Event Abstraction for Partial Order Patterns

Chiao-Yun Li[1,2], Sebastiaan J. van Zelst[2,1], and Wil M.P. van der Aalst[1,2]

[1] RWTH Aachen University, Aachen, Germany
{chiaoyun.li,wvdaalst}@pads.rwth-aachen.de
[2] Fraunhofer FIT, Birlinghoven Castle, Sankt Augustin, Germany
sebastiaan.van.zelst@fit.fraunhofer.de

Abstract. *Process mining* endeavors to extract fact-based insights into processes based on *event data* stored in information systems. Due to the variety of processes in different fields and organizations, there does not exist a universal technique to allow for putting the process mining outcome directly into action. Various techniques have been developed to support human analysis. Meanwhile, as raw event data are often provided at the system level, the *abstraction principle* is applied to "lift" the data to a higher level for human interpretation, which is called *event abstraction*. Owing to the limitation of the information systems deployed in practice, most abstraction techniques are developed based on the assumption that all process activities are performed sequentially, ignoring the fact that there may be activities performed concurrently or the relation of the activity executions could not be clearly defined. In this paper, we propose an event abstraction framework based on partial order patterns. We extract the candidate pattern instances and abstract event data based on the pattern instances identified. Moreover, we instantiate the framework and optimize the implementation. The framework is evaluated with synthetic event data, and a case study based on a real-life process is performed, demonstrating the applicability of the framework.

Keywords: Process mining · Event abstraction · Partial orders.

1 Introduction

Modern organizations rely on business processes executed with the support of information systems, which generate *event data* recorded during process execution. *Process mining* aims to extract valuable insights from such event data [2]. Numerous process mining techniques were developed to gain insights into various aspects; *process discovery* reveals the actual behavior of a process, which is often represented with a *process model* [4]; *conformance checking* identifies deviations in a process [6]; *performance analysis* detects inefficiencies and bottlenecks in a process [7]; *process enhancement* attempts to enhance a process model based on factual insights discovered [20].

To turn process mining results into actionable insights, the outcomes must be interpretable for humans, which is often achieved through the use of a process model annotated with relevant information for a limited number of activities.

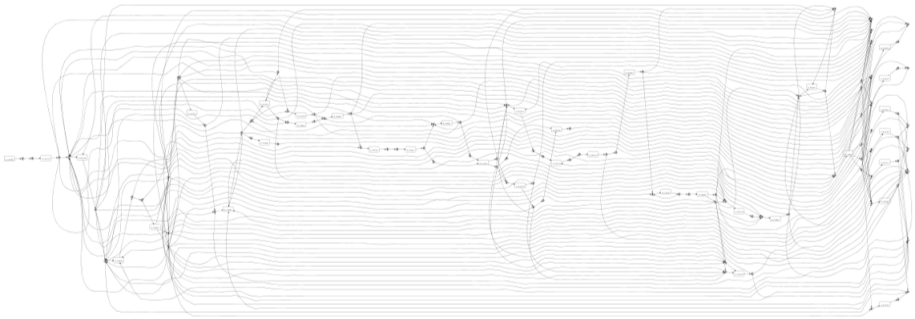


Fig. 1: A process model discovered based on real-life event data [19], which abstracts the behavior of 36 activities executed in 4,366 ways in a process instance.

Typically, process mining techniques are directly applied to raw event data, i.e., event data as recorded in information systems, resulting in outcomes that may be too detailed or complex for human analysts as shown in Figure 1, which is impossible for humans to derive valuable insights without going into detail. Due to the highly flexible and complex nature of real-life processes, the field of *event abstraction* emerged to abstract event data to a higher level based on the predefined or identified regularity of the execution of *activities*, i.e., well-defined process steps, for human interpretability. By abstracting event data to a higher level, the complexity is simplified, allowing stakeholders to understand and interpret the results.

As most information systems deployed in practice support sequence data, event data are often structured as a total order of the execution of activities. Consequently, most event abstraction techniques are developed based on the assumption that activities are executed sequentially. In practice, activities can be executed concurrently, e.g., when a person multitasks, and/or the order of executions cannot be clearly defined, e.g., when the executions are recorded at the granularity of *days*, which leads to *partially ordered* event data.

We propose a *framework to extract patterns from partially ordered event data*. By leveraging a *pattern class* defined by domain experts as an expected relation of the execution of concepts, e.g., activities, in a process, the framework identifies the corresponding *pattern instances*, i.e., the executions of the pattern class. The abstraction is achieved by aggregating pattern instances; thereby, the framework can be iteratively applied to construct a hierarchy of abstractions. We initiate and implement the framework with a generic approach for identifying pattern instances. Furthermore, we optimize the framework for extracting *candidate pattern instances*, i.e., potential sets of event data that *may* be pattern instances. We apply the framework to synthetic event data and experiment with the effect of noises. To demonstrate the applicability, we conduct a case study based on real-life event data based on the abstraction obtained with the framework.

Table 1: A running example of partially ordered event data. Every row is a record representing an activity instance characterized by its identifier (`AID`), the identifier of the process instance it belongs to (`cid`), the activity name (`Activity`), and the duration of the execution (`Start` and `Complete Timestamp`).

cid	Activity (Abbre.)	AID	Start Timestamp	Complete Timestamp
1	Get Appointment (A)	1	2021-03-26 10:36:09	2021-03-26 10:36:09
1	Consult (C)	2	2021-03-26 11:07:53	2021-03-26 11:20:23
1	Review History (R)	3	2021-03-26 11:07:07	2021-03-26 11:22:10
1	Phlebotomize (P)	4	2021-03-26 13:36:16	2021-03-26 13:39:27
1	Conduct Lab Test (L)	5	2021-03-29 00:00:00	2021-04-04 00:00:00
1	Conduct Lab Test (L)	6	2021-03-29 00:00:00	2021-04-04 00:00:00
1	Diagnose (D)	7	2021-04-09 15:32:02	2021-04-09 15:47:20
1	Provide Treatment (T)	8	2021-04-15 00:00:00	2022-05-22 00:00:00
1	Provide Treatment (T)	9	2021-05-04 00:00:00	2022-05-26 00:00:00
1	Provide Treatment (T)	10	2021-05-22 00:00:00	2022-09-03 00:00:00
1	Phlebotomize (P)	11	2022-09-08 20:09:40	2022-09-08 20:11:51
1	Conduct Lab Test (L)	12	2022-09-11 00:00:00	2022-09-17 00:00:00
1	Conduct Lab Test (L)	13	2022-09-13 00:00:00	2022-09-18 00:00:00
1	Evaluate (E)	14	2022-09-21 05:04:36	2022-09-21 05:32:15

The paper is structured as follows. A running example is presented in Section 2. Section 3 introduces the mathematical concepts, which are applied to define the framework in Section 4. We introduce the implementation in Section 5 and show the experiments in Section 6. Finally, we review related work in Section 7 and discuss future directions in Section 8.

2 Running Example - A Treatment Procedure

Table 1 presents an excerpt of synthetic event data, which serves as a running example that we use throughout the paper. Every row represents an activity instance. The table records the activities executed in a treatment procedure of a patient. After the patient got an appointment at 10:36:09, he/she consulted the general practitioner and the practitioner reviewed the medical history of the patient at the same time. Then, a nurse phlebotomized the patient and sent the blood samples to two laboratories for different hematological tests. The reports were then sent back to the general practitioner for a diagnosis. Based on the outcome, the patient was sent to three specialists for further treatment. After roughly 1 year of treatment, the same blood tests are conducted again and the outcome of treatment is evaluated.

In Table 1, `Get Appointment` is executed in a time moment, as is often assumed in classical event data; `Conduct Lab Test` and `Provide Treatment` are recorded at the granularity of *days*, which causes unclear ordering, e.g., the two lab tests conducted at the first time; other activities are executed and recorded in time duration. Due to the time interleaving and different granularity recorded, such data form *partially ordered* event data[1], requiring a different abstraction mechanism compared to sequentially ordered event data.

Figure 2 visualizes a pattern class, assumed to be provided by domain experts, which specifies that two lab tests must be performed concurrently *after* phlebotomization. By representing an activity instance with its abbreviated activity and its identifier as a subscript, we can extract three sets of activity instances in Table 1 given the pattern class: $\{P_4, L_5, L_6\}$, $\{P_{11}, L_{12}, L_{13}\}$, and $\{P_4, L_{12}, L_{13}\}$. Every set of activity instances forms a *pattern instance*. As the blood sample phlebotomized with P_4 is used for L_5 and L_6, and the blood sample collected with P_{11} is used for L_{12} and L_{13}, we characterize the former two pattern instances as *local pattern instances*.

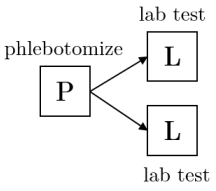


Fig. 2: Visualization of a pattern class.

3 Preliminaries

Let X be an arbitrary set. $\mathcal{P}(X) = \{X' \mid X' \subseteq X\}$ denotes the powerset of X and $|X|$ denotes the number of elements in X. A sequence over X is a function $\sigma : \{1, 2, ..., n\} \to X$, where σ is written as $\langle x_1, x_2, \ldots, x_n \rangle$. A strict partial order is a binary relation $\prec \subseteq X \times X$ on X, written as $(X, \prec)$, which is irreflexive ($\forall x \in X, x \not\prec x$), asymmetric ($\forall x, y \in X, x \prec y \Rightarrow y \not\prec x$), and transitive ($\forall x, y, z \in X, x \prec y \wedge y \prec z \Rightarrow x \prec z$). $(X, \prec)$ denotes the covering relation of $(X, \prec)$ such that $\forall x_1, x_2 \in X (x_1 \prec x_2)$, we write $x_1 \prec x_2$ if and only if $\nexists x' \in X (x_1 \prec x' \wedge x' \prec x_2)$. For simplicity, we write $\underline{(X, \prec)} = X$ as the shorthand for the elements in $(X, \prec)$ and refer to a strict partial order as a partial order.

Given an arbitrary set X, l is a function of X to a set of labels; a partial order on X with such a function is called a *labeled partial order* and written as $(X, \prec, l)$. Let X and Y be two arbitrary sets. Given $(X, \prec_X, l_X)$ and $(Y, \prec_Y, l_Y)$, $(X, \prec_X, l_X)$ and $(Y, \prec_Y, l_Y)$ are label-preserving isomorphic, denoted as $(X, \prec_X, l_X) \simeq (Y, \prec_Y, l_Y)$, iff there exists a bijective relation $b : X \to Y$ s.t. $\forall x_1, x_2 \in X, x_1 \prec_X x_2 \Leftrightarrow b(x_1) \prec_Y b(x_2)$, and $\forall x \in X, l_X(x) = l_Y(b(x))$.

Let $(X, \prec, l)$ be a labeled partial order on an arbitrary set X. Let z be an arbitrary element, where $z \notin X$, and l_z is a labeling function of z. The function $\mathsf{ADD}((X, \prec, l), z, l_z)$ adds z into $(X, \prec, l)$ s.t. $\mathsf{ADD}((X, \prec, l), z, l_z) = (X', \prec', l')$ where $X' = X \cup \{z\}$, $\prec' = \prec \cup (X \times \{z\})$.

[1] A collection of time intervals must be a partial order; nevertheless, the proposed framework is based on partial orders, which is more generically applicable. The example is provided with timestamps as a motivating example.

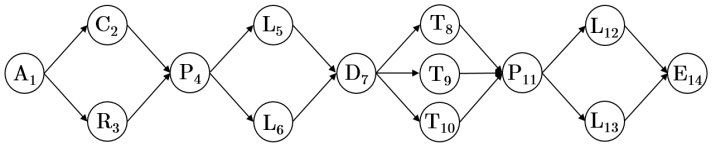


Fig. 3: Visualization of the covering relation of the activity instances in Table 1.

Definition 1 (Event Data). *A case is a process instance.* U_{con} *is the universe of concepts defined in a process, e.g., an activity;* U_{inst} *is the universe of the instances, e.g., an activity instance;* U_{cid} *is the universe of case identifiers. A log,* $L = (\mathsf{CI}, \ , \pi_{con}, \pi_{cid})$*, where*

- $\mathsf{CI} \ \ U_{inst}$ *is a set of instances;*
- $\ \ = \mathsf{CI} \ \ \mathsf{CI}$ *is a partial order on* CI*;*
- $\pi_{con} \colon \mathsf{CI} \ ! \ U_{con}$*, where* $\pi_{con}(\mathsf{ci})$ *is the concept of an instance* $\mathsf{ci} \ 2 \ \mathsf{CI}$*;*
- $\pi_{cid} \colon \mathsf{CI} \ ! \ U_{cid}$*, where* $\pi_{cid}(\mathsf{ci})$ *is the identifier of the case that an instance* $\mathsf{ci} \ 2 \ \mathsf{CI}$ *belongs to.*

We let $\mathsf{CID}(L) = f\pi_{cid}(\mathsf{ci}) \ j \ \mathsf{ci} \ 2 \ \mathsf{CI}g$ *denote the case identifiers in* L*. Given* $c \ 2 \ \mathsf{CID}(L)$*,* $\mathsf{CI}_c = f\mathsf{ci} \ 2 \ \mathsf{CI} \ j \ \pi_{cid}(\mathsf{ci}) = cg$ *and* $\mathsf{c}_L = (\mathsf{CI}_c, \ _c)$*, where* $\ _c = \ \setminus (\mathsf{CI}_c \ \ \mathsf{CI}_c)$*.*

Figure 3 visualizes the covering relation of the instances in the case in Table 1. Every node represents an instance, which is labeled with its identifier as a subscript and the corresponding (abbreviated) concept, i.e., the activity. The arrows indicate the covering relation among the instances.

4 Framework

We introduce and define the framework in this section. First, we outline the mechanism of the proposed framework in Section 4.1. Based on the mathematical notations introduced, we define a pattern class and the corresponding pattern instances in Section 4.2. The extraction of candidate pattern instances is introduced in Section 4.3. Finally, we detail the abstraction with the identification of pattern instances in Section 4.4.

4.1 Overview

Figure 4 presents a schematic overview of the proposed framework. We assume that a log with partially ordered event data is provided. A pattern class can be defined by a domain expert or with the knowledge obtained during the exploration of the log. Given a pattern class, the framework exhaustively extracts all the candidate pattern instances in the log. Next, one identifies partial orders of pattern instances from the candidate pattern instances where the relationship between the identified pattern instances is defined in a flexible manner. Finally,

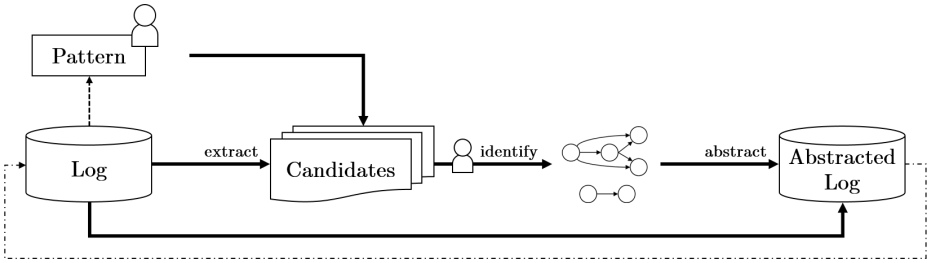


Fig. 4: A schematic overview of the framework. There are three key steps in the framework, the extraction of candidate pattern instances, the identification of partial orders of pattern instances, and the abstraction based on the pattern class.

an abstracted log based on the pattern class is constructed. Since the pattern class and the extraction and the identification of pattern instances support partially ordered event data, the proposed framework can be iteratively applied to the abstracted log based on another pattern class.

4.2 Pattern Class and Pattern Instance

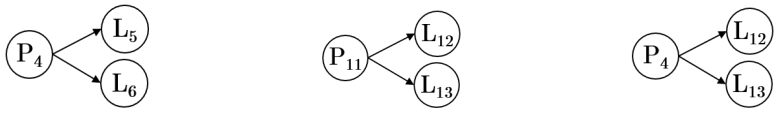
A *pattern class* is an expected relation of the execution of concepts; a *pattern instance* is, intuitively, a set of instances that adhere to the expectation. We formally define a pattern class as follows.

Definition 2 (Pattern Class). *A pattern class* $\mathsf{PC} = (X, \ , l)$*, where* $l\colon X \ ! \ U_{con}$*, is a labeled partial order where* $jXj \quad 2$*. We assume that a pattern class is a concept defined in a process such that* $\mathsf{PC} \ 2 \ U_{con}$*. We say that* $fl(x) \, j \, x \ 2 \ Xg$ *are the underlying concepts of a pattern class.*

A pattern instance is a labeled partial order of instances that is label-preserved isomorphic to a pattern class as defined below.

Definition 3 (Pattern Instance). *Let* L *be a log. Given* $c \ 2 \ \mathsf{CID}(L)$*, let* $\mathsf{c}_L = (\mathsf{CI}_c, \ _c)$*. Given* $\mathsf{CI} \quad \underline{\mathsf{c}_L}$ *and a pattern class* $\mathsf{PC} \ 2 \ U_{con}$*, a pattern instance* $\mathsf{pi} = (\mathsf{CI}, \ , \pi_{con})$*, where* $= \ _c \setminus (\mathsf{CI} \quad \mathsf{CI})$*, is a labeled partial order over* CI *where* $\mathsf{pi} \ ' \ \mathsf{PC}$*. We assume that a pattern instance* pi *is an instance s.t.* $\mathsf{pi} \ 2 \ U_{inst}$ *and* $\pi^{\mathsf{p}}_{cid}\colon U_{inst} \ ! \ U_{cid}$ *and* $\pi^{\mathsf{p}}_{con}\colon U_{inst} \ ! \ U_{con}$*, where* $\pi^{\mathsf{p}}_{cid}(\mathsf{pi}) = c$ *and* $\pi^{\mathsf{p}}_{con}(\mathsf{pi}) = \mathsf{PC}$*. We say that* $\underline{\mathsf{pi}}$ *are the underlying instances of* pi*.*

We define a pattern class with at least two elements since it is trivial with a single element as it simply implies the projection of the label of the element on instances. Meanwhile, with the constraint imposed on a pattern class, a pattern instance consists of at least two instances. Note that a pattern instance is defined in the context of a case and the definition above allows for the extraction of a pattern instance where the relations of the underlying instances are undefined. Figure 5 presents the pattern instances of the pattern class PC visualized in



(a) pi_1 (b) pi_2 (c) pi_3

Fig. 5: Visualization of the pattern instances of the pattern class PC defined in Figure 2 for the case in Table 1, where $\forall 1 \quad i \quad 3$, $\pi^{\mathsf{p}}_{con}(\mathsf{pi}_i) = \mathsf{PC}$ and $\pi^{\mathsf{p}}_{cid}(\mathsf{pi}_i) = 1$. Note that pi_1 and pi_2 are local pattern instances.

Figure 2. To differentiate with the visualization of event data, an element in a pattern class is visualized with a square labeled with the corresponding concept; the arrows indicate the partial order relation of the elements.

Meanwhile, a pattern instance may consist of instances that are hardly related in practice. For example, the pattern instance pi_3 visualized in Figure 5 suggests that the lab tests are conducted based on the blood sampled one and a half years ago. Compared to pi_3, the other pattern instances, pi_1 and pi_2 in Figure 5, are more likely to be the pattern instances that one has in mind. Hence, to further identify a pattern instance with closely related instances, we characterize a pattern instance as a *local pattern instance* if the underlying instances are *related* based on the covering relation.

Definition 4 (Local Pattern Instance). *Let* pi 2 U_{inst} *be a pattern instance. We characterize* pi *as a local pattern instance iff* $\forall \mathsf{ci}_1, \mathsf{ci}_n$ 2 $\mathsf{pi}(\mathsf{ci}_1$ $\not=$ $\mathsf{ci}_n)$, $\exists \langle \mathsf{ci}_1, \mathsf{ci}_2, \ldots, \mathsf{ci}_n \rangle$, *where* $\forall 1 \quad i \quad n, \mathsf{ci}_i$ 2 pi *and* $\forall 1 \quad i<n, \mathsf{ci}_i \quad \mathsf{ci}_{i+1} \_ \mathsf{ci}_{i+1} \quad \mathsf{ci}_i$.

By iteratively applying the framework, we identify pattern instances in every iteration. Hence, a pattern instance consists of a set of instances that may be activity instances and/or pattern instances identified in the previous iteration.

4.3 Candidate Pattern Instances

Given a pattern class, first, we search for all the possible pattern instances of a pattern class in a log, which we name as *candidate pattern instances*. Since an instance is only related to one case, the search is performed for every case and the collection of the candidate pattern instances identified for every case in a log is the candidate pattern instances in the log.

Definition 5 (Candidate Pattern Instance Extraction). *Let* L *be a log and* PC *be a pattern class. Let* $\mathsf{c}_L = (\mathsf{CI}_c, \quad _c)$, *where* c 2 $\mathsf{CID}(L)$. *We define function* $\mathsf{EXT}_L : U_{cid} \quad U_{con} \; ! \; P(P(U_{inst}) \quad P(U_{inst} \quad U_{inst}))$, *where* $\mathsf{EXT}_L(c, \mathsf{PC}) = \{\mathsf{can} = (\mathsf{CI}, \quad , \pi_{con}) \mid \mathsf{CI} \quad \mathsf{CI}_c, \quad = \quad _c \setminus (\mathsf{CI} \quad \mathsf{CI}), \mathsf{can} \; ' \; \mathsf{PC}\}$ *denotes the candidate pattern instances of* PC *in* c.

The candidate pattern instances in a case are a set of partial orders of instances that are isomorphic to the pattern class. With the same example, the partial orders of activity instances visualized in Figure 5 are extracted as the candidate pattern instances.

4.4 Abstraction based on Patterns

We aggregate the underlying instances of a pattern instance to construct an abstracted log. The pattern instances are identified for every case in a log and the relation between an instance in a case and a pattern instance is inferred from the underlying instances of the pattern instance. First, we generalize the identification of pattern instances as follows.

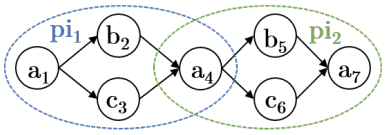
Definition 6 (Pattern Instance Identification). *Let L be a log and* CAN *be the candidate pattern instances in c 2* CID(L)*. We define the function* IDEN$_c$: $P(P(U_{inst}) \quad P(U_{inst} \quad U_{inst})) \; ! \; P(U_{inst}) \quad P(U_{inst} \quad U_{inst})$*, where* IDEN$_c$(CAN) = (PI,) *is a partial order of pattern instances* PI CAN *in c, where* 8pi$_1$, pi$_2$ *2* PI(pi$_1$ 6= pi$_2$), pi$_1$ pi$_2$ =) 9ci *2* pi$_1$8CI^0 *2* pi$_2$(ci ci^0).

The pattern instances in a case are a subset of the candidate pattern instances in the case. The identification can be initiated in a flexible manner while the partial order relation of the pattern instances identified must not violate the minimum requirement specified in Definition 6. Note that two different pattern instances may share some underlying instances.

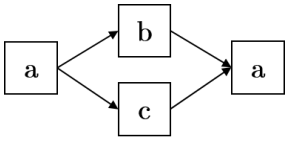
Figure 6 motivates the necessity of the flexible instantiation of the identification of pattern instances. Suppose that we extract two local pattern instances pi$_1$ and pi$_2$ as specified in Figure 6. By simply inferring the relation of the pattern instances based on the underlying instances, pi$_1$ pi$_2$ since they share a$_4$. However, assume that the a$_4$ represents a milestone achieved in a waterfall process; it is more reasonable to define the relation as pi$_1$ pi$_2$. Hence, we generalize the identification of pattern instances and allow one to impose the constraints that are applicable to the organization; nevertheless, a generic instantiation is also implemented and introduced in the next section.

Finally, the abstraction is realized by *aggregating* the pattern instance identified and defining the relation between the pattern instances and other instances.

Definition 7 (Abstraction). *Let $L = (CI, \; , \pi_{con}, \pi_{cid})$ be a log. Let* PI$_L$ = *f*pi *j c 2* CID(L): π^{p}_{cid}(pi) = *cg be the pattern instances in L;* $_p$= PI$_L$ PI$_L$ *denotes the partial order on* PI$_L$*. Given* CI$_{rst}$ = CI *n* $\bigcup_{pi2PI_L}$ pi *and* $_{rst}$ = \ (CI$_{rst}$ CI$_{rst}$)*, i.e., the partial order of instances that are not in any pattern instances. An abstracted log, $L^0 = (CI^0, \; ^0, \pi^0_{con}, \pi^0_{cid})$, is a log derived from L where*



(a) Visualization of the covering relation of a case and the pattern instances identified.



(b) A pattern class.

Fig. 6: A motivating example of pattern instance identification, where the identified pattern instances, pi$_1$ and pi$_2$, are annotated and pi$_1$ pi$_2$.

- $\mathsf{CI}^0 = \mathsf{CI}_{\mathsf{rst}} \cup \mathsf{PI}_L$ *is a set of instances;*
- $^0 = {}_{\mathsf{rst}} \cup {}_{\mathsf{p}} \cup (\mathsf{CI}_{\mathsf{rst}} \times \mathsf{PI}_L)$ *is a partial order on* CI^0*, where* $\forall \mathsf{ci} \in \mathsf{CI}_{\mathsf{rst}} \forall \mathsf{pi} \in \mathsf{PI}_L(\pi_{cid}(\mathsf{ci}) = \pi^{\mathsf{p}}_{cid}(\mathsf{pi}))$*,* $\mathsf{ci} \quad \mathsf{pi} \Leftrightarrow \neg(\mathsf{ci} \quad \mathsf{pi}) \wedge \forall \mathsf{ci}^0 \in \underline{\mathsf{pi}}(\mathsf{ci} \quad \mathsf{ci}^0)$ *and* $\mathsf{pi} \quad \mathsf{ci} \Leftrightarrow \neg(\mathsf{ci} \quad \mathsf{pi}) \wedge \forall \mathsf{ci}^0 \in \underline{\mathsf{pi}}(\mathsf{ci}^0 \quad \mathsf{ci})$*;*
- $\pi^0_{con} \colon \overline{\mathsf{CI}}^0 \to U_{con}$*, where* $\forall \mathsf{ci} \in \mathsf{CI}^0, \mathsf{ci} \in \mathsf{CI}_{\mathsf{rst}} \Rightarrow \pi^0_{con}(\mathsf{ci}) = \pi_{con}(\mathsf{ci}) \wedge \mathsf{ci} \in \mathsf{PI}_L \Rightarrow \pi^0_{con}(\mathsf{ci}) = \pi^{\mathsf{p}}_{con}(\mathsf{ci})$*;*
- $\pi^0_{cid} \colon \mathsf{CI}^0 \to U_{cid}$*, where* $\forall \mathsf{ci} \in \mathsf{CI}^0, \mathsf{ci} \in \mathsf{CI}_{\mathsf{rst}} \Rightarrow \pi^0_{cid}(\mathsf{ci}) = \pi_{cid}(ci) \wedge \mathsf{ci} \in \mathsf{PI}_L \Rightarrow \pi^0_{cid}(\mathsf{ci}) = \pi^{\mathsf{p}}_{cid}(\mathsf{ci})$*.*

We define the key artifacts, i.e., a pattern class and a pattern instance, in this section. We further impose constraints on the relation among the underlying instances of a pattern instance to identify pattern instances that could be more suitable under certain circumstances in practice. The proposed framework is introduced and illustrated with the running example described in Section 2.

5 Implementation

In this section, we present the implementation of the framework. We explain the implementation of candidate pattern instances extraction in Section 5.1, followed by the extraction of local pattern instances in Section 5.2. A generic method to identify pattern instances is introduced in Section 5.3.

5.1 Extracting Candidate Pattern Instances

To extract the candidate pattern instances in a case, we incrementally add instances to a partial order of instances in the case and check for isomorphism between the instances selected and a pattern class. Let $\mathsf{PC} = (X, \quad, l)$ be a pattern class and L be a log. Given a case $c \in \mathsf{CID}(L)$, $\mathsf{c}_L = (\mathsf{CI}_c, \quad_c)$. Given $\mathsf{CI} \quad \mathsf{CI}_c$ and $\quad_{\mathsf{po}} = \quad_c \setminus (\mathsf{CI} \quad \mathsf{CI})$, we let $\mathsf{po} = (\mathsf{CI}, \quad_{\mathsf{po}}, \pi_{con})$. We define the following functions:

- $\mathsf{INIT} \colon \mathcal{P}(U_{inst}) \quad \mathcal{P}(U_{inst} \quad U_{inst}) \to \mathcal{P}(\mathcal{P}(U_{inst}) \quad \mathcal{P}(U_{inst} \quad U_{inst}))$, where $\mathsf{INIT}(\mathsf{c}_L)$ initiates a set of partial orders of instances to be extended where $\forall(\mathsf{CI}^0, \quad ^0) \in \mathsf{INIT}(\mathsf{c}_L), \mathsf{CI}^0 \quad \mathsf{CI}_c$ and $^0 = \quad_c \setminus (\mathsf{CI}^0 \quad \mathsf{CI}^0)$.
- $\mathsf{SEL} \colon \mathcal{P}(U_{inst}) \quad \mathcal{P}(U_{inst} \quad U_{inst}) \quad \mathcal{P}(U_{inst}) \quad \mathcal{P}(U_{inst} \quad U_{inst}) \to \mathcal{P}(U_{inst})$, where $\mathsf{SEL}(\mathsf{po}, \mathsf{c}_L) \quad \mathsf{CI}_c$ selects a set of instances to be added into po where $\forall \mathsf{ci} \in \mathsf{SEL}(\mathsf{po}, \mathsf{c}_L), \mathsf{ci} \notin \mathsf{po}$.
- $\mathsf{CHECK} \colon \mathcal{P}(U_{inst}) \quad \mathcal{P}(\overline{U}_{inst} \quad U_{inst}) \quad U_{inst} \quad U_{con} \to \{true, false\}$, where $\mathsf{CHECK}(\mathsf{po}, \mathsf{ci}, \mathsf{PC})$ checks if adding $\mathsf{ci} \in \mathsf{CI}_c$ into po *may* form a candidate pattern instance of PC, i.e., let $\mathsf{po}^0 = \mathsf{ADD}(\mathsf{po}, \mathsf{ci}, \pi_{con})$, $\mathsf{CHECK}(\mathsf{po}, \mathsf{ci}, \mathsf{PC}) = true$ iff $\exists X^0 \quad \underline{\mathsf{PC}}(|X^0| \quad 1 \wedge \mathsf{po}^0 \ ' \ (X^0, \quad_{X'}, l))$, where $\quad_{X'} = \quad \setminus (X^0 \quad X^0)$; if $X^0 = \underline{\mathsf{PC}}$, po^0 forms a candidate pattern instance.

Algorithm 1 illustrates the implementation of $\mathsf{EXT}_L(c, \mathsf{PC})$ in Definition 5 with the three key functions defined. The algorithm initiates a set of partial orders of instances and incrementally adds other instances. If there are no instances

to be added such that it *may* form a candidate pattern instance of the input pattern class in the later iteration, we discard the partial order of instances. Otherwise, we check if the instances form a candidate pattern instance. If so, the instances form a candidate pattern instance, or the partial order of instances is added back to the open items to be checked in the next iteration.

A (labeled) partial order can be easily converted into a (labeled) directed acyclic graph (DAG). By representing a partial order of instances and a pattern class as labeled DAGs, we apply *graph edit distance* [17], i.e., a measure of similarity between two graphs that searches for the minimal cost of graph operations to make one graph isomorphic to the other, to check for the isomorphism between two partial orders.

Algorithm 1 Candidate Pattern Instance Extraction

Input: case $c \in \mathsf{CID}(L)$, a pattern class $\mathsf{PC} \in \mathcal{U}_{con}$
Output: a collection of candidate pattern instances of PC in c, i.e., *candidates*

$\mathsf{c}_L = (\mathsf{CI}_c, \prec_c)$
open $\leftarrow \mathsf{INIT}(\mathsf{c}_L)$ ▷ a set of partial orders of instances
candidates $\leftarrow \{\}$ ▷ an empty set to collect candidate pattern instances
while *open* **do**
  $\mathsf{po} \leftarrow$ *open.pop*()
  $\mathsf{CI}^{\theta} \leftarrow \mathsf{SEL}(\mathsf{po}, \mathsf{c}_L)$ ▷ instances to add incrementally
  for $\mathsf{ci} \in \mathsf{CI}^{\theta}$ **do**
    if $\mathsf{CHECK}(\mathsf{po}, \mathsf{ci}, \mathsf{PC})$ **then** ▷ if po can be a potential candidate
      ▷ pattern instance of PC by adding ci
      $\mathsf{po}^{\theta} \leftarrow \mathsf{ADD}(\mathsf{po}, \mathsf{ci}, \pi_{con})$
      if $\mathsf{po}^{\theta} \simeq \mathsf{PC}$ **then**
        candidates.add(po^{θ})
      else
        open.add(po^{θ}) ▷ add po^{θ} back to *open* for the next iteration
      end if
    end if
  end for
end while

Optimization. We optimize the implementation by reducing the search space while ensuring that the relation among the instances is not altered. The optimization of the algorithm can be easily achieved by directly projecting the relevant instances in a case, i.e., the instance of the underlying concepts of a pattern class, since the relation among the instances remains after the projection.

5.2 Extracting Local Pattern Instances

The local pattern instances may be extracted by selecting from the pattern instances identified. Alternatively, we search for candidate pattern instances with

the property of local pattern instances. The search of local pattern instances may be implemented in a similar way as described in Algorithm 1, however, with the isomorphism check based on the covering relation of partial orders. The check is realized by representing the covering relation of a partial order with a DAG, which is the *transitive reduction* of the DAG representing the partial order.

Nevertheless, the optimization of the search cannot be performed in the same way as the projection of relevant instances may result in missing relations in the graph representing the covering relation of a partial order. Hence, we must ensure the connectivity of the covering relation of a partial order of instances. Let $\mathsf{PC} = (X, \quad, l)$ be a pattern class. Given start activities $\mathsf{SA} = \mathit{fl}(x)\ \mathit{j}\ x\ \mathit{2}\ X, \mathit{@}x^{\mathit{0}}\ \mathit{2}\ X(x^{\mathit{0}} \quad x)\mathit{g}$, we add an artificial start node and the relation from the start node to every node labeled with activity in SA to the graph representing a case. We remove a node if there does not exist an *undirected* path from the start node to the node. Then, the projection of relevant instances may be performed and we optimize the search with the bread-first search strategy.

5.3 Identification of Pattern Instances

One may impose semantic constraints on IDEN_c to identify pattern instances. Alternatively, for a generic application of the framework, we implement IDEN_c by introducing an overlapping threshold $t\ \mathit{2}\ \mathsf{R}$, where $0 \quad t < 1$. Let L be a log and CAN denote the candidate pattern instances in a case $c\ \mathit{2}\ \mathsf{CID}(L)$. For any $\mathsf{pi}_1, \mathsf{pi}_1\ \mathit{2}\ \underline{\mathsf{IDEN}_c(\mathsf{CAN})}$, $\mathit{j}\underline{\mathsf{pi}_1} \setminus \underline{\mathsf{pi}_2}\mathit{j}/\mathit{j}\underline{\mathsf{pi}_1} \mathit{[}\ \underline{\mathsf{pi}_2}\mathit{j} \quad t$; if $t = 0$, a set of disjoint candidate pattern instances are identified as pattern instances. We determine the relation between pi_1 and pi_2 based on the majority relation of the non-shared underlying instances, i.e., given $\mathsf{CI}_1 = \underline{\mathsf{pi}_1}\ \mathit{n}\ \underline{\mathsf{pi}_2}$ and $\mathsf{CI}_2 = \underline{\mathsf{pi}_2}\ \mathit{n}\ \underline{\mathsf{pi}_1}$, pi_1 $\mathsf{pi}_2 \quad \mathit{(\ )} \quad \mathit{jf}\mathsf{ci}_1\ \mathit{2}\ \mathsf{CI}_1\ \mathit{j}\ \mathsf{ci}_2\ \mathit{2}\ \mathsf{CI}_2(\mathsf{ci}_1 \quad \mathsf{ci}_2)\mathit{gj}/\mathit{j}\mathsf{CI}_1\mathit{j} > 0.5$; note that we assume that $\pi^{\mathsf{p}}_{con}(\mathsf{pi}_1) = \pi^{\mathsf{p}}_{con}(\mathsf{pi}_2)$ s.t. $\mathit{j}\mathsf{CI}_1\mathit{j} = \mathit{j}\mathsf{CI}_2\mathit{j}$. The instantiation is non-deterministic.

The abstraction, based on the given pattern instances and their relation, is straightforward to implement, following the guidelines outlined in Definition 7. As a result, an abstracted log is computed based on the pattern class, which serves as the input for the subsequent iteration of another pattern class.

6 Experiments

In this section, we present the application of the proposed framework with a synthetic log and conduct a case study based on a real-life log [19].

6.1 Evaluation

We construct a partially ordered log containing $5,000$ cases based on the process in Figure 7, from which Table 1 is extracted. We evaluate the proposed framework in three aspects: the number of pattern instances selected, the performance with optimization applied, and the quality metrics of the process models discovered.

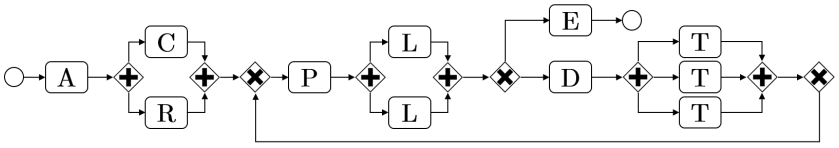
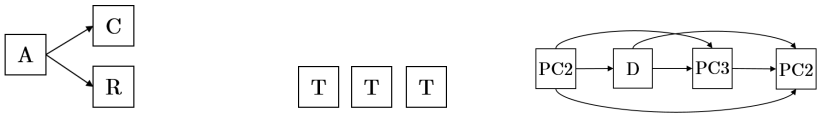


Fig. 7: A process model used for generating the synthetic log. The model is represented with BPMN [1]. The labels correspond to the abbreviation of activity labels in Table 1.

Defining Pattern Classes. We define four pattern classes. Let the pattern class visualized in Figure 2 be pattern class PC2; other pattern classes are visualized in Figure 8. We identify local pattern instances for PC1 (Figure 8a) and PC2 (Figure 2). The pattern class PC3 defined in Figure 8b consists of three concurrent Provide Treatment. The pattern class PC4 is defined by other pattern classes and activity `Diagnose` (`D`). As the framework abstracts a log based on one pattern class at a time, the numbers indicate the corresponding iteration.

Experimental Setup. Inspired by [12], we evaluate the impact of noises on the proposed framework. We inject $n\%$, where $n \in \{10, 20, 30\}$, of noises to the log by injecting $n\%$ of noises to every case as illustrated in Algorithm 2. We randomly select $n\%$ of instances in a case and swap the relation with one of the directly succeeding instance for every instance selected, i.e., given an instance ci, we select an instance ci'($ci \neq ci'$), where ci $\neq$ ci', and swap the relation. For discovering the process models, we applied Inductive Miner - Infrequent with a noise threshold of 0.2 [8].

Figure 9 shows the number of pattern instances identified for every pattern class. Since the first three pattern classes do not share common activity labels, the number of pattern instances extracted is independent; however, the number of pattern instances of Treatment Process (PC4) dramatically decreases since the pattern class is defined based on other pattern classes; hence, the number of pattern instances abstracted is also limited to the number of pattern instances of other pattern classes. The number of pattern instances identified for PC2 is much higher than other pattern classes since the pattern class can be conducted several times in a case. In addition, with the percentage of noise injected increasing, the



(a) Basic Consultation (PC1) (b) Treatment (PC3) (c) Treatment Process (PC4)

Fig. 8: Pattern classes defined for synthetic log. Abbreviated labels are provided for simplicity. Note that PC4 is defined based on PC2, PC3, and `Diagnose`.

Algorithm 2 Noise Injection of Case

Input: case $c \in \mathsf{CID}(L)$, a noise percentage $n \in \{10\%, 20\%, 30\%\}$
Output: case with noise $\mathsf{c}^\theta = (\mathsf{CI}^\theta, \prec^\theta)$
1: $\mathsf{c}^\theta = (\mathsf{CI}_c, \prec_c)$ ▷ initiate a partial order of instances in case c
2: $cnt \leftarrow floor(|\mathsf{CI}_c| \times n)$ ▷ number of pairs of instances to swap
3: $\mathsf{CI} \leftarrow$ randomly select cnt instances ▷ $\mathsf{CI} \subseteq \mathsf{CI}_c$, where $|\mathsf{CI}| = cnt$
4: **for** $\mathsf{ci} \in \mathsf{CI}$ **do**
5: $\mathsf{ci}^\theta \leftarrow$ SelectFollowingNeighbor$(\mathsf{ci}, \mathsf{c}_L)$ ▷ $\mathsf{ci} \prec \mathsf{ci}^\theta$
6: $\mathsf{c}^\theta \leftarrow$ Swap$(\mathsf{ci}, \mathsf{ci}^\theta)$
7: **end for**

number of pattern instances identified decreases because the noise alters the relation of the instances in the log.

Figure 10 presents the average time required to abstract a case based on the pattern classes defined. Regardless of the optimization, abstracting a case based on a pattern class requires less than 1 second. With optimization, the runtime is further reduced to 4 times faster; for the pattern class of `Basic Consultation` (PC1), the optimization even results in 16 times faster. Meanwhile, we observe that the abstraction based on the pattern class `Treatment` (PC3) is much faster than other abstractions since the checking for isomorphism is much faster as the graph representing the pattern class contains only isolated nodes labeled with `Provide Treatment` and no relation, represented with edges in a graph, needs to be examined.

Figure 11 reports the quality metrics of the process models discovered based on the abstracted logs. We see that the noise has a negative impact on the fitness as shown in Figure 11a. Except for the abstraction based on PC4 without noise injection, the fitness is correlated with the number of pattern instances abstracted as shown in Figure 9. Compared to fitness, the noise has less impact on the precision as shown in Figure 11b. Interestingly, with more noise injected, which causes fewer pattern instances abstracted, the precision increases. By analyzing the conformance details, we infer that it is due to the unmatched instances that are labeled with the underlying concepts of a pattern class, which impacts the number of instances in a case and further influences the metrics. The har-

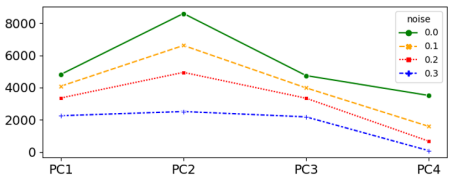


Fig. 9: Number of pattern instances abstracted per pattern class with different noise injection.

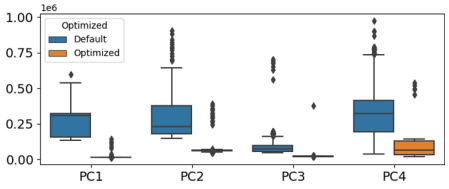
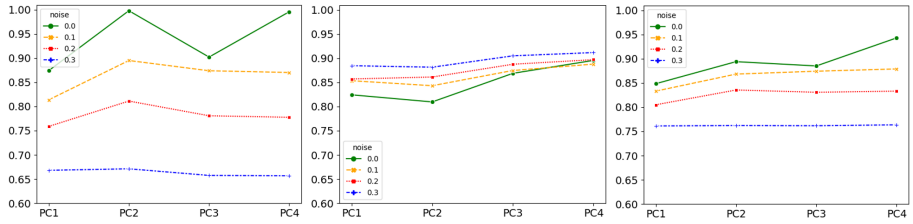


Fig. 10: Average abstraction time in microsecond per case based on the pattern classes defined.



(a) Fitness (b) Precision (c) F-measure

Fig. 11: Quality metrics of process models discovered using abstracted logs. The ranges of the y-axis of the plots are uniformly set from 0.6 to 1.0 for comparison.

monic mean of the fitness and the precision, F-measure, as shown in Figure 11c enhances due to the increase in fitness.

6.2 Case Study

To demonstrate the proposed framework in practice, we apply the framework to a real-life log [19]. We preprocess the log to construct a partially ordered log based on timestamps. We pair every start record with exactly one complete record based on their order to construct an activity instance. The pattern classes are defined by sets of activities indicating a successful operation. Figure 12 shows one of the pattern classes, indicating a successful application.[2]

Figure 14 presents an excerpt of the analysis. The figure shows the behavior of the process with the relative frequency projected on the visualization. The analysis shows that, since the pattern classes are defined by sets of successful operation occurring sequentially in a process, as the process continues, the percentage of successful operations also decreases. The percentage of successful end-to-end operation, i.e., PC4, depends on the last successful operations, i.e., PC3.

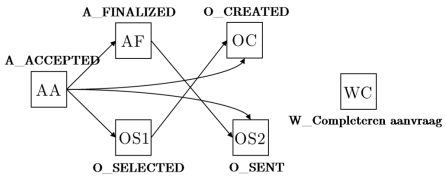


Fig. 12: Successful Application (PC1), where WC is expected to be executed concurrently in time with activities.

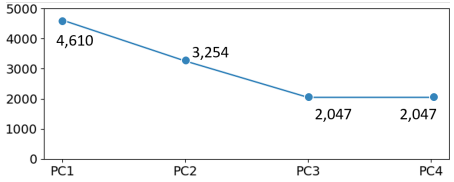


Fig. 13: Number of pattern instances abstracted per pattern class defined for case study.

[2] We define PC2 as concurrent {O_SENT_BACK, W_Nabellen offertes}, PC3 as concurrent {O_ACCEPTED, A_APPROVED, A_REGISTERED, A_ACTIVATED, W_Valideren aanvraag}, and PC4 as a sequence of PC1, PC2, and PC3 (considering that a total order is also a partial order).

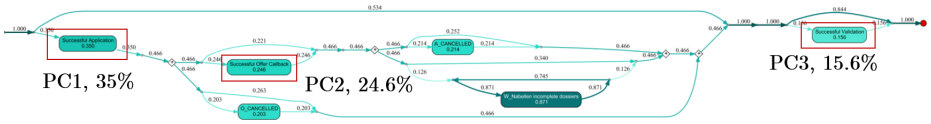


Fig. 14: An excerpt of the analysis based on an abstracted log. The visualization is based on IMflc [9] and the shade of color shows the relative importance based on the number of instances identified on the path. We highlight the defined pattern classes.

The analysis is further supported by the number of pattern instances abstracted based on the pattern classes defined in Figure 13. Meanwhile, with abstraction, we can easier identify and further investigate the unsuccessful executions, e.g., other activities shown in the excerpt. In addition, we apply the abstraction with the technique proposed in [14], where we model the undefined relations in the pattern classes as parallelism in the representation of the regularity defined in the work. With the same sets of activities in the regularity defined as the pattern classes, the technique does not identify any instances.

7 Related Work

This section discusses the research in event abstraction in the field of process mining. Numerous event abstraction techniques focus on identifying regularity in event data to group activities or records of activities. The authors in [18] apply a statistical model to *predict* the class of a record at a higher level. Nguyen et al. decompose a process into sets of activities by exploiting the modularity metrics based on a graph constructed from event data [15]. In [11], the authors apply clustering based on the features encoded from fragments of a sequence of event data, which are seen as an instance at the higher level and are provided with domain knowledge. The work focuses on identifying the regularity from sequential event data with strong assumptions on a process, e.g., sensor data and milestone existence; the identification of the instances at the higher level is rather straightforward or ignored due to the assumption of classical event data.

To facilitate analyzing event data at a higher level, it is important that, not only concepts at the higher level, an instance of a concept at the higher level is also identified such that one may apply abstraction iteratively based on their needs. Some work applies the hierarchy of concepts to construct a hierarchy of abstractions [10,13]. In Lu et al. [13], an instance at the higher level is extracted by projecting the relevant records. Leemans et al. apply the discovering techniques to discover groups of process models at different levels and compose them to construct a complete model at the specified level [10]; the extraction of an instance is achieved with alignment [3]. The alignment is also exploited to identify instances of the regularity identified in [14]. In Bose and van der Aalst [5], the authors discover frequent local execution regularity and abstract accordingly.

The techniques discussed consider the iterative applicability; however, except for simply projecting the records in [13], the extraction of the instances at the higher level tends to be limited to local regularity.

This paper explicitly considers partially ordered event data and focuses on the extraction of the pattern instances. For discovering the regularity, since a pattern class and a case can be represented as labeled DAGs, we argue that existing techniques for frequent graph pattern mining may be exploited [16]. With a pattern class defined based on partial order relation, the extraction achieves beyond local regularity and the reliability of the abstraction can be enhanced with the support of human analysts by providing a more comprehensible representation, i.e., the regularity that can be directly mapped to the behavior of the execution of concepts, e.g., activities, observed in real-life.

8 Conclusion

Motivated by the applicability of event abstraction in practice, we present and define a framework for identifying pattern instances based on partially ordered event data, which reflect the behavior of the activities performed in real-life. The framework abstracts a log based on a pattern class by extracting candidate pattern instances and identifying pattern instances of the pattern class. We implemented the framework and conducted experiments by constructing a hierarchy of abstractions based on a synthetic and a real-life log. The experiments demonstrate the impact of noises and how one can obtain insights from the analysis based on abstracted logs. The framework is also applicable to classical event data since a total order is also a partial order. For future work, our objectives are two-fold. First, we aim to further support in defining the order of pattern classes as specified by domain experts. Second, we plan to extend the extraction to identify unexpected behavior in partially ordered event data, while explicitly considering the causal relation of the instances.

References

1. Business process model and notation (bpmn) version 2.0. Object Management Group (2011)
2. van der Aalst, W.M.P.: Process Mining - Data Science in Action, Second Edition. Springer (2016)
3. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying history on process models for conformance checking and performance analysis. WIREs Data Mining Knowl. Discov. **2**(2), 182–192 (2012)
4. Augusto, A., Conforti, R., Dumas, M., Rosa, M.L., Maggi, F.M., Marrella, A., Mecella, M., Soo, A.: Automated discovery of process models from event logs: Review and benchmark. IEEE Trans. Knowl. Data Eng. **31**(4), 686–705 (2019)
5. Bose, R.P.J.C., van der Aalst, W.M.P.: Abstractions in process mining: A taxonomy of patterns. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) Business Process Management, 7th International Conference, BPM 2009, Ulm, Germany, September 8-10, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5701, pp. 159–175. Springer (2009)

6. Carmona, J., van Dongen, B.F., Solti, A., Weidlich, M.: Conformance Checking - Relating Processes and Models. Springer (2018)
7. Hornix, P.T.G.: Performance analysis of business processes through process mining. Master's Thesis, Eindhoven University of Technology (2007)
8. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs containing infrequent behaviour. In: Lohmann, N., Song, M., Wohed, P. (eds.) Business Process Management Workshops - BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers. Lecture Notes in Business Information Processing, vol. 171, pp. 66–78. Springer (2013)
9. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Using life cycle information in process discovery. In: Reichert, M., Reijers, H.A. (eds.) Business Process Management Workshops - BPM 2015, 13th International Workshops, Innsbruck, Austria, August 31 - September 3, 2015, Revised Papers. Lecture Notes in Business Information Processing, vol. 256, pp. 204–217. Springer (2015)
10. Leemans, S.J.J., Goel, K., van Zelst, S.J.: Using multi-level information in hierarchical process mining: Balancing behavioural quality and model complexity. In: van Dongen, B.F., Montali, M., Wynn, M.T. (eds.) 2nd International Conference on Process Mining, ICPM 2020, Padua, Italy, October 4-9, 2020. pp. 137–144. IEEE (2020)
11. de Leoni, M., Dündar, S.: Event-log abstraction using batch session identification and clustering. In: Hung, C., Cerný, T., Shin, D., Bechini, A. (eds.) SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020. pp. 36–44. ACM (2020)
12. de Leoni, M., Marrella, A.: Aligning real process executions and prescriptive process models through automated planning. Expert Syst. Appl. **82**, 162–183 (2017)
13. Lu, X., Gal, A., Reijers, H.A.: Discovering hierarchical processes using flexible activity trees for event abstraction. In: van Dongen, B.F., Montali, M., Wynn, M.T. (eds.) 2nd International Conference on Process Mining, ICPM 2020, Padua, Italy, October 4-9, 2020. pp. 145–152. IEEE (2020)
14. Mannhardt, F., de Leoni, M., Reijers, H.A., van der Aalst, W.M.P., Toussaint, P.J.: Guided process discovery - A pattern-based approach. Inf. Syst. **76**, 1–18 (2018)
15. Nguyen, H., Dumas, M., ter Hofstede, A.H.M., Rosa, M.L., Maggi, F.M.: Stage-based discovery of business process models from event logs. Inf. Syst. **84**, 214–237 (2019)
16. Nguyen, L.B.Q., Zelinka, I., Snásel, V., Nguyen, L.T.T., Vo, B.: Subgraph mining in a large graph: A review. WIREs Data Mining Knowl. Discov. **12**(4) (2022)
17. Sanfeliu, A., Fu, K.: A distance measure between attributed relational graphs for pattern recognition. IEEE Trans. Syst. Man Cybern. **13**(3), 353–362 (1983)
18. Tax, N., Sidorova, N., Haakma, R., van der Aalst, W.M.P.: Event abstraction for process mining using supervised learning techniques. In: Bi, Y., Kapoor, S., Bhatia, R. (eds.) Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016 - Volume 1, London, UK, 21-22 September 2016. Lecture Notes in Networks and Systems, vol. 15, pp. 251–269. Springer (2016)
19. van Dongen, B.F.: BPI challenge 2012 (2012), `https://data.4tu.nl/articles/dataset/BPI_Challenge_2012/12689204`
20. Yasmin, F.A., Bukhsh, F.A., de Alencar Silva, P.: Process enhancement in process mining: A literature review. In: Ceravolo, P., López, M.T.G., van Keulen, M. (eds.) Proceedings of the 8th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2018), Seville, Spain, December 13-14, 2018. CEUR Workshop Proceedings, vol. 2270, pp. 65–72. CEUR-WS.org (2018)