, \sigma_{et_k}^{id_k})\}$ of transitions replayed in the different traces is either singleton $c = \{t\}$ or a channel $c \in C$ of S .

Figure 5 shows how the multi-entity event log L of Tab. 1 can be replayed over the proctlet system of Fig. 4. Each dashed rectangle in Figure 5(a) abstractly illustrates how one sequential trace in L is replayed over one of the proctlets in Fig. 4, the circles indicate parts of the markings reached after replaying each event, e.g., replaying e_0 in proctlet $Queue-c3:m3$ yields a token ($c3 : m3, [50]$) on place $queue$, etc. The dashed lines indicate how the channel constraints are satisfied by this replay. For instance, event e_4 is replayed by transition $m4_c$ in proctlet $Queue - c4 : m4$ (trace $\sigma_{qid}^{c4:m4}$), by $m4_c$ in $Process$ (trace σ_{pid}^{50}), and by $m4_c$ in $MergeUnit - m4$ (trace σ_{rid}^{m4}); all three $m4_c$ transitions form a channel in the proctlet system in Fig. 4.

5.4. PQR systems

In the following, we only study a specific sub-class of CPN proctlet systems which describes processes with shared resources and queues, which we call PQR-systems. Each PQR system is composed of one process proctlet and multiple resource and multiple queue proctlets in a specific way. Figure 4 shows an example of a PQR system. We define each proctlet type first and then the entire composition.

Intuitively, a process proctlet describes a sequential process where each process step has designated *start* and *complete* transitions, i.e., each step is non-atomic and start and complete are separately observable. Moreover, the process proctlet allows creating arbitrarily many fresh process instances through source transitions without pre-places; cases that complete are consumed by sink transitions without post-places. This is different from the concept of workflow nets [34] which model only the evolution of a single case and abstract from case creation and deletion.

Definition 5.5. (Process proctlet)

A *Process-proctlet* (or P-proctlet) is a CPN proctlet (N, pid) , $N = (P, T, F, \Sigma, \ell, Var, Types, colSet, m_0, arcExp, arcTime)$ where the following properties hold:

1. $P = P_{activity} \uplus P_{handover}$ (places either describe that an activity is being executed or that a case being handed over to the next activity);
2. $T = T_{start} \uplus T_{complete}$ (transitions either describe that an activity is being started or being completed)
3. N is a state-machine, i.e., $|\bullet t| \leq 1$ and $|t \bullet| \leq 1$ and all nodes are connected,
4. N is transition-bordered, i.e., $|\bullet p| \geq 1$ and $|p \bullet| \geq 1$ and the sets $T_{source} = \{t \in T \mid \bullet t = \emptyset\} \neq \emptyset$ and $T_{sink} = \{t \in T \mid t \bullet = \emptyset\} \neq \emptyset$ of *source* and *sink* transitions are non-empty.

5. Each activity place $p \in P_{activity}$ is only entered via start transitions and only left with complete transitions, i.e., $\bullet p \subseteq T_{start}$ and $p \bullet \subseteq T_{complete}$.
6. Each handover place $p \in P_{handover}$ is only entered via exactly one complete transitions (of the preceding activity) and left only via exactly one start transition (of the succeeding activity), i.e., $\bullet p = \{t\} \subseteq T_{complete}$ and $p \bullet = \{t\} \subseteq T_{start}$.
7. All arcs carry the entity identifier pid : $arcExp(x, y) = pid \in Var$ for all $(x, y) \in F$.
8. No place carries an initial token: $m_0(p) = []$ for all $p \in P$.

The proclat *Process* in Figure 4 is a P-proclat.

In this paper, a resource proclat defines the most basic life-cycle of a shared resource: the resource is initially *idle* (available to do work), then starts an activity making the resource *busy*. There is a *minimum service time* tsr the resource is busy before the task completes. After completing the task, the resource is *idle* again but requires a *minimum waiting time* twr before being able to work again. Figure 4 shows several resource proclats which we formally capture in the following definition.

Definition 5.6. (Resource proclat)

A *Resource-proclat* (or R-proclat) is a CPN proclat (N, rid) , $N = (P, T, F, \Sigma, \ell, Var, Types, colSet, m_0, arcExp, arcTime)$ for a resource with minimum service time tsr and minimum waiting time twr when the following properties hold:

1. $P = \{p_{idle}, p_{busy}\}$;
2. $T = T_{start} \uplus T_{complete}$ (transitions either describe that an activity is being started or being completed);
3. $p_{idle} \bullet = T_{start} = \bullet p_{busy}$ (resources go from idle to busy via start transitions);
4. $p_{busy} \bullet = T_{complete} = \bullet p_{idle}$ (resources go from busy to idle via start transitions);
5. $arcExp(x, y) = rid$ for all $(x, y) \in F$
6. $arcTime(t_{start}, p_{busy}) = tsr$ for all $t_{start} \in T_{start}$ and $arcTime(t_{complete}, p_{idle}) = twr$ for all $t_{complete} \in T_{complete}$
7. $m_0(p_{idle}) = [rid]$ and $m_0(p_{busy}) = []$ (the resource is idle initially)

An R-proclat has multiple start and complete transitions to mirror that an activity in a P-proclat has multiple start and complete transitions. This will simplify the composition of proclats later on. For example *Merging-Unit-m2* in Fig. 4 has two start and one complete transitions while *Diverting-Unit-d1* has one start and two complete transitions.

In this paper, a queue proclat defines the most basic operation of a queue: it ensures that items leave the queue in the order in which they entered the queue; moreover, we specify that traversing the queue requires a minimal waiting time twq .

Definition 5.7. (Queue proctet)

A *Queue-proctet* (or Q-proctet) is a CPN proctet (N, qid) , $N = (P, T, F, \Sigma, \ell, Var, Types, colSet, m_0, arcExp, arcTime)$ for a queue identified by entity identifier value Q with minimum waiting time twq when N is an instance of the CPN template shown in Fig. 6.

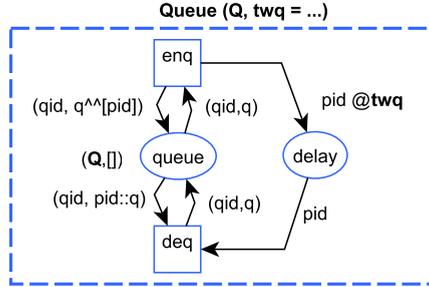


Figure 6. CPN template for Queue proctet

For example proctet *Queue-c1:m2* in Fig. 4 is identified by entity identifier value *c1:m2* and has minimum waiting time $twq_{c1:m2}$.

We can now formally define a PQR system as a CPN proctet system of one process proctet and multiple resource and queue proctets. A PQR system has specific synchronization constraints. Each activity in a process proctet (a place $p \in P_{activity}$ with corresponding start and complete transitions) synchronizes with one resource proctet which is responsible for executing this activity for any incoming case. Each handover between two activities in a process proctet (a place $p \in P_{handover}$) synchronizes with one queue moving cases from one activity to the next.

Definition 5.8. (PQR system)

A PQR system is a CPN proctet system $S = (\{Process_0, R_1, \dots, R_k, Q_{k+1}, \dots, Q_n\}, C)$ where the following properties hold:

1. $Process_0$ is a process proctet, R_1, \dots, R_k are resource proctets, Q_{k+1}, \dots, Q_n are queue proctets.
2. Each transition $t \in T_0 \cup \dots \cup T_n$ is in exactly one channel $c \in C$, denoted by $c(t)$.
3. For each activity place $p \in P_{activity}$ in $Process_0$ exists a resource proctet R_i so that (1) for each $t_{start,0} \in \bullet p$ (in $Process_0$) exists $t_{start,i} \in T_{start,i}$ (in R_i) with $c(t_{start,0}) = c(t_{start,i})$ and (2) for each $t_{complete,0} \in p^\bullet$ (in $Process_0$) exists $t_{complete,i} \in T_{complete,i}$ (in R_i) with $c(t_{complete,0}) = c(t_{complete,i})$.
4. For each handover place $p \in P_{handover}$ in $Process_0$ exists a queue proctet Q_i so that (1) transition $\{t_{complete,0}\} = \bullet p$ (in $Process_0$) synchronizes with $t_{enqueue,i} \in T_i$ (in Q_i) via $c(t_{complete,0}) = c(t_{enqueue,i})$ and (2) transition $\{t_{start,0}\} = p^\bullet$ (in $Process_0$) synchronizes with $t_{dequeue,i} \in T_i$ (in Q_i) via $c(t_{start,0}) = c(t_{dequeue,i})$.

The proctet system in Fig. 4 is a PQR system.

The above definition is rather declarative. To satisfy the above constraints there have to be enough R-proclets (one per activity place) of the correct shape (to match the start and complete transitions in the P-proclet), and enough Q-proclets (one per handover place). Moreover, the transition labels in all proclets have to match to form valid channels.

The definition enforces that in a PQR system each activity in a process is carried out by a shared resource of limited capacity (condition 3 in Def. 5.8). Thus when multiple case arrive at the same activity, only for one of them the activity can be started while the others have to wait. Further, when an activity for a case is completed, the case enters a queue and can only reach the next activity when all other cases before it have reached that activity (condition 4 in Def. 5.8).

Many real-life processes show more general use of shared resources and handover of cases than these very strict constraints. However, they are satisfied by material handling systems such as baggage handling systems. Generalizing the definition to other types of processes with shared resources is beyond the scope of this paper.

5.5. Restoring partial event logs of a PQR system

We can now formally state our research problem.

Let L be a multi-entity event log. And let S be a PQR system defining proclets for a process (with case identifier pid), multiple resources (with entity identifier rid) and queues (with entity identifier qid); see Def. 4.4 and Def. 5.8.

L is *correct and complete* log of S iff L can be *replayed* over the entire system S ; see Def. 5.4. The event log of Tab. 1 is a complete log of the PQR system in Fig. 4.

A correct and complete log L has at least entity identifiers pid , rid , and qid (as these are required by a PQR system). Further, each trace σ_{et}^{id} can be replayed on the corresponding proclet, i.e., each trace describes a complete execution of the proclet for instance id . Further, all traces $\sigma_{pid}^{id} \in \sigma(L, pid)$ of process entities (pid) are ordered relative to each other via the shared resources and queues as described in S .

In reality often only a subset of activities $B \subseteq A = \{\#_{act}(e) \mid e \in L\}$ and the control-flow identifier pid have been recorded in a log, making it *partial*.

Definition 5.9. (Partial Log, Observed Event, Corresponds)

A log $L' = (E', AN', \{pid\}, \#')$ is a *partial* (and correct) log of PQR system S if there exists a correct and complete log $L = (E, AN, ET, \#)$ of S such that

1. $E' \subseteq E$, $AN' \subseteq AN$, and $\#' = \#|_{E' \times AN'}$ is the restriction of $\#$ to E' and AN' ,
2. each $e \in E'$ has only case notion pid , i.e., $\#_{pid}(e) \neq \perp$ (and $\#_{rid}(e) = \#_{qid}(e) = \perp$),
3. $\#_{time}(e)$ is defined, and
4. for each complete process trace $\sigma(L, pid = id) = \langle e_1, \dots, e_n \rangle$ the partial trace $\sigma(L', pid = id) = \langle f_1, \dots, f_k \rangle$ records at least the first and last event $e_1 = f_1$ and $e_n = f_k$.

We call each event $e \in E'$ an *observed event*. We say that the complete log L *corresponds to* the partial log L' .

Table 2. Partial event log of the complete event log in Tab. 1, missing events and attributes shown in grey

event id	pid	activity	time	rid	qid
e_0	50	$c3_c$	01.01.20 9:00:15	\perp	c3:m3
e_1	50	$m3_s$	01.01.20 9:00:30	m3	c3:m3
e_2	50	$m3_c$	01.01.20 9:00:40	m3	m3:m4
e_3	50	$m4_s$	01.01.20 9:00:45	m4	m3:m4
e_4	50	$m4_c$	01.01.20 9:00:50	m4	m4:d1
e_7	50	$d1_s$	01.01.20 9:01:05	d1	m4:d1
e_8	50	$d1_c$	01.01.20 9:01:10	d1	d1:s1
e_{18}	50	$s1_s$	01.01.20 9:01:15	\perp	d1:s1
e_{17}	51	$c4_c$	01.01.20 9:00:35	\perp	c4:m4
e_5	51	$m4_s$	01.01.20 9:00:55	m4	c4:m4
e_6	51	$m4_c$	01.01.20 9:01:00	m4	m4:d1
e_9	51	$d1_s$	01.01.20 9:01:15	d1	m4:d1
e_{10}	51	$d1_c$	01.01.20 9:01:20	d1	d1:d2
e_{11}	51	$d2_s$	01.01.20 9:01:25	d2	d1:d2
e_{12}	51	$d2_c$	01.01.20 9:01:30	d2	d2:s2
e_{19}	51	$s2_s$	01.01.20 9:01:35	\perp	d2:s2

Thus, a partial log L' contains for each case pid at least one *partial trace* σ_{pid}^{id} recording the entry and exit of the case and preserving the order of observed events, i.e., it can be completed to fit the model. An MHS typically records a partial log as defined above. Tab. 2 shows a partial event log of the complete log of Tab. 1. Fig. 5(b) highlights the events that are recorded in the partial event log .

Note that a partial event log coincides with the definition of a classical single-entity event log (Def. 4.1). In a partial event log, events of different process cases are less ordered, e.g., observed events e_1 and e_5 in Fig. 5 are unordered wrt. any resource or queue whereas they are ordered in the corresponding complete event log.

Lemma 5.10. Let L' be a partial event log of a PQR system S . Let L be a complete event log of S that corresponds to L' . Let $\pi(L')$ and $\pi(L)$ be the system-level runs of L' and L , respectively. Then for each $e_1, e_2 \in E'$: $e_1 < e_2$ in L' implies $e_1 < e_2$ in L .

Proof:

For any $e_1 < e_2$ in L' holds $\#_{pid}(e_1) = \#_{pid}(e_2)$ and $\#_{time}(e_1) < \#_{time}(e_2)$. These properties also hold in L , thus $e_1 < e_2$ in L . \square

The converse does not hold. In the complete system-level run $\pi(L)$ in Fig. 5(b), $e_5 < e_7$ (due to $qid = m4 : d1$), whereas $e_5 \not< e_7$ in the system-level run of the partial log L' (where e_5 and e_7 are unrelated). In the following, we investigate how to infer missing events and infer missing time-stamps, and thereby reconstruct the missing ordering relations.

Formal problem statement Let S be a model of a PQR system defining life-cycles of process, resource, and queue proplets, which resources and queues synchronize on which process step, and for each resource the minimum service time tsr and waiting time twr and for each queue the minimum waiting time twq . Given S and a partial event log L_1 of S , we want to construct a complete log L_2 of S that corresponds to L_1 (and can be replayed on S) according to Def. 5.9.

Restoring the *exact* timestamp is generally infeasible and for most use cases also not required. Thus, our problem formulation does not require to reconstruct the exact time-stamps. Our CPN replay semantics allows to fire transitions after their first moment of enabling, however they have to fire “early enough” so that time constraints do not conflict with later observed events. Thus, we have to reconstruct *time-windows* providing minimal and maximal timestamps for each unobserved event, resulting in the following sub-problems:

- Infer unobserved events E_u for all process cases in L_1 and their relations to queues and resources (infer missing identifiers)
- Infer for each unobserved event $e \in E_u$ a time-window of earliest and latest occurrence of the event $\#_{tmin}(e), \#_{tmax}(e)$ so that setting $\#_{time}(e) = \#_{tmin}(e)$ or $\#_{time}(e) = \#_{tmax}(e)$ for $e \in E_u$ results in a complete log of S .

6. Inferring timestamps along entity traces

In Sect. 5.5, we presented the problem of restoring missing events and time-windows for their timestamps from a partial event log $L_1 = (E_1, AN_1, \{pid\}, \#^1)$ such that the resulting log is consistent with resource and queueing behavior specified in a PQR System S . In this section, we solve the problem for PQR systems with *acyclic* process proplets by casting it into a constraint satisfaction problem, that can be solved using Linear Programming (LP) [8]. In all subsequent arguments, we make extensive use of the fact that we can see any multi-entity event log L_1 equivalently as family of sequential event logs $\sigma(L_1, et)$ with traces σ_{et}^{id} and as the system-level run $\pi(L_1) = (E_1, <_1, AN_1, \{pid\}, \#^1)$ with strict partial order $(E_1, <_1)$.

In Sect. 6.1, we show how to infer unobserved events and resource and queue identifiers (from S) to construct an under-specified intermediate system-level run $\pi_2 = (E_2, <_2, AN_2, \{pid, rid, qid\}, \#^2)$ where all unobserved events $E_u = E_2 \setminus E_1$ have no timestamp but where $<_2$ already contains all ordering constraint that must hold in S .

In Sect. 6.2 we then refine π_2 into $\pi(L_3) = (E_2, <_3, AN_3, \{pid, rid, qid\}, \#^3)$ where $<_3$ is no longer explicitly constructed but completely inferred from time stamps that fit S . We determine minimal and maximal timestamps $\#_{tmin}^3(e)$ and $\#_{tmax}^3(e)$ for each unobserved event $e \in E_u$ (through a linear program) so that if we set $\#_{time}^3(e) = \#_{tmin}^3(e)$ or $\#_{time}^3(e) = \#_{tmax}^3(e)$, the induced partial order $<_3$ refines $<_2$, i.e., $<_2 \subseteq <_3$. By construction of $\#_{tmin}^3(e)$ and $\#_{tmax}^3(e)$, L_3 is a complete log of M and has L_1 as a partial log. We explain our approach using another (more compact running) example shown in Fig. 7(a) for two bags 53 and 54 processed in the system of Fig 4. The events in grey italic (i.e., f3, f5, f6, f14) are unobserved.

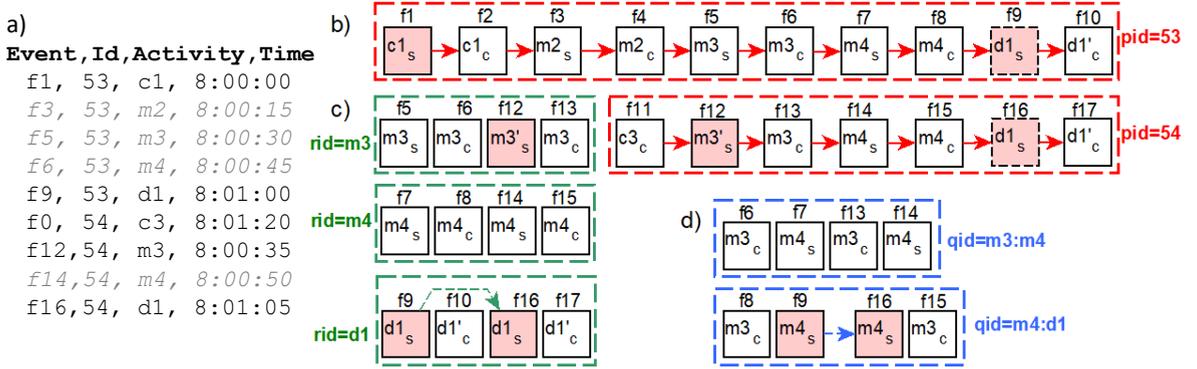


Figure 7. Another partial event log of the system in Fig. 4 for bags 53 and 54 (a), partially complete traces of the Process (b), Resource (c) and Queue (d) procllets, restored by oracles O_1, O_2 . Only observed events are ordered, e.g., $f_9 <_{rid}^{d1} f_{16}$, while the other events are isolated.

6.1. Infer potential complete runs from a partial run

We first infer from the partial event log L_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 $\pi_{2,pid}^{id}$ is complete (can be replayed on the process procllet in S). In a second step, we relate each unobserved event $e \in E_u = E_2 \setminus E_1$ to a corresponding resource and/or queue identifier which orders observed events wrt. $<_{rid}$ and $<_{qid}$. All unobserved events $e \in E_u$ lack a timestamp and hence are left unordered wrt. $<_{rid}$ and $<_{qid}$ in $\pi(L_2)$; we later refine $<_2$ in Sect 6.2.

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 $L_2 = \{\sigma_{pid}^{id} \mid id \in pid(L_1)\} = O_1(L_1, S)$ by completing each partial trace $\sigma(L_1, pid = id)$ of any process case $id \in pid(L_1)$ into a complete trace σ_{pid}^{id} that can be replayed on the process procllet of S . Let $E_2 = \{e \in \sigma_{pid}^{id} \mid \sigma_{pid}^{id} \in L_2\}$. The restored *unobserved* events $E_u = E_2 \setminus E_1$ only have attributes *act* and *pid* and events are *totally ordered* along *pid* in each trace σ_{pid}^{id} . O_1 can be implemented using well-known trace alignment [35] by aligning each sequential trace $\sigma(L_1, pid = id)$ on the skeleton net (P, T, F) of the P-procllet of S . For example, applying O_1 on the partial log of Fig. 7(a) results in the complete process traces of Fig. 7(b).

At this point, the events $e \in E_u$ have no time-stamp and the ordering of events is only available in the explicit sequences $\sigma_{pid}^{id} = \langle e_1, \dots, e_n \rangle$. Until we have determined $\#_{time}(e_i)$, the SPO $<_2$ has to be constructed explicitly from the ordering of events in the traces σ_{pid}^{id} , i.e., we define $<_2$ as $e_i < e_j$ iff there ex. a trace $\langle \dots, e_i, \dots, e_j, \dots \rangle = \sigma_{pid}^{id} \in L_2$ (see Cor. 4.9).

Moreover, as each trace σ_{pid}^{id} can be replayed on the process procllet, each event is either a start event (replays a start transition $t \in T_{start}$) or a complete event (replays a complete transition $t \in T_{complete}$, see Def. 5.5).

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 $e \in E_2$ if resource r is involved in the step $\#_{act}(e)$, then $\#_{rid}(e) = r$ and if queue q was involved, then $\#_{qid}(e) = q$.

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 $\#_{rid}(e) = r \neq \perp$, then $\#_{tsr}(e)$ and $\#_{twr}(e)$ hold the minimum service and waiting time of r , and if $\#_{qid}(e) = q \neq \perp$, then $\#_{twq}(e)$ hold the minimum waiting time of q .

For the concrete PQR systems considered in this paper, we set $\#_{rid}(e) = r$ based on the model S if r is the identifier of the resource procllet that synchronizes with transition $t = \#_{act}(e)$ via a channel $c(t)$ (there is at most one). Attributes $\#_{tsr}(e)$, $\#_{twr}(e)$, can be set from the model as they are parameters of the resource procllet. To ease the LP formulation, if e is unrelated to a resource, we set $\#_{rid}(e) = r^*$ to fresh identifier and $\#_{tsr}(e) = \#_{twr}(e) = 0$; $\#_{qid}(e)$ and $\#_{twq}(e)$ are set correspondingly. By annotating the events in E_2 as stated above, we obtain $\pi_2 = (E_2, <_2, AN_2, \{pid, rid, qid\}, \#^2)$. Moreover, we can update the SPO $<_2$ by inferring $<_{rid}$ and $<_{qid}$ from $\#_{time}(e)$ for all events where $\#_{rid}(e) \neq \perp$ and $\#_{qid}(e) \neq \perp$ (see Def. 4.6).

The system-level run π_2 , contains complete entity-level runs for pid (except for missing time stamps). The entity-level runs queues (qid) and resources (rid) already contain all events to be complete wrt. S but only the observed events are ordered (due to their time stamps). For example, Fig. 7(d) shows the entity-level run $\pi_{qid}^{m4:d1}$ containing events f_8, f_9, f_{16}, f_{15} with only $f_9 <_{qid} f_{16}$. Next, we define constraints based on the information in this intermediate run π to infer timestamps for all unobserved events.

6.2. Restoring timestamps of unobserved events by linear programming

The SPO $\pi_2 = (E_2, <_2, AN_2, \{pid, rid, qid\}, \#^2)$ obtained in Sect. 6.1 from partial log L_1 includes all unobserved events $E_u = E_2 \setminus E_1$ of the complete log, but lacks timestamps for each $e \in E_u$, $\#_{time}(e) = \perp$. Each observed $e \in E_1$ has a timestamp $\#_{time}(e)$ and we also added minimum service time $\#_{tsr}$, waiting time $\#_{twr}(e)$ of the resource $\#_{rid}(e)$ involved in e and minimum waiting time $\#_{twq}(e)$ of the queue involved in e . We now define a constraint satisfaction problem that specifies the earliest $\#_{tmin}(e)$ and latest $\#_{tmax}(e)$ timestamps for each $e \in E_u$ so that all earliest (latest) timestamps yield a consistent ordering of all events in E wrt. $<_{pid}$ (events follow the process), $<_{rid}$ (events follow resource life-cycle), and $<_{qid}$ (events satisfy queueing behavior). The problem formulation propagates the known $\#_{time}(e)$ values along with the different case notions $<_{pid}$, $<_{rid}$, $<_{qid}$, using tsr , twr , twq . For that, we introduce variables $x_e^{tmin}, x_e^{tmax} \geq 0$ for representing event attributes $tmin, tmax$ of each $e \in E_u$. For all observed events $e \in E_1$, we set $x_e^{tmin} = x_e^{tmax} = \#_{time}(e)$ 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, we assume for the sake of simpler constraints presented in this paper, that all observed events are start events (which is in line with logging in an MHS). The constraints can easily be reformulated to assume only complete events were observed (as in most business process event logs) or a mix (requiring further case distinctions).

6.2.1. Propagate information along process traces

The first group propagates constraints for $\#_{time}(e)$ along \prec_{pid} , i.e., for each process-level run (viz. process trace) π_{pid}^{id} of pid in π . By the steps in Sect. 6.1, events in π_{pid}^{id} are totally ordered and derived from the trace $\sigma_{pid}^{id} = \langle e_1 \dots e_m \rangle$. Each process step has a start and a complete event in σ_{pid}^{id} , 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 $\#_{tsr}(e_{2i-1})$ in Sect. 6.1). Thus the following constraints must hold for the earliest and latest time of e_{2i-1} and e_{2i} .

$$x_{e_{2i}}^{tmin} = x_{e_{2i-1}}^{tmin} + \#_{tsr}(e_{2i-1}), \quad (1)$$

$$x_{e_{2i}}^{tmax} = x_{e_{2i-1}}^{tmax} + \#_{tsr}(e_{2i-1}). \quad (2)$$

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:

$$x_{e_{2i-1}}^{tmin} \leq x_{e_{2i-1}}^{tmax}. \quad (3)$$

We write $e_i^s = e_{2i-1}$ for the start event of the i -th process step in σ_{pid}^{id} and write $\theta_{pid}^{id} = \langle e_1^s, \dots, e_m^s \rangle$ for the sub-trace of start events of σ_{pid}^{id} . Any event $e_i^s \in \theta_{pid}^{id}$ that was observed in L_1 , i.e., $e_i^s \in E_1$, has $\#_{time}(e_i^s) \neq \perp$ defined. By Def. 5.9, σ_{pid}^{id} as well as θ_{pid}^{id} always start and end with observed events, i.e., $e_1^s, e_y^s \in E_1$ and $\#_{time}(e_1^s), \#_{time}(e_y^s) \neq \perp$. An unobserved event e_i^s has no timestamp $\#_{time}(e_i^s) = \perp$ yet, but $\#_{time}(e_i^s)$ is bounded by $\#_{time}(e_1^s)$ (minimally) and $\#_{time}(e_y^s)$ (maximally). Furthermore, any two succeeding start events in $\theta_{pid}^{id} = \langle \dots, e_{i-1}^s, e_i^s, \dots \rangle$ are separated by the service time $\#_{tsr}(e_{i-1}^s)$ of step e_{i-1}^s and the waiting time $\#_{twq}(e_i)$ of the queue from e_{i-1} to e_i . Similar to Eq. 1 and 2, we formulate this constraint for both x_e^{tmin} and x_e^{tmax} variables:

$$x_{e_k^s}^{tmin} \geq x_{e_{k-1}^s}^{tmin} + (\#_{tsr}(e_{k-1}^s) + \#_{twq}(e_k^s)), \quad (4)$$

$$x_{e_k^s}^{tmax} \leq x_{e_{k+1}^s}^{tmax} - (\#_{tsr}(e_k^s) + \#_{twq}(e_{k+1}^s)). \quad (5)$$

Fig. 8 uses the *Performance Spectrum* [3] to illustrate the effect of applying our approach step by step to the partially complete traces of Fig. 7 obtained in the steps of Sect. 6.1. The straight lines in Fig. 8(a) from f_1 to f_9 (for $pid=53$) and from f_{12} to f_{16} (for $pid=54$) illustrate that L_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 tsr, twr, twq . The sum $tsr + twq$ is visualized as bars on the time axis in Fig. 8(a), the duration of twr is shown in Fig. 8(b).

We now explain the effect of applying Eq. 4 on $pid=53$ for f_3, f_5 and f_7 . We have $\theta_{pid}^{53} = \langle f_1, f_3, f_5, f_7, f_9 \rangle$ with f_1 and f_9 observed, thus $x_{f_i}^{tmin} = x_{f_i}^{tmax} = \#_{time}(f_i)$ for $i \in \{1, 9\}$. By Eq. 4, we obtain the lower-bound for the time for f_3 by $x_{f_3}^{tmin} \geq x_{f_1}^{tmin} + \#_{tsr}(f_1) + \#_{twq}(f_3)$ with $\#_{tsr}(f_1)$ and $\#_{twq}(f_3)$ the service time of resource $c1$ and waiting time of queue $c1:m2$. Similarly, Eq. 4 gives the lower bound for f_5 from the lower bound from f_3 etc. Conversely, the upper bounds $x_{f_i}^{tmax}$ are

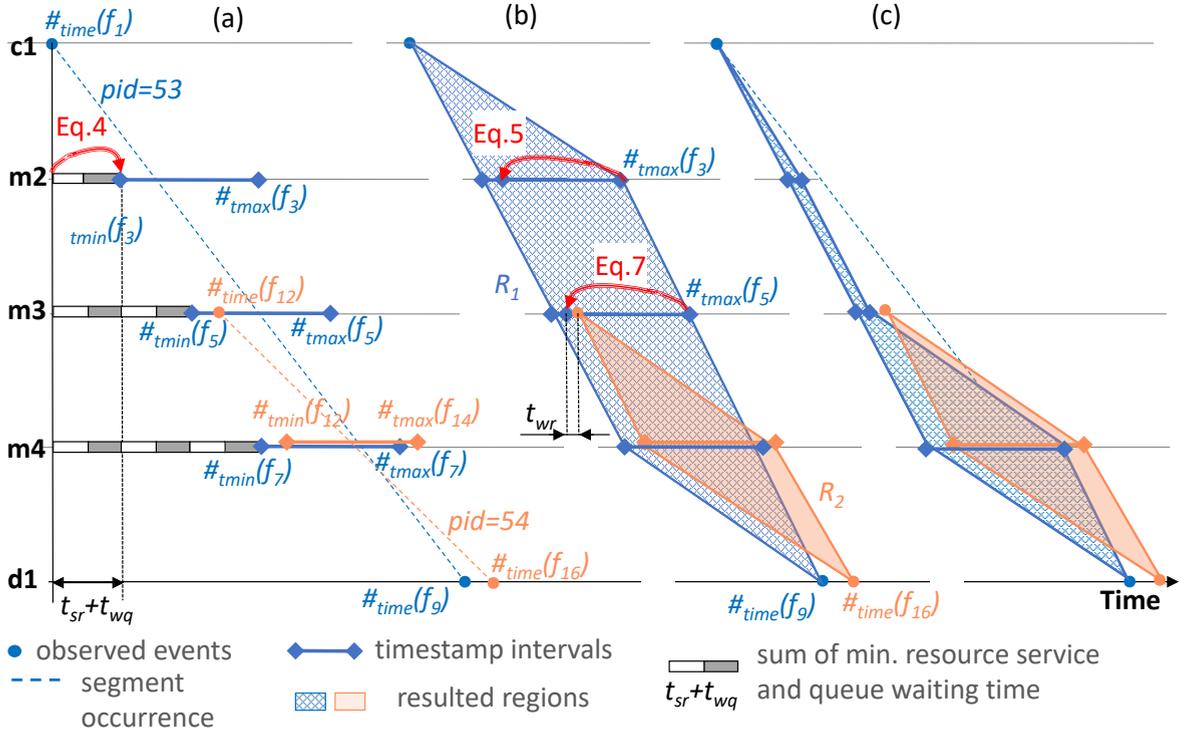


Figure 8. Equations 1-5 define time intervals for unobserved events (a), defining regions for the possible traces (b). Equations 6-7 propagate orders of cases observed on one resource to other resources (b), resulting in tighter regions (c).

derived from f_9 “downwards” by Eq. 5. This way, we obtain for each $f_i \in \theta_{pid}^{53}$ an initial interval for the time of f_i between the bounds $x_{f_i}^{tmin} \leq x_{f_i}^{tmax}$ as shown by the intervals in Fig. 8(a). As $x_{f_1}^{tmin} = x_{f_1}^{tmax} = \#time(f_1)$ and $x_{f_9}^{tmin} = x_{f_9}^{tmax} = \#time(f_9)$, the lower and upper bounds for the unobserved events in θ_{pid}^{53} form a polygon as shown in Fig. 8(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_{pid}^{53} ; we call this polygon the *region* of case 53. The region for case 54 overlaps with the region for case 53.

6.2.2. Propagate information along resource traces

We now introduce a second group of constraints by which we infer more tight bounds for $x_{e_i}^{tmin}$ and $x_{e_i}^{tmax}$ based on the overlap with other regions. While the first group of constraints traversed entity traces along pid (i.e., process traces), the second group of constraints traverses entity traces for resources along rid .

Each resource trace π_{rid}^r in π_2 , contains all events E_{rid}^r resource r was involved in – across multiple different process traces. The SPO \langle_{rid}^r orders *observed* events of this resource trace due to their known timestamps; e.g. in Fig. 8(b) $f_9 \langle_{rid}^{m1} f_{16}$ with f_9 from $pid=53$ and f_{16} from $pid=54$.

The order of the two events $e_{p1}^s <_{rid}^r e_{p2}^s$ for the *same* step $\#_{act}(e_{p1}^s) = \#_{act}(e_{p2}^s) = t_1$ in different cases $\#_{pid}(e_{p1}^s) = p1 \neq \#_{pid}(e_{p2}^s) = p2$ propagates “upwards” and “downwards” the process traces π_{pid}^{p1} and π_{pid}^{p2} as follows. Let events $f_{p1}^s \in E_{pid}^{p1}$ and $f_{p2}^s \in E_{pid}^{p2}$ be events in process traces π_{pid}^{p1} and π_{pid}^{p2} of the same step $\#_{act}(f_{p1}^s) = \#_{act}(f_{p2}^s) = t_n$. 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 proctet (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 $f_{p1}^s <_{rid}^{r2} f_{p2}^s$ on the resource $r2$ involved in t_n (as the case $p1$ cannot overtake the case $p2$ along this path). Thus $x_{f_{p1}^s}^{tmin} \leq x_{f_{p2}^s}^{tmin}$ must hold. More specifically, $x_{f_{p1}^s}^{tmin} + \#_{tsr}(f_{p1}^s) + \#_{twr}(f_{p1}^s) \leq x_{f_{p2}^s}^{tmin}$ must hold as the service time and waiting time of the resource involved in f_{p1}^s must elapse.

For any pair $e_{p1}^s, e_{p2}^s \in E_{rid}^r$ with $e_{p1}^s <_{rid}^r e_{p2}^s$ and any other trace θ_{rid}^{r2} for resource $r2$ and any pair $f_{p1}^s, f_{p2}^s \in E_{rid}^{r2}$ such that $\#_{pid}(e_{p1}^s) = \#_{pid}(f_{p1}^s), \#_{pid}(e_{p2}^s) = \#_{pid}(f_{p2}^s)$ and transition $\#_{act}(e_{p1}^s)$ is in FIFO relation with $\#_{act}(f_{p1}^s)$, we generate the following constraint for *tmin* between different process cases $p1$ and $p2$:

$$x_{f_{p1}^s}^{tmin} \leq x_{f_{p2}^s}^{tmin} - (\#_{tsr}(f_{p1}^s) + \#_{twr}(f_{p1}^s)), \quad (6)$$

and the following constraint for *tmax*:

$$x_{f_{p1}^s}^{tmax} \leq x_{f_{p2}^s}^{tmax} - (\#_{tsr}(f_{p1}^s) + \#_{twr}(f_{p1}^s)), \quad (7)$$

In the example of Fig. 8(b), we observe $f_9 <_{d1}^{d1} f_{16}$ (both of transition $d1_s$) along resource $d1$ at the bottom of Fig. 8(b). By Fig. 4, $d1_s$ and $m3_s$ are in FIFO-relation. Applying Eq. 7 yields $x_{f_5}^{tmax} \leq \#_{time}(f_{12}) - (\#_{tsr}(f_5) + \#_{twr}(f_5))$, 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 Eq. 5, the tighter upper bound for f_5 also propagates along the trace $pid=53$ to f_3 , i.e., $x_{f_3}^{tmax} \leq x_{f_5}^{tmax} - (\#_{tsr}(f_3) + \#_{twq}(f_5))$, resulting in a tighter region as shown in Fig. 8(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 Eq. 4,6 in a similar way. In general, the more cases interact through shared resources, the more accurate timestamp intervals can be restored by Eq. 1-7 as we will show in Sect. 7.

To construct the linear program, we generate equations 1 to 5 by iteration of each process trace in L_2 . Further, iterate over each resource trace and for each pair of events $e_{p1} <_{rid}^r e_{p2}$ we generate equations 6,7 for each other pair of events $f_{p1} <_{rid}^{r2} f_{p2}$ that is in FIFO relation. The objective function to maximize is the sum of all intervals $\sum_{e \in E_2} (x_e^{tmax} - x_e^{tmin})$ to maximize the coverage of possible time-stamp values by those intervals.

Solving this linear program assigns to each event $e \in E_2$ upper and lower bounds $\#_{tmin}(e)$ and $\#_{tmax}(e)$ for $\#_{time}(e)$; $\#_{tmin}(e) = \#_{time}(e) = \#_{tmax}(e)$ for all $e \in E_1$ (by 1 and 2 the solutions for the start events propagate to complete events with time difference tsr). By setting $\#_{time}(e) = \#_{tmin}(e)$ (or $\#_{time}(e) = \#_{tmax}(e)$) we obtain $L_3 = (E_2, AN_3, \{pid, rid, qid\}, \#^3)$ where the SPO $<_3$ of the system-level run $\pi(L_3)$ refines the SPO $<_2$ constructed explicitly in Sect. 6.1.

By oracle O_1 , $\sigma(L_3, pid)$ can be replayed on the P-proctet.

By 1 and 6, for any two events $e \prec_{rid} e'$ the time difference is $\#_{time}(e') - \#_{time}(e) > twr$ or $\#_{time}(e') - \#_{time}(e) > tsr$ of the corresponding R-proclet R_i (depending on whether e replays by the start or the complete transition of R_i). Thus, $\sigma(L_3, r)$ can be replayed on the correspond R-proclet for any resource r in L_3 .

By 1 and 4, the time-stamps of $e \prec_{pid} e'$ where e replays t_{enq} and e' replays t_{deq} of a Q-proclet Q_i have at least time difference twq of Q_i (i.e., the time constraint of Q_i is satisfied). If for two process cases $p1$ and $p2$ we observe $e_{p1} \prec_{rid} e_{p2}$ at the same step $\#_{act}(e_{p1}) = \#_{act}(e_{p2})$ with $\#_{pid}(e_1) = p1 \neq p2 = \#_{pid}(e_2)$ at some step, then we also observe $f_{p1} \prec_{rid} f_{p2}$ at another step $\#_{act}(f_{p1}) = \#_{act}(f_{p2})$ with $\#_{pid}(f_1) = p1 \neq p2 = \#_{pid}(f_2)$ at later events $e_1 \prec_{pid} f_1$ and $e_2 \prec_{pid} f_2$ (by 6 and 7). As in a PQR system, for each queue, the enqueue transition synchronizes with a different resources than the dequeue transition, the relation $e_{p1} \prec_{qid} e_{p2}$ and $f_{p1} \prec_{qid} f_{p2}$ 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, $\sigma(L_3, q)$ can be replayed on the correspond Q-proclet for any queue q in L_3 .

Altogether, L_3 is a complete log that can be replayed on the PQR system S (by Def. 5.8 and Def. 5.4).

7. 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 resource or stop/restart of an MHS conveyor, on the accuracy of restored timestamps and the computed load? For that, we extended the interactive ProM plug-in ‘‘Performance Spectrum Miner’’ with an implementation of our approach that solves the constraints using heuristics¹. As input we considered the process of a part of real-life BHS shown in Fig. 9 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

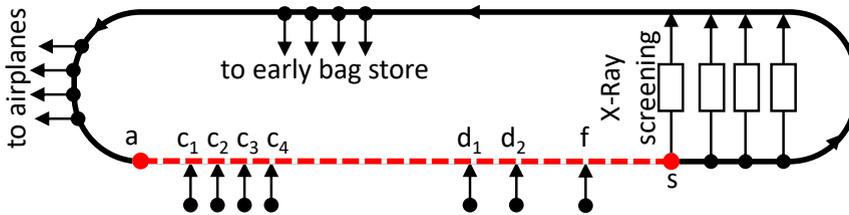


Figure 9. In the BHS bags come from check-in counters c_{1-4} and another terminals d_{1-2}, f , go through mandatory screening and continue to other locations.

¹The simulation model, simulation logs, ProM plugin, and high-resolution figures are available on <https://github.com/processmining-in-logistics/psm/tree/rel>.

RL contained both performance characteristics. All logs were partial as described in Sect. 5.5. We selected the acyclic fragment highlighted in Fig. 9 for restoring timestamps of steps c_{1-4} , d_{1-2} , f , s .

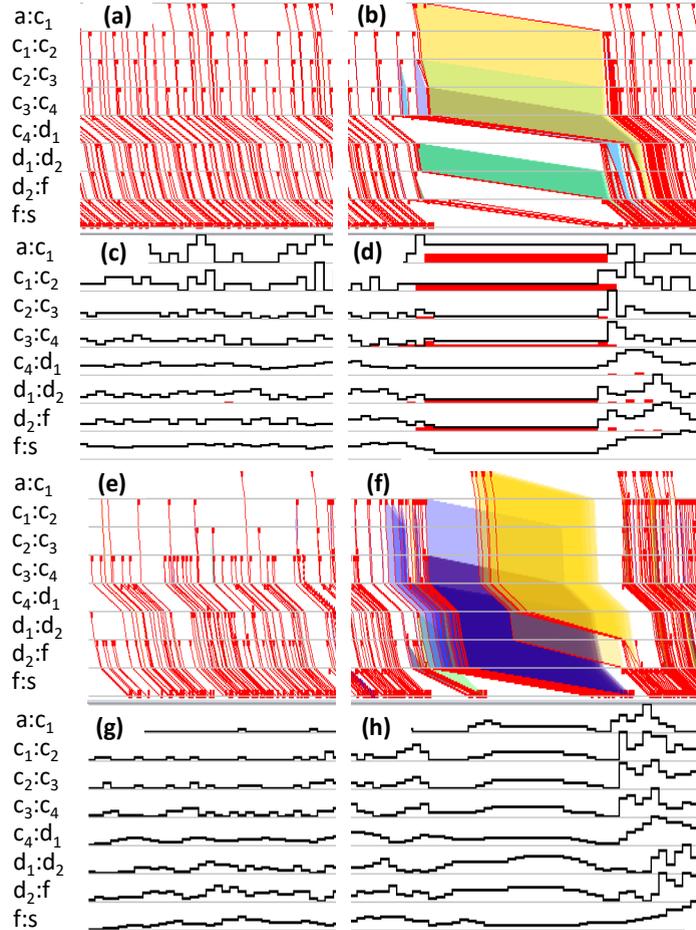


Figure 10. Restored Performance Spectrum for synthetic (a,b) and real-life (e,f) logs. The estimated load (computed on estimated timestamps) for synthetic (c,d) and real-life (g,h) logs. For the synthetic logs, the load error is measured and shown in red (c,d). Colored-shaded regions indicate for selected traces the boundaries of timestamps of reconstructed events between different observed events a to s (yellow), c_1 to s (blue), d_1 to s (green).

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 minimal service and waiting times of all 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 regular behavior, we observed very narrow time intervals for the estimated timestamps, shown in Fig. 10(a), and a MAE of $< 5\%$. The MAE of the estimated load (computed on estimated timestamps), shown in Fig. 10(c), was

$< 2\%$. For SL with blockage behavior, the intervals grew proportionally with the duration of blockages (Fig. 10(b)), leading to a proportional growth of the MAE for the timestamps. However, the MAE of the estimated load (Fig. 10(d)) was at most 4%. The load MAE for different processing steps for both scenarios are shown in Table 3. Notably, both observed and reconstructed load showed load peaks each time the conveyor belt starts moving again.

Table 3. The estimated load (computed on estimated timestamps) Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) are shown in % of max. load.

Scenario	$c_4 : d_1$	$d_1 : d_2$	$f : s$
no blockages			
MAE	0.16	0.22	0.17
RMSE	1.01	1.66	0.89
blockages			
MAE	1.67	3.19	0.15
RMSE	4.8	7.17	0.75

When evaluating on the real-life event log, we measured errors of timestamps estimation as the length of the estimated intervals (normalized over the sum of minimal service and waiting times of all involved steps). Performance spectra built using the restored RL logs are shown in Fig. 10(e,f), and the load computed using these logs is shown in Fig. 10(g,h). The observed MAE was $< 5\%$ in regular behavior and increased proportionally as observed on SL. The load error could not be measured, but similarly to 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 tsr , twr , twq , 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 shows known characteristics from real-life systems on SL and RL (Q3).

8. Conclusion

In this paper, we studied the problem of repairing a partial event log with missing events for the performance analysis of systems where cases interact and compete for shared limited resources. We addressed the problem of repairing partial event logs that contain only a subset of events which impede the performance analysis of systems with shared limited resources and queues.

To study and solve the problem, we had to develop novel syntactic and semantic models for behavior over multiple entities. We specifically introduced a generalized model of event data over multiple behavioral entities that can be viewed, both, as sequential traces (with shared events) and as a partial order over the entire system behavior. We have shown in solving our problem that both perspectives are needed when reasoning about behavior of multiple entities.

To express domain knowledge about resources and queues, we had to extend the model of synchronous proclats [6] with concepts for time and data, resulting in the notion of CPN proclat systems. A side effect of our work is a replay semantics for regular coloured Petri nets. We defined the sub-class of PQR systems to model processes served by shared resources and queues. Our model allows to decompose the interactions of resources and queues over multiple process cases into entity traces for process cases, resources and queues that synchronize on shared events (both on the syntactic and on the semantic level).

We exploit the decomposition when restoring missing events along the process traces using trace alignment [18]. We exploit the synchronization when formulating linear programming constraints over timestamps of restored events along, both, the process and the resource traces. As a result, we obtain timestamps which are consistent for all events along the process, resource, and queue dimensions. The evaluation of our implementation on synthetic and real-life data shows errors of the estimated timestamps and of derived performance characteristics (i.e., load) of $< 5\%$ under regular performance, while correctly restoring real-life dynamics (i.e. load peaks) after irregular performance behavior.

Limitations The work made several limiting assumptions. (1) Although the proclat formalism allows for arbitrary, dynamic synchronizations between process steps, resources, and queues, we limited ourselves in this work to a static known resource/queue id per process step. The limitation is not severe for some use cases such as analyzing MHS, but generalizing oracle O_2 to a dynamic setting is an open problem. (2) The LP constraints to restore timestamps assume an acyclic process proclat 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. (3) 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. (4) 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. Points (3) and (4) require attention to further improve event log quality for performance analysis.

Future work Besides addressing the above limitations, our novel syntactic and semantic models open up new alleys of research for modeling and analyzing behavior over multiple entities, including more general conformance and process discovery. Moreover, the replay semantics for coloured Petri nets is likely to enable new kinds of process mining and conformance checking analyses beyond the types of systems studied in this paper.

Acknowledgements The research leading to these results has received funding from Vanderlande Industries in the project “Process Mining in Logistics”. We also thank Mitchel Brunings for his comments that greatly improved our approach.

References

- [1] Maruster L, van Beest NRTP. Redesigning business processes: a methodology based on simulation and process mining techniques. *Knowl. Inf. Syst.*, 2009. **21**(3):267–297. doi:10.1007/s10115-009-0224-0.

- [2] Márquez-Chamorro AE, Resinas M, Ruiz-Cortés A. Predictive Monitoring of Business Processes: A Survey. *IEEE Transactions on Services Computing*, 2018. **11**(6):962–977. doi:10.1109/TSC.2017.2772256.
- [3] Denisov V, Fahland D, van der Aalst WMP. Unbiased, Fine-Grained Description of Processes Performance from Event Data. In: Weske M, Montali M, Weber I, vom Brocke J (eds.), *Business Process Management*. Springer International Publishing, Cham. 2018 pp. 139–157. ISBN:978-3-319-98648-7.
- [4] Ahmed T, Pedersen TB, Calders T, Lu H. Online Risk Prediction for Indoor Moving Objects. In: 2016 17th IEEE International Conference on Mobile Data Management (MDM), volume 1. 2016 pp. 102–111. doi:10.1109/MDM.2016.27.
- [5] Denisov V, Fahland D, van der Aalst WMP. Predictive Performance Monitoring of Material Handling Systems Using the Performance Spectrum. In: 2019 International Conference on Process Mining (ICPM). 2019 pp. 137–144. doi:10.1109/ICPM.2019.00029.
- [6] Fahland D. Describing Behavior of Processes with Many-to-Many Interactions. In: Donatelli S, Haar S (eds.), *Application and Theory of Petri Nets and Concurrency*. Springer International Publishing, Cham. 2019 pp. 3–24. ISBN:978-3-030-21571-2.
- [7] Jensen K, Kristensen LM. Colored Petri nets: a graphical language for formal modeling and validation of concurrent systems. *Commun. ACM*, 2015. **58**(6):61–70. doi:10.1145/2663340.
- [8] Schrijver A. *Theory of Linear and Integer Programming*. John Wiley & Sons, Chichester, 1986.
- [9] van der Aalst WMP. *Process Mining - Data Science in Action, Second Edition*. Springer, 2016. ISBN-10:9783662498507, 13:978-3662498507.
- [10] Senderovich A, Francescomarino CD, Maggi FM. From knowledge-driven to data-driven inter-case feature encoding in predictive process monitoring. *Inf. Syst.*, 2019. **84**:255–264. doi:10.1016/j.is.2019.01.007.
- [11] Gans N, Koole G, Mandelbaum A. Telephone Call Centers: Tutorial, Review, and Research Prospects. *Manufacturing & Service Operations Management*, 2003. **5**(2):79–141. doi:10.1287/msom.5.2.79.16071.
- [12] Brown L, Gans N, Mandelbaum A, Sakov A, Shen H, Zeltyn S, Zhao L. Statistical Analysis of a Telephone Call Center. *Journal of the American Statistical Association*, 2005. **100**(469):36–50. doi:10.1198/016214504000001808.
- [13] Senderovich A, Weidlich M, Gal A, Mandelbaum A. Queue Mining - Predicting Delays in Service Processes. In: *Advanced Information Systems Engineering - 26th International Conference, CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014. Proceedings*, volume 8484 of *Lecture Notes in Computer Science*. Springer, 2014 pp. 42–57. doi:10.1007/978-3-319-07881-6_4.
- [14] Senderovich A, Beck J, Gal A, Weidlich M. Congestion Graphs for Automated Time Predictions. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019. **33**:4854–4861. doi:10.1609/aaai.v33i01.33014854.
- [15] Suriadi S, Andrews R, ter Hofstede A, Wynn M. Event log imperfection patterns for process mining: Towards a systematic approach to cleaning event logs. *Information Systems*, 2017. **64**:132 – 150. doi:https://doi.org/10.1016/j.is.2016.07.011.
- [16] Conforti R, La Rosa M, ter Hofstede A. Timestamp Repair for Business Process Event Logs. Technical report, 2018. URL <http://hdl.handle.net/11343/209011>.
- [17] Martin N, Depaire B, Caris A, Schepers D. Retrieving the resource availability calendars of a process from an event log. *Information Systems*, 2020. **88**:101463. doi:https://doi.org/10.1016/j.is.2019.101463.

- [18] Carmona J, van Dongen B, Solti A, Weidlich M. Conformance Checking - Relating Processes and Models. Springer, 2018. ISBN:978-3-319-99414-7.
- [19] Pegoraro M, van der Aalst WMP. Mining Uncertain Event Data in Process Mining. In: International Conference on Process Mining, ICPM 2019, Aachen, Germany, June 24-26, 2019. IEEE, 2019 pp. 89–96. doi:10.1109/ICPM.2019.00023.
- [20] Pegoraro M, Uysal MS, van der Aalst WMP. Discovering Process Models from Uncertain Event Data. In: Di Francescomarino C, Dijkman R, Zdun U (eds.), Business Process Management Workshops. Springer International Publishing, Cham. 2019 pp. 238–249. ISBN:978-3-030-37453-2.
- [21] van der Aalst WMP, Barthelmess P, Ellis CA, Wainer J. Proclets: A Framework for Lightweight Interacting Workflow Processes. *International Journal of Cooperative Information Systems*, 2001. **10**(04):443–481.
- [22] van der Aalst WMP, Berti A. Discovering Object-centric Petri Nets. *Fundam. Informaticae*, 2020. **175**(1-4):1–40. doi:10.3233/FI-2020-1946.
- [23] Ghilardi S, Gianola A, Montali M, Rivkin A. Petri Nets with Parameterised Data - Modelling and Verification. In: Fahland D, Ghidini C, Becker J, Dumas M (eds.), Business Process Management - 18th International Conference, BPM 2020, Seville, Spain, September 13-18, 2020, Proceedings, volume 12168 of *Lecture Notes in Computer Science*. Springer, 2020 pp. 55–74. doi:10.1007/978-3-030-58666-9_4.
- [24] Steinau S, Andrews K, Reichert M. Coordinating Large Distributed Process Structures. In: Reinhartz-Berger I, Zdravkovic J, Gulden J, Schmidt R (eds.), Enterprise, Business-Process and Information Systems Modeling - 20th International Conference, BPMDS 2019, 24th International Conference, EMMSAD 2019, Held at CAiSE 2019, Rome, Italy, June 3-4, 2019, Proceedings, volume 352 of *Lecture Notes in Business Information Processing*. Springer, 2019 pp. 19–34. doi:10.1007/978-3-030-20618-5_2.
- [25] Popova V, Fahland D, Dumas M. Artifact Lifecycle Discovery. *Int. J. Cooperative Inf. Syst.*, 2015. **24**(1):1550001:1–1550001:44. doi:10.1142/S021884301550001X.
- [26] van der Aalst WMP. Object-Centric Process Mining: Dealing with Divergence and Convergence in Event Data. In: Ölveczky PC, Salaün G (eds.), Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings, volume 11724 of *Lecture Notes in Computer Science*. Springer, 2019 pp. 3–25. doi:10.1007/978-3-030-30446-1_1.
- [27] Lu X, Nagelkerke M, van de Wiel D, Fahland D. Discovering Interacting Artifacts from ERP Systems. *IEEE Trans. Serv. Comput.*, 2015. **8**(6):861–873. doi:10.1109/TSC.2015.2474358.
- [28] Werner M, Gehrke N. Multilevel Process Mining for Financial Audits. *IEEE Trans. Serv. Comput.*, 2015. **8**(6):820–832. doi:10.1109/TSC.2015.2457907.
- [29] Esser S, Fahland D. Storing and Querying Multi-dimensional Process Event Logs Using Graph Databases. In: Francescomarino CD, Dijkman RM, Zdun U (eds.), Business Process Management Workshops - BPM 2019 International Workshops, Vienna, Austria, September 1-6, 2019, Revised Selected Papers, volume 362 of *Lecture Notes in Business Information Processing*. Springer, 2019 pp. 632–644. doi:10.1007/978-3-030-37453-2_51.
- [30] Berti A, van der Aalst WMP. Extracting Multiple Viewpoint Models from Relational Databases. In: Ceravolo P, van Keulen M, López MTG (eds.), Data-Driven Process Discovery and Analysis - 8th IFIP WG 2.6 International Symposium, SIMPDA 2018, Seville, Spain, December 13-14, 2018, and 9th International Symposium, SIMPDA 2019, Bled, Slovenia, September 8, 2019, Revised Selected Papers, volume 379 of *Lecture Notes in Business Information Processing*. Springer, 2019 pp. 24–51. doi:10.1007/978-3-030-46633-6_2.

- [31] Esser S, Fahland D. Multi-Dimensional Event Data in Graph Databases. *J. Data Semant.*, 2021. **10**(1):109–141. doi:10.1007/s13740-021-00122-1.
- [32] Denisov V, Fahland D, van der Aalst WMP. Repairing Event Logs with Missing Events to Support Performance Analysis of Systems with Shared Resources. In: Janicki R, Sidorova N, Chatain T (eds.), *Application and Theory of Petri Nets and Concurrency - 41st International Conference, PETRI NETS 2020*, Paris, France, June 24-25, 2020, Proceedings, volume 12152 of *Lecture Notes in Computer Science*. Springer, 2020 pp. 239–259. doi:10.1007/978-3-030-51831-8_12.
- [33] Rosa-Velardo F, de Frutos-Escrig D. Name Creation vs. Replication in Petri Net Systems. *Fundam. Inform.*, 2008. **88**(3):329–356.
- [34] van der Aalst WMP, Weijters AJMM, Maruster L. Workflow mining: discovering process models from event logs. *IEEE TKDE*, 2004. **16**:1128–1142. doi:10.1109/TKDE.2004.47.
- [35] van der Aalst WMP, Adriansyah A, Dongen B. Replaying History on Process Models for Conformance Checking and Performance Analysis. *WIREs Data Mining and Knowledge Discovery*, 2012. **2**:182–192. doi:10.1002/widm.1045.