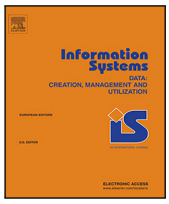
Information Systems 144 (2027) 102808



Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is



Recursive object-centric process discovery with fitness and rediscovery guarantees

Jan Niklas van Detten [a,b,*], Pol Schumacher [b], Sander J.J. Leemans [a]

[a] RWTH Aachen University, Ahornstraße 55, Aachen, 52074, Germany
[b] Celonis, Theresienstraße 6, München, 80333, Germany

ARTICLE INFO

Keywords:
Process mining
Process discovery
Object-centric
Guarantees

ABSTRACT

Digitized processes are an integral part of businesses to achieve their operational goals. Process mining aims to analyze such processes based on digital track records. These so-called event logs can be extracted from commonly used information systems. Traditionally, process mining remains limited as it requires individual business objects to define isolated process executions. However, in real-life processes, it is much more common to observe multiple business objects of various types that interact with each other. Object-centric process mining acknowledges this fact and fundamentally enhances traditional process mining techniques to analyze processes with interacting business objects. This analysis often requires a process model that describes the interacting control flows of all objects. Discovery algorithms are used to automatically construct such models from logs. For further steps, it is important that the discovery algorithm *rediscovers* a process model that is equivalent to the underlying business process. If it is impossible to rediscover the full behavior of a process, the discovered model should at least be *fitting* for the observed behavior. However, no existing object-centric discovery technique guarantees rediscovery or fitness. In this paper, we introduce a new algorithm that recursively constructs tree-structured object-centric models. We prove that our approach produces fitting models for all logs and rediscovers a subclass of processes within the used representational bias from logs that are sufficiently complete and free of incorrect recordings, unobserved or duplicated activities and loops. We evaluate our approach on a range of public logs and find it to produce fitting models in feasible runtime.

1. Introduction

Modern businesses often achieve their operational goals by means of digitized processes. Typically, the execution of such processes leaves distinct traces in corresponding corporate information systems. These records provide a solid data-based foundation to analyze and optimize core business processes. This analysis is often of dire necessity to ensure appropriate resource usage, the adherence to legal requirements and ultimately profitability.

Process mining has emerged as research field to provide conceptual frameworks and algorithmic techniques for this exact purpose [1]. It encompasses the extraction of process-related data from enterprise systems into standardized records, called event logs, and the subsequent analysis thereof. As such it provides the necessary tools to align processes with operational goals and has attracted correspondingly high interest in the industrial settings.

However, process mining traditionally suffers from a core limitation that subjects every stage of the extraction and analysis pipeline to a certain bias. That is, it requires the specification of individual core business objects, that each define their own isolated execution of the process. The non-trivial choice of what constitutes such objects in any given domain is already made during extraction, and severely influences the results of any subsequent analysis.

Unfortunately, this restrictions maps poorly onto real-life processes, which rarely contain such clearly separable and isolated business objects. Instead, they typically consist of multiple objects of various types, that interact and depend on each other. An order, for example, often refers to multiple production items and leads to a corresponding invoice. Object-centric process mining emerged to provide techniques that account for such processes [2].

The analysis of object-centric event logs often requires the construction of a process model that depicts the control flows of all objects and the interactions between them. These models can be automatically constructed by object-centric discovery algorithms [3,4]. Based on such models, various analysis techniques can subsequently be applied to detect unintended behavior. These techniques often enrich the control flow perspective of the model with supplementary information, such as

* Corresponding author at: RWTH Aachen University, Ahornstraße 55, Aachen, 52074, Germany.
E-mail address: n.vandetten@bpm.rwth-aachen.de (J.N. van Detten).

https://doi.org/10.1016/j.is.2026.102808
Received 30 July 2025; Received in revised form 29 May 2026; Accepted 12 September 2026
Available online 5 October 2026
0306-4379/© 2026 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).
Open access funding enabled and organized by Projekt DEAL.

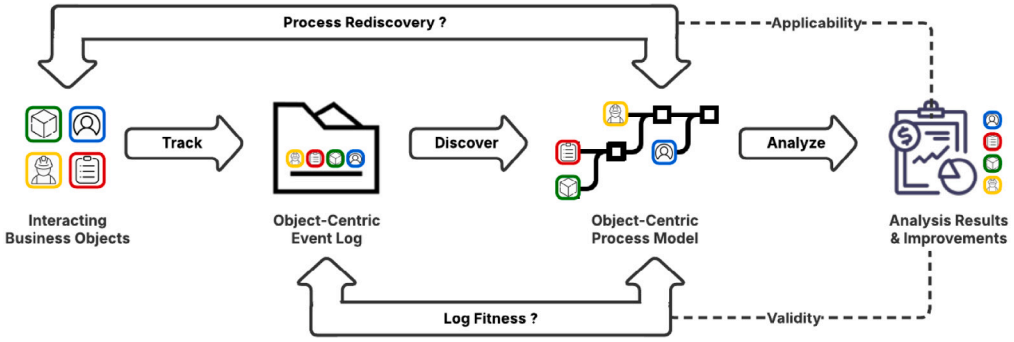


Fig. 1. Object-centric analysis pipeline and the guarantees of rediscovery and fitness to ensure the validity and applicability of the analysis results to the underlying process.

the time needed for each step, to identify optimization potential in the underlying business process [5].

However, the validity of such an analysis depends heavily on how well the discovered process model describes the investigated object-centric log and the business process that generated it [6]. A mismatch between the model and the log, or the model and the underlying business process, can induce flawed analytical results. In a real-life application of object-centric process mining, this can in return lead to questionable action recommendations with negative consequences. Therefore, discovery algorithms should guarantee that such a mismatch, given enough valid information in the input log, cannot occur.

Currently, practitioners are facing a problem with existing approaches in that regard. If the resulting model does not turn out as expected despite sufficient data being available, it is unclear if this is caused by algorithmic failure, or the process actually behaving differently. This mismatch between the model and the process induces uncertainty on further analysis steps.

Potential mismatches between the model and the log have even more severe consequences, as they straight up impede the subsequent application of certain analysis techniques. For example, a common visual strategy for performance analysis includes replaying the log on the resulting model, making time-based performance problems incredibly obvious. However this assumes that the log can be replayed on the model without mismatches between them. Fixing such mismatches to approximate the closest possible model behavior to the log is known to suffer from exponential runtime complexity [7].

As such, an approach with formal guarantees on the absence of such mismatches is needed to allow that algorithmic failure can be ruled out in case of unexpected process models and to ensure the subsequent applicability of replay-based analysis techniques. In that context, observed behavior that is not represented in the model is referred to as a lack of fitness. Reversely, additional behavior included in the model that cannot be observed in the input log is denoted as a lack of precision. A fitting model is able to replay the entire input log, offering a validated basis to analyze the respective control flow in a replay. A rediscovered model is behaviorally equivalent to the underlying business process of the input log, ensuring that all analysis results are applicable. Fig. 1 shows the most common analysis pipeline for object-centric processes and the important role of rediscovery and fitness.

To the best of our knowledge, no existing object-centric discovery technique provides any fitness or rediscovery guarantee. As a result, practitioners do not have any algorithmic means to reliably construct object-centric models that represent the behavior of a given log or its underlying business process.

In this paper, we introduce a new discovery algorithm to fill that gap. We utilize the modeling formalism of object-centric process trees [3], enabling a recursive construction of the models and an inductive proof of their guarantees. We prove that our approach produces fitting models and rediscovers processes that can be represented as a subclass of object-centric process trees from logs that include sufficient, and correctly recorded, behavior. Specifically, the subclass of trees that is free of unobserved or duplicated activity labels, fully optional sub processes and loops, since individual iterations might not be bound to a control flow by at least one object type.

Our paper makes the following contributions (1) a new discovery technique for object-centric process trees (2) a proof of all input logs to fit the discovered model (3) a proof of the guaranteed rediscovery of a subclass of object-centric process trees from sufficiently complete logs free of incorrect recordings. Conceptually, our contribution lifts the field of object-centric process discovery with formal guarantees from a purely technical discussion to an algorithmic framework with practically applicable instantiations that can be extended upon in the future. Specifically, it provides a foundation to extend further corresponding discovery approaches into the object-centric setting [8].

We evaluate our approach on a range of object-centric event logs with regards to its feasibility, the applicability of the utilized representational bias, and the fitness and precision of the resulting models in comparison to existing discovery techniques. Our results indicate that our approach is computationally feasible. We found the precision decreases at those logs that do not perfectly fit into the utilized representational bias to be competitive.

The remainder of this paper is structured as follows: Section 2 provides required background information on object-centric event logs and object-centric process trees. Section 3 introduces our new discovery techniques. The provided formal guarantees are discussed in Section 4. Our evaluation setups and results are described and discussed in Section 5. Section 6 summarizes related work. Section 7 summarizes our work and further research.

2. Preliminaries

In this section we formalize object-centric event logs and object-centric process trees. We write $\{\dots\}$, $[\dots]$ and $\langle\dots\rangle$ for sets, multisets and sequences respectively and include zero in $\mathbb{N}$ if not stated otherwise. Additionally, we write $\langle\dots\rangle + \langle\dots\rangle$ for the concatenation of two or more sequences. Similarly, we write $\langle\dots\rangle \sqcup\!\sqcup \langle\dots\rangle$ for arbitrary interleavings of sequences.

2.1. Object-centric event logs

Object-centric event logs track the behavior of interacting objects [9]. They contain the executions of activities, called events, with the involved object sets. Additionally, each object is assigned a unique object type.

Let $\mathbb{U}_{obj}, \mathbb{U}_{typ}, \mathbb{U}_{act}$ be the universes of objects, types and activities. Then, an object-centric event log is a sequence of events $\langle (a_1, O_1) \dots (a_n, O_n) \rangle$ with activities $a_i \in \mathbb{U}_{act}$ and object sets $O_i \subset \mathbb{U}_{obj}$. Each object is linked to its type with the mapping function $\omega : \mathbb{U}_{obj} \mapsto \mathbb{U}_{typ}$. We write $\Theta \subset \mathbb{U}_{obj}$ for all objects in a log and $\Omega \subset \mathbb{U}_{typ}$ for all contained

Table 1
Part 1 — Object-centric log with customers (c), orders (o) and items (i). The temporal execution order of the events is row-by-row, as indicated by the event number.

Event #	Activity	Customers	Orders	Items
1	IDENTIFY	c_1		
2	REJECT	c_1		
3	IDENTIFY	c_1		
4	PLACE	c_1	o_1	i_1, i_2
5	PLACE	c_1	o_2	i_3, i_4
6	PRODUCE	c_1	o_1	i_1
7	PRODUCE	c_1	o_1	i_2
8	PAY	c_1	o_1	i_1, i_2
9	PAY	c_1	o_2	i_3, i_4
10	PRODUCE		o_2	i_3
11	PRODUCE		o_2	i_4
12	PLACE	c_1	o_3	i_5, i_6
13	PRODUCE		o_3	i_6
14	PAY	c_1	o_3	i_5, i_6
15	PRODUCE	c_1	o_3	i_5
16	PLACE	c_1	o_4	i_7, i_8

Table 2
Part 2 — Object-centric log with customers (c), orders (o) and items (i). The temporal execution order of the events is row-by-row, as indicated by the event number.

Event #	Activity	Customers	Orders	Items
17	IDENTIFY	c_2		
18	PRODUCE		o_4	i_7
19	PRODUCE		o_4	i_8
20	PLACE	c_2	o_5	i_9
21	PAY	c_1	o_4	i_7, i_8
22	STORE		o_1	i_1
23	STORE		o_1	i_2
24	PAY	c_2	o_5	i_9
25	STORE		o_2	i_3
26	SEND		o_2	i_4
27	SEND		o_3	i_5
28	PRODUCE	c_2	o_5	i_9
29	STORE		o_3	i_6
30	SEND		o_4	i_8
30	SEND		o_4	i_7
30	SEND		o_5	i_9

object types. The set of activities in a log is denoted as $\Sigma \subset \mathbb{U}_{act}$, the so-called activity alphabet. We write $L\restriction_{\Sigma'}$ to filter a given log L onto the events with activities in Σ'.

Tables 1 and 2 show an example object-centric event log. It describes an order management process that involves customers, orders and items. Customers need to provide identification credentials, which might be rejected. As soon as valid credentials are provided, orders for potentially multiple items can be placed by the customer. The orders are subsequently produced and payed. In case of customized items, the customer is involved in the production step. Finally, each item is either send out or stored for pick up.

2.2. Interaction patterns

We use four patterns to describe the relation between individual object types $ot \in \Omega$ and activities $a \in \Sigma$. We consider an object type and an activity as *related* if there is an event with that activity and any object of that type:

$$ot \in rel_a \iff \exists (a_i, O_i) \in L : a_i = a \wedge \exists o \in O_i : \omega(o) = ot$$

Furthermore, we differentiate the relation between an object type and an activity based on the multiplicities of the respective objects in events. The relation is referred to as *convergent* if there is an event with the activity and multiple objects of the object type in the object-centric event log:

$$ot \in con_a \iff ot \in rel_a \wedge \exists (a_i, O_i) \in L : a_i = a \wedge |\{o \in O_i \mid \omega(o) = ot\}| > 1$$

In the example log, multiple items can be ordered at once, causing convergence. Similarly, an object type and an activity are called *deficient* if there is an event with the activity that does not involve any objects of the type:

$$ot \in def_a \iff ot \in rel_a \wedge \exists (a_i, O_i) \in L : a_i = a \wedge |\{o \in O_i \mid \omega(o) = ot\}| < 1$$

In our running example, each item can be produced with or without the involvement of the customer, causing deficiency. Note that both, convergence and deficiency, refer to the multiplicity of objects of the respective type for a single event with the activity. Reversely, we also consider how many events there are with the activity for a single object of the type. We refer to the relation between an object type and an activity as *divergent* if any object of that type is involved in multiple or no events with the activity:

$$ot \in div_a \iff \exists o \in \Theta : \omega(o) = ot \wedge |\{i \mid (a_i, O_i) \in L \wedge a_i = a \wedge o \in O_i\}| \neq 1$$

For example, each customer can place many independent orders, resulting in divergence. Note that divergence typically indicates an object type to exhibit resource-like or multi-instance-based behavior in relation to the respective activity. That is, since it is involved in multiple objects of another type, there is seemingly random behavior from the perspective of the divergent type since the objects of the secondary type behave independently.

2.3. Graph abstractions

We use auxiliary constructs to describe the behavior of individual object and types. For an object, we use the *object trace* which is the sequence of activities from those events in which the object is involved in. Formally, for a log L and an object $o \in \Theta$ we write $L\restriction_o = \langle a_1 \dots a_n \mid (a_i, O_i) \in L \wedge o \in O_i \rangle$. For individual object types, we aggregate the object traces of all objects of that type into a relation to summarize in which order activities are executed. The so-called directly-follows graph for an object type contains nodes for all related activities and relations between them if they follow each other in any object trace. Additionally, the graph contains special start and end nodes for those activities that occur in the first or last place in any object trace.

Formally, the directly-follows graph for an object type $ot \in \Omega$ and an object-centric event log L is a graph with nodes in $\{a \mid ot \in rel_a\}$ and edges $a \twoheadrightarrow_{L,ot} b$ between a, b if $\exists o \in \Theta : \omega(o) = ot \wedge L\restriction_o = \langle \dots a, b, \dots \rangle$. We write $a \twoheadrightarrow^{+}_{L,ot} b$ for the edges in the transitive closure of that relation. Additionally, the set of start and end activities in the directly-follows graph are defined as the first and last activities of object traces per object type:

$$start_{ot}(L) = \{a \in rel_a \mid \exists o \in \Theta : \omega(o) = ot \wedge L\restriction_o = \langle a, \dots \rangle\}$$
$$end_{ot}(L) = \{a \in rel_a \mid \exists o \in \Theta : \omega(o) = ot \wedge L\restriction_o = \langle \dots, a \rangle\}$$

For a set of logs $\mathbb{L}$, we define the cumulative relation $a \twoheadrightarrow_{\mathbb{L},ot} b$ that holds if it holds for any $L \in \mathbb{L}$. Similarly, we define $start_{ot}(\mathbb{L})$ and $end_{ot}(\mathbb{L})$. Fig. 2 shows the directly-follows graphs for the types of customers, orders and items from the log in Tables 1 and 2 with starts and ends.

2.4. Object-centric process trees

Object-centric process trees are a modeling formalism for languages of object-centric event logs [3]. They consist of inner operator nodes and outer leaf nodes in a tree structure. Inner operator nodes describe the control flow relation between the subtrees below them. Outer leaf nodes specify the related object types for each of the activities. Additionally, they define which object types are allowed to exhibit convergence, divergence and deficiency.

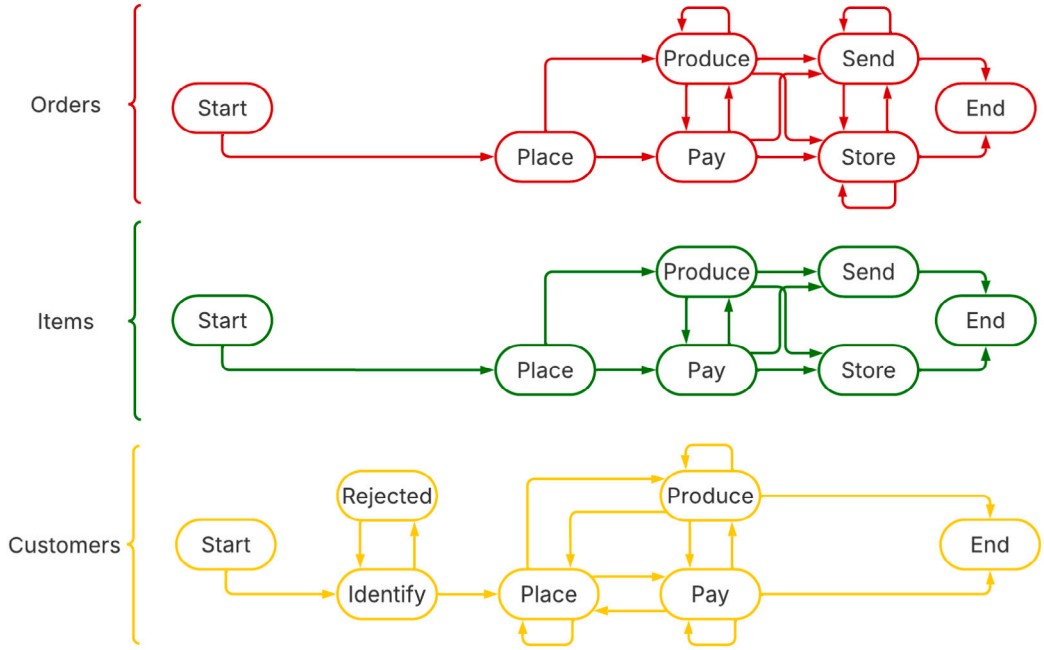


Fig. 2. Directly-follows relations of the object-centric example log from Tables 1 and 2 for orders (top, red), items (middle, green) and customers (bottom, yellow). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

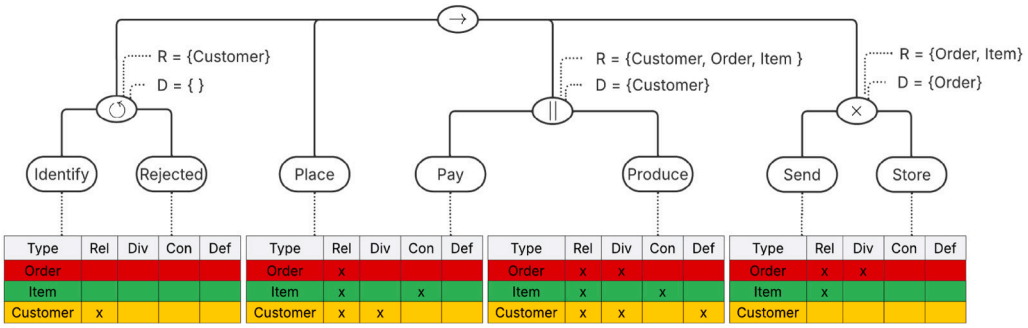


Fig. 3. Object-centric process tree for example order management process. The colored leaf configuration indicates which object types are included in $rel_a, div_a, con_a, def_a$. Additionally, the auxiliary constructs of $R(\ldots)$ and $D(\ldots)$ are indicated per operator node. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Formally, each leaf node in an object-centric process tree is a tuple $(a, rel_a, div_a\ con_a, def_a)$ with an activity label $a \in \mathbb{U}_{act}$ or the special character τ and a set of related object types $rel_a \subseteq \mathbb{U}_{typ}$. The special character τ can be used to express unobserved state changes that do not refer to any visible activity in a log. For each leaf, the subsets $div_a, con_a, def_a \subseteq rel_a$ specify the types that are allowed to exhibit divergence, convergence and deficiency respectively. Operator nodes $(\oplus, S_1, \ldots, S_n)$ are defined by an operator $\oplus \in \{\rightarrow, \times, \|, \circlearrowleft\}$ that specifies the control flow relation between the ordered subtrees $S_1 \ldots S_n$. The operators can express sequences ($\rightarrow$), concurrency ($\|$), exclusive choices ($\times$) or loops ($\circlearrowleft$) between the subtrees beneath the respective operator node. Fig. 3 shows an object-centric process tree for our running example.

The language of an object-centric process tree S is defined over all potential sets of objects $\Theta \subset \mathbb{U}_{obj}$, i.e. $\mathcal{L}(S) = \cup_{\Theta \subset \mathbb{U}_{obj}} \mathcal{L}(S, \Theta)$. The language of an object-centric tree S for a specific set of objects Θ is defined recursively as $\mathcal{L}(S, \Theta)$. For an object-centric leaf node $(a, rel_a, div_a, con_a, def_a)$, the activity a can be executed with objects from Θ, where only objects of related types can be involved in the execution (2). Additionally, objects are only allowed to cause divergence, convergence or deficiency if the type of these objects is in the respective type set of the leaf specification (3), (4), (5).

$$\mathcal{L}((a, rel_a, div_a, con_a, def_a), \Theta) = \{\langle (a, O_1) \ldots (a, O_n) \rangle \quad (1)$$

$$|\ \forall_{ot \in \Omega, 1 \leq i \leq n} : |\{o \in O_i \mid \omega(o) = ot\}| \neq \emptyset \iff ot \in rel_a \wedge \quad (2)$$

$$\forall_{ot \in rel_a, 1 \leq i \leq n} : |\{o \in O_i \mid \omega(o) = ot\}| > 1 \implies ot \in con_a \wedge \quad (3)$$

$$\forall_{ot \in rel_a, 1 \leq i \leq n} : |\{o \in O_i \mid \omega(o) = ot\}| < 1 \implies ot \in def_a \wedge \quad (4)$$

$$\forall_{ot \in rel_a, o \in \Theta} : |\{O_i \mid o \in O_i\}| \neq 1 \implies \omega(o) \in div_a\} \quad (5)$$

Operator nodes merge the languages of subtrees in accordance with the operator. The operator restrictions are enforced for each individual object. Additionally, the enforcement is conditioned on whether an object type is divergent or unrelated to all activities in the leaf nodes of a subtree. If an object type is divergent or unrelated to all leaf nodes in a subtree, it denotes that the respective object type does not have any distinct control flow in that part of the process, because of its interactions with otherwise independent objects of another type. Hence, if an object type is unrelated or divergent in all leaves of a subtree, the operator restriction does not apply for objects of that type. Based upon this conditioning, the operators are defined.

Formally, we introduce two core auxiliary constructs to keep track of the set of related and divergent object types respectively for each

node in the tree. Both constructs can be recursively defined. For individual leaf nodes, the set of related types can simply be extracted from the specification (6). For any operator node above, the union across all leaves below is created (7). In case of the fully divergent types, the specification is again simply copied from any given leaf node (8). For inner operator nodes, only those types are accounted for as fully divergent if they diverge on all related leaves (9).

$$R((a, rel_a, div_a, con_a, def_a)) = rel_a \tag{6}$$

$$R(S_1 \dots S_n) = \{ot \in \Omega \mid \exists_{1 \leq i \leq n} : ot \in R(S_i)\} \tag{7}$$

$$D((a, rel_a, div_a, con_a, def_a)) = div_a \tag{8}$$

$$D(S_1 \dots S_n) = \{ot \in R(S_1 \dots S_n) \mid \forall_{1 \leq i \leq n} : ot \notin R(S_i) \vee ot \in D(S_i)\} \tag{9}$$

For our running example, both constructs are indicated for some operator nodes in the tree structure in Fig. 3. Note that being unrelated to a leaf does not break the divergent aggregation. Using these two constructs, we can define the language of object-centric event logs accepted by an operator node. The concurrent operator (||) interleaves logs from each subtrees (10):

$$\mathcal{L}(\|\ (S_1, \dots, S_n), \Theta) = \{L \mid L \in \sqcup_{\{1 \leq i \leq n\}} L_i \wedge L_i \in \mathcal{L}(S_i, \Theta)\} \tag{10}$$

The exclusive choice operator also interleaves sublogs from all subtrees (11), but restricts the number of subtrees in which each object can occur. An object can only occur in sublogs from two different subtrees, if the respective object type diverges in all related leaf nodes of these subtrees (12).

$$\mathcal{L}(\times(S_1, \dots, S_n), \Theta) = \{L \mid L \in \sqcup_{\{1 \leq i \leq n\}} L_i \wedge L_i \in \mathcal{L}(S_i, \Theta_i) \tag{11}$$

$$\wedge \cup_{1 \leq i \leq n} \Theta_i = \Theta \wedge \forall_{i \neq j} \forall_{ot \notin D(S_i, S_j)} : \{o \in \Theta_i \cap \Theta_j \mid \omega(o) = ot\} = \emptyset\} \tag{12}$$

The sequence operator interleaves sublogs from all subtrees (13), but restricts the order in which objects can occur. For each pair of subtrees, objects must pass through them in order, except for objects of a type that diverges on all leaves of both subtrees and the subtrees between (14), (16):

$$\mathcal{L}(\rightarrow (S_1 \dots S_n), \Theta) = \{L \mid L \in \sqcup_{\{1 \leq i \leq n\}} L_i \wedge L_i \in \mathcal{L}(S_i, \Theta) \wedge \tag{13}$$

$$L = \langle (a_1, O_1) \dots (a_m, O_m) \rangle \wedge \forall_{i<j}, \forall_{ot \notin D(S_i \dots S_j)}, \forall_{(a_k, O_k), (a_l, O_l) \in L} : \tag{14}$$

$$(a_k, O_k) \in L_i \wedge (a_l, O_l) \in L_j \implies \tag{15}$$

$$k \leq l \vee \{o \in O_k \cap O_l \mid \omega(o) = ot\} = \emptyset\} \tag{16}$$

Lastly, the loop operator is defined as an interleaving between sublogs that alternate between the body part of the loop and the redo part of the loop. It can be defined as a recursive application of the sequence and exclusive choice operator for each of the potential loop iterations. For this purpose, we use the auxiliary constructs $S_{\circlearrowleft^1} = S_1$ and $S_\tau = (\tau, \Omega, \Omega, \Omega, \Omega)$. Then, we define the language for a fixed number of loop iterations (17) and merge the language of the loop operator nodes across all numbers of iterations (18).

$$S_{\circlearrowleft^i} = (\rightarrow, S_{\circlearrowleft^{i-1}}, (\times(S_\tau, (\rightarrow, (\times, S_2, \dots, S_n)), S_1))) \tag{17}$$

$$\mathcal{L}(\circlearrowleft\ (S_1, \dots, S_n), \Theta) = \{L \mid \exists m \in \mathbb{N} : L \in \mathcal{L}(S_{\circlearrowleft^m}, \Theta)\} \tag{18}$$

We refer to an object-centric log that is in the language of an object-centric process tree, except for all τ events, as perfectly fitting. For our running example, the log is in the language of the tree and as such fitting.

3. The object-centric inductive miner

In this section, we introduce the Object-Centric Inductive Miner (OCIM). In line with existing inductive discovery algorithms [10–14] it employs a recursive strategy to construct object-centric process trees from a set of object-centric logs $\mathbb{L}$ and their contained objects $\mathbb{O}$. At the beginning of the recursion, this set of input logs typically only includes a single log, i.e. the input log for the discovery with $\mathbb{L} = \{L\}$ and $\mathbb{O} = \Theta$. However, the recursion may result in multiple logs and different object sets later on.

Within each recursion step, we first check if any operator node with a τ-leaf needs to be added to represent the input logs. This may be necessary to guarantee fitness in case of fully optional subprocesses, especially within loop structures. If not, the alphabet Σ of the input logs is partitioned into multiple parts $\Sigma_1, \dots \Sigma_n$ with an operator $\oplus \in \{\rightarrow, \|, \times, \circlearrowleft\}$. We refer to this combination of an alphabet partition and an operator as a *cut*. Each cut becomes an operator node in the resulting object-centric process tree and is chosen such that it cannot impede the fitness of the object-centric input logs.

If no cut can be found, so-called *fallthrough* methods are applied. These methods ensure that even if the input log cannot be precisely represented as an object-centric process tree, it still fits the resulting structure. Conceptually, this is done by sacrificing precision of the final model. Based on the cut or fallthrough, the input logs are then split into sets of sublogs and passed down into further recursion steps. As soon as only a single activity is left in the alphabet of the input logs the recursion base case is triggered which adds a leaf node to the tree. Algorithm 1 show these steps of our approach.

In the following subsections we define cuts for each of the operators (Section 3.1) and specify their detection (Section 3.2). Then, we analogously define fallthroughs (Section 3.3) and their detection (Section 3.5). Finally, we specify methods used for the detection of τ-constructs (Section 3.6) the splitting of logs and the addition of leaf nodes in the base case (Section 3.7).

3.1. Cut definition

We define cuts for each of the operators in object-centric process trees. Conceptually, a cut refers to an operator and an alphabet partition that fulfill a set of conditions. These conditions were chosen such the operator, according to the language definition of object-centric process trees, enables exactly the behavior between the partition parts observed in the log. Hence, locally within the current recursion step, it embodies the optimal solution with regards to finding an object-centric process tree representation.

Formally, for a set of object-centric logs $\mathbb{L}$ with the activities Σ, a cut is an operator $\oplus \in \{\rightarrow, \|, \times, \circlearrowleft\}$ and a complete partition of the alphabet $\Sigma_1, \dots \Sigma_n$ with $\forall_{1 \leq i \leq n} : \Sigma_i \neq \emptyset \wedge \forall_{1 \leq i,j \leq n} : i \neq j \implies \Sigma_i \cap \Sigma_j = \emptyset$ and $\cup_{1 \leq i \leq n} \Sigma_i = \Sigma$. Additionally, the log must adhere to operator-specific requirements of a cut that reflect the language definition of the operator.

In the following, we present the conditions specific to each operator. Conceptually, these conditions can be grouped into four categories among two core dimension. First, they can be separated into conditions

Algorithm 1: Object-Centric Inductive Miner

Function OCIM($\mathbb{L}, \mathbb{O}$)
- $\oplus_\tau, \mathbb{O}' \leftarrow$ TAUCASE($\mathbb{L}, \mathbb{O}$)
- **if** $\oplus_\tau$ **then**
  - $\mathbb{L}', \emptyset \leftarrow$SPLITLOG($\mathbb{L}, \oplus_\tau, \Sigma, \emptyset$)
  - **return** $\oplus_\tau$, OCIM($\mathbb{L}', \mathbb{O}'$), $(\tau, \Omega, \Omega, \Omega, \Omega)$
- **if** $|\Sigma| = 1$ **then**
  - **return** BASECASE($\mathbb{L}$)
- $\oplus, \Sigma_1, \dots, \Sigma_n \leftarrow$FINDCUT($\mathbb{L}$)
- **if** ***not*** $\oplus$ **then**
  - $\oplus, \Sigma_1, \dots, \Sigma_n \leftarrow$FALLTHROUGH($\mathbb{L}$)
- $\mathbb{L}_1, \dots, \mathbb{L}_n \leftarrow$SPLITLOG($\mathbb{L}, \oplus, \Sigma_1, \dots, \Sigma_n$)
- **return** $\oplus$(OCIM($\mathbb{L}_1, \mathbb{O}$), … , OCIM($\mathbb{L}_n, \mathbb{O}$))

Table 3
Commonly used auxiliary constructs, such as quantifiers for the cut conditions.

Construct	Conceptual Meaning	Relevance
$\forall_{1\leq i<j\leq n} : \forall_{a\in\Sigma_i} : \forall_{b\in\Sigma_j} :$	for all pairs of activities from any two ordered partition parts	$\rightarrow$
$\forall_{1\leq i\neq j\leq n} : \forall_{a\in\Sigma_i} : \forall_{b\in\Sigma_j} :$	for all pairs of activities from any two different partition parts	$\|\|, \times$
$rel_a \cap rel_b \setminus D(\ldots)$	all shared related types of a and b that are locally fully divergent	$\rightarrow, \times, \circlearrowleft$
$rel_a \cap rel_b \cap D(\ldots)$	all shared related types of a and b that are locally not divergent	$\rightarrow, \times, \circlearrowleft$

that ensure the log to not violate any control flow restrictions of the operator and conditions that ensure the log to fully represent all possible behavior that is locally induced by the operator. Conceptually, they directly correspond to the preservation of fitness and precision. On a more technical level, these conditions can be further differentiated into those that refer to the distinct control flow restrictions of non-divergent object types and the allowed arbitrary behavior of fully divergent object types. To ease the interpretation, we summarize the most commonly used condition constructs in Table 3.

3.1.1. Sequence cut

Conceptually, the sequence operator ($\rightarrow$) enforces the order in which objects are involved in activities of different partition parts. That is, objects of each type can only perform the respective activities in the order of the partition parts. Exceptions from that order are only allowed for diverging object types with arbitrary control flow relations between the respective activities. As such, a sequence cut requires that the specified order is not violated by objects of non-diverging types (19). Diverging object types are required to exhibit a fully connected follows-relation for the respective partition parts (20). This ensures that the log fully represents the behavior allowed by the sequence operator for diverging types. Additionally, to avoid trivial cuts, the sequence operator requires the existence of shared, non-diverging object types between partition parts, if they follow each other (21). If no such type exist, any operator would be able to describe the control flow relation.

$$\forall_{1\leq i<j\leq n} : \forall_{a\in\Sigma_i} : \forall_{b\in\Sigma_j} : \forall_{ot\in rel_a\cap rel_b\setminus D(\Sigma_i,\ldots,\Sigma_j)} : a \twoheadrightarrow^{+}_{\mathbb{L},ot} b \wedge b \not\twoheadrightarrow^{+}_{\mathbb{L},ot} a \tag{19}$$

$$\forall_{1\leq i<j\leq n} : \forall_{a\in\Sigma_i} : \forall_{b\in\Sigma_j} : \forall_{ot\in rel_a\cap rel_b\cap D(\Sigma_i,\ldots,\Sigma_j)} : a \twoheadrightarrow_{\mathbb{L},ot} b \wedge b \twoheadrightarrow_{\mathbb{L},ot} a \tag{20}$$

$$\forall_{1<i<n} : \exists_{a\in\Sigma_i} : \exists_{b\in\Sigma_{i+1}} : \exists_{ot\in\Omega} : ot \in (rel_a \cap rel_b) \setminus (div_a \cap div_b) \tag{21}$$

Fig. 4 shows a sequence cut on the directly-follows relations of the example log in Tables 1 and 2 with four partition parts. Note that for the object types of items and orders, which do not diverge in all related activities of any two following partition parts, the activities are executed in a sequential correct order. For the customer, which diverges in all activities of the second and third partition part, those activities are fully connected.

3.1.2. Concurrent cut

The concurrent operator (||) interleaves sublogs from both subtrees. Conceptually, this means that activities in different partition parts need to be bidirectionally connected by the directly-follows relation for the operator to perfectly represent the input logs (22). Since the arbitrary interleaving of the concurrent operator keeps the partial order within each log from each subtree, the respective start and end activities are preserved. That is, every start activity of an individual partition part must also be a start activity of the overall logs (23). The same condition applies to the end activities (24).

$$\forall_{i\neq j} : \forall_{a\in\Sigma_i} : \forall_{b\in\Sigma_j} : \forall_{ot\in rel_a\cap rel_b} : a \twoheadrightarrow_{\mathbb{L},ot} b \wedge b \twoheadrightarrow_{\mathbb{L},ot} a \tag{22}$$

Table 4
Log from Tables 1 and 2 filtered on the third partition part in Fig. 4.

Event #	Activity	Customers	Orders	Items
6	PRODUCE	c_1	o_1	i_1
7	PRODUCE	c_1	o_1	i_2
8	PAY	c_1	o_1	i_1, i_2
9	PAY	c_1	o_2	i_3, i_4
10	PRODUCE		o_2	i_3
11	PRODUCE		o_2	i_4
13	PRODUCE		o_3	i_6
14	PAY	c_1	o_3	i_5, i_6
15	PRODUCE	c_1	o_3	i_5
18	PRODUCE		o_4	i_7
19	PRODUCE		o_4	i_8
21	PAY	c_1	o_4	i_7, i_8
24	PAY	c_2	o_5	i_9
28	PRODUCE	c_2	o_5	i_9

Table 5
Log from Tables 1 and 2 filtered on the last partition part in Fig. 4.

Event #	Activity	Customers	Orders	Items
22	STORE		o_1	i_1
23	STORE		o_1	i_2
25	STORE		o_2	i_3
26	SEND		o_2	i_4
27	SEND		o_3	i_5
29	STORE		o_3	i_6
30	SEND		o_4	i_8
30	SEND		o_4	i_7
30	SEND		o_5	i_9

$$\forall_{i\leq n} : \forall_{a\in\Sigma_i} : \forall_{ot\in rel_a} : a \in start_{ot}(\{L\downarrow_{\Sigma_i} \mid L \in \mathbb{L}\}) \implies a \in start_{ot}(\mathbb{L}) \tag{23}$$

$$\forall_{i\leq n} : \forall_{a\in\Sigma_i} : \forall_{ot\in rel_a} : a \in end_{ot}(\{L\downarrow_{\Sigma_i} \mid L \in \mathbb{L}\}) \implies a \in end_{ot}(\mathbb{L}) \tag{24}$$

Fig. 5 shows the directly-follows relations for the subset of the activities that corresponds to the third partition part of the sequence cut in Fig. 4. The correspondingly filtered log can be seen in Table 4. On this directly-follows graph, a concurrent cut with two partition parts exists, since all object types have bidirectional connections between the activities in both parts. Additionally, start and end activities of the parts are also start and end activities of the overall log and as such preserved correctly by the cut (see Fig. 6).

3.1.3. Exclusive choice cut

For the exclusive choice ($\times$), the log is not allowed to include any connections between activities in different partition parts if the respective object type does not diverge on all activities in both partitions (25). This enforces the exclusive choice between the activities in each partition part. Reversely, object types that diverge on two parts, must show full connectivity between related activities in different parts (26). This again ensures the full coverage of the locally allowed behavior of the operator in the input logs. Additionally, start and end activities must be preserved as for the concurrent operator (23), (24). Lastly, for the cut to not be trivial, each pair of partition parts must share at least one non-divergent type between their activities (27).

$$\forall_{i\neq j} : \forall_{a\in\Sigma_i} : \forall_{b\in\Sigma_j} : \forall_{ot\in rel_a\cap rel_b\setminus D(\Sigma_i,\Sigma_j)} : a \not\twoheadrightarrow_{\mathbb{L},ot} b \wedge b \not\twoheadrightarrow_{\mathbb{L},ot} a \tag{25}$$

$$\forall_{i\neq j} : \forall_{a\in\Sigma_i} : \forall_{b\in\Sigma_j} : \forall_{ot\in D(\Sigma_i,\Sigma_j)} : a \twoheadrightarrow_{\mathbb{L},ot} b \wedge b \twoheadrightarrow_{\mathbb{L},ot} a \tag{26}$$

$$\forall_{i\neq j} : \exists_{a\in\Sigma_i} : \exists_{b\in\Sigma_j} : \exists_{ot\in\Omega} : ot \in (rel_a \cap rel_b) \setminus (div_a \cap div_b) \tag{27}$$

Fig. 5 shows the directly-follows relations for the last partition part from the cut in Fig. 4, with the correspondingly filtered log in Table 5. These directly-follows relations result in an exclusive choice cut. The items, which are not divergent on the two partition parts, do not have any directly-follows relations between them, while the divergent order is fully connected. Additionally, start and end activities are correctly

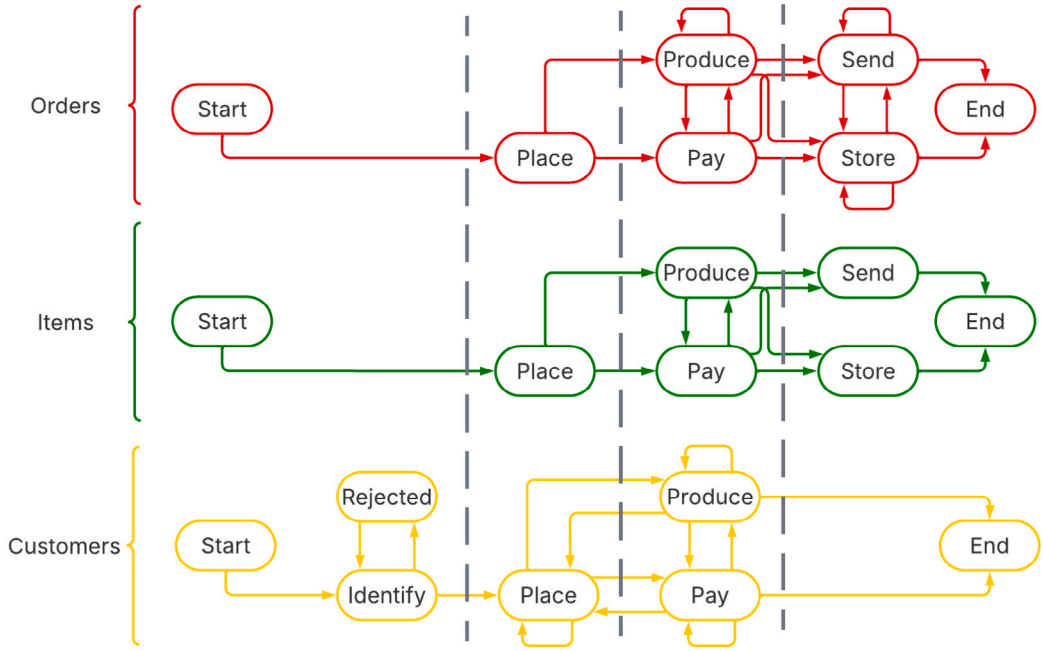


Fig. 4. Sequence cut for the object-centric example log from Tables 1 and 2.

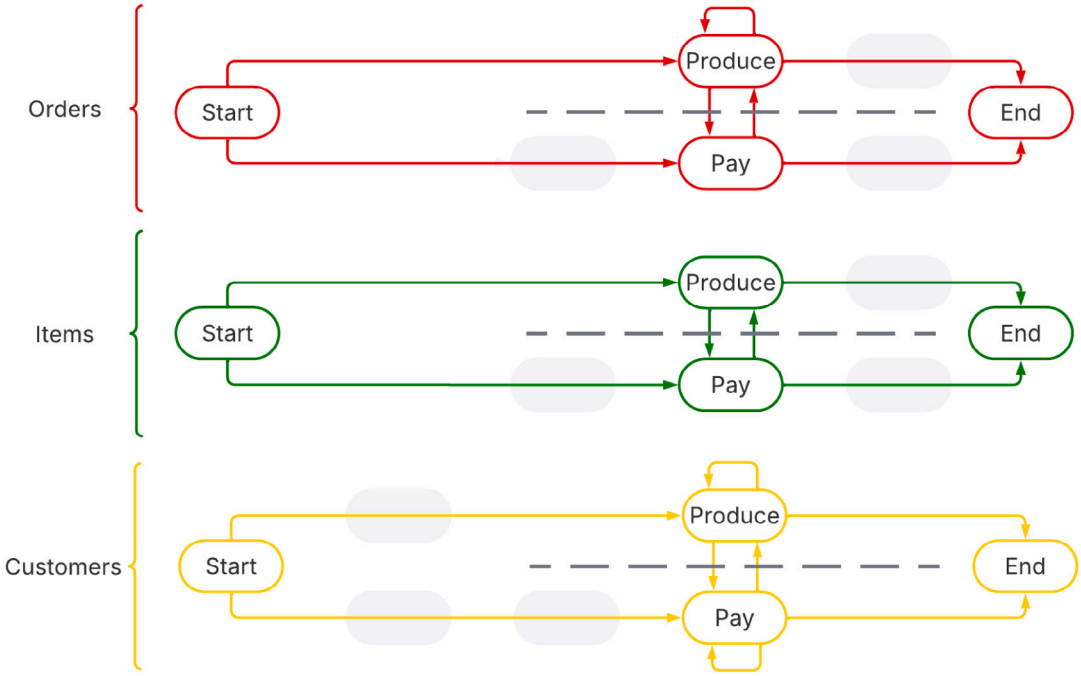


Fig. 5. Concurrent cut for the filtered object-centric example log in Table 4.

preserved. Note that we omit the customer here, as this type is not related to any of the activities.

3.1.4. Loop cut

In accordance with the language definition for object-centric process trees [3], we only consider loop cuts with exactly two parts, referred to as the body part and the redo part of the loop. The loop operator ($\circlearrowleft$) allows repetitive behavior of the body part with each iteration being initiated by the redo part. This control flow is again only enforced for objects of non-diverging Between the two partition parts, there has to be at least one object type that is not fully divergent on its related activities for the cut to not be trivial (28). As for the previous operators, diverging object types are required to exhibit a fully connected directly-follows relation between the partition parts (29).

$$\exists_{a \in \Sigma_1} : \exists_{b \in \Sigma_2} : \exists_{ot \in \Omega} : ot \in rel_a \cap rel_b \setminus D(\Sigma_1, \Sigma_2) \tag{28}$$

$$\forall_{a \in \Sigma_1} : \forall_{b \in \Sigma_2} : \forall_{ot \in rel_a \cap rel_b \cap D(\Sigma_1, \Sigma_2)} : a \twoheadrightarrow_{\mathbb{L}, ot} b \wedge b \twoheadrightarrow_{\mathbb{L}, ot} a \tag{29}$$

The non-diverging object types can perform any unlimited number of loop iterations. As such, for the log to be perfectly represented by

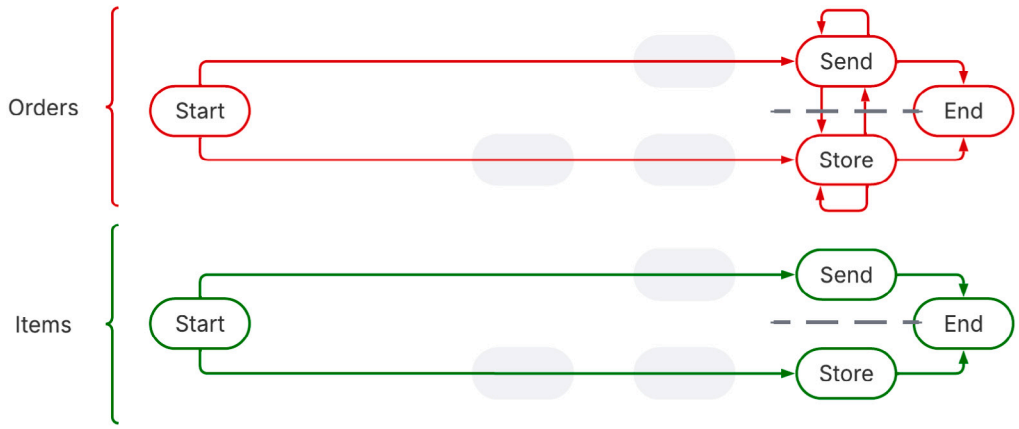


Fig. 6. Exclusive choice cut for the filtered object-centric example log in Table 5.



Fig. 7. Loop cut for the filtered log for the first partition part in Fig. 4.

that operator, any pair of activities will eventually follow each other, given enough loop iterations. Due to this structure, all object types should exhibit transitive paths between all activity pairs in their follows relation (30). Non-diverging object types that are related to activities in both partition parts must have all start and end activities in the body part of the loop (31), (32). This is strictly necessary as the loop can only start and end with the body part of the loop. Additionally, directly-follows relations for non-diverging object types that cross partition parts must involve a start or end activity (33), (34). Conceptually, this ensures that no shortcuts are used into or from any of the redo parts, that do not originate or tie back into the body part correctly.

$$\forall_{a,b\in\Sigma} : \forall_{ot\in rel_a\cap rel_b} : a \twoheadrightarrow^{+}_{\mathbb{L},ot} b \wedge b \twoheadrightarrow^{+}_{\mathbb{L},ot} a \tag{30}$$

$$\forall_{ot\in\Omega} : \exists_{a\in\Sigma_1,b\in\Sigma_2} : (ot \in rel_a \cap rel_b \setminus D(\Sigma_1,\Sigma_2)) \implies start_{ot}(\mathbb{L}) \subseteq \Sigma_1 \tag{31}$$

$$\forall_{ot\in\Omega} : \exists_{a\in\Sigma_1,b\in\Sigma_2} : (ot \in rel_a \cap rel_b \setminus D(\Sigma_1,\Sigma_2)) \implies end_{ot}(\mathbb{L}) \subseteq \Sigma_1 \tag{32}$$

$$\forall_{a\in\Sigma_1} : \forall_{b\in\Sigma_2} : \forall_{ot\in rel_a\cap rel_b\setminus D(\Sigma_1,\Sigma_2)} : a \twoheadrightarrow_{\mathbb{L},ot} b \implies a \in end_{ot}(\mathbb{L}) \tag{33}$$

$$\forall_{b\in\Sigma_1} : \forall_{a\in\Sigma_2} : \forall_{ot\in rel_a\cap rel_b\setminus D(\Sigma_1,\Sigma_2)} : a \twoheadrightarrow_{\mathbb{L},ot} b \implies b \in start_{ot}(\mathbb{L}) \tag{34}$$

Fig. 7 shows the directly follow relation for the two activities in the first partition part of the sequence cut in Fig. 4. The correspondingly filtered log is $\langle \text{identify}\{c_1\}, \text{reject}\{c_1\}, \text{identify}\{c_1\}, \text{identify}\{c_2\}\rangle$. Fig. 7 additionally indicates a loop cut, with the object types of items and orders being omitted as they are not related to the respective activities. For the object type of customers, all relevant conditions from (28)–(34) hold. The only start and end activity, the identification, forms the body part of the loop, and the other remaining activity, the rejection, can be reached in both direction, without it being a start or end activity of the log itself.

3.2. Cut detection

Next, we propose detection methods to decide in polynomial runtime complexity if a cut with a given operator exists and to determine its respective alphabet partition. For each of the operators, we prove that if a cut exists, the corresponding partition will be found by our method. Reversely, we show that if a partition is found by our method it constitutes a cut. The details of the individual procedures are provided in Appendix 8, since they require excessive space to specify and prove. Here, we focus on the conceptual ideas behind the detection strategy and an application to our running example.

The four operators are sequentially checked for cuts in the following order: $\rightarrow, \times, \circlearrowleft, ||$. The detection methods for the concurrent, exclusive choice, sequence and loop operator employ a common strategy. First, pairs of activities are iteratively identified such that a separation of them into different parts of the alphabet partition would inevitably result in a violation of one of the conditions. These activity pairs are then conceptualized as edges in an undirected graph with nodes for each activity. Subsequently, we detect the connected components in that graph to ensure that the resulting partition does not violate any of the cut conditions. If the graph cannot be split into multiple components, no cut with the respective operator exists. If it can be split, we return the respective alphabet partition as a cut for the input logs.

We illustrate that strategy on our input log for the running example. For the exclusive choice and sequence operator, the graph with edges for cut violations upon partitioning activities is shown as an example in Fig. 8. The colors indicate which object type is responsible for each edge, i.e. would violate a cut with that operator. For the exclusive choice operator, the customer connects the three activities of reject, identify and place. This is due to the existence of corresponding directly-follows relations between these activities for that type. If an exclusive choice cut would be present, those activities can only be in the same partition part for that reason. Similarly, the order connects the activity place with the activities produce and pay, while the items has



Fig. 8. Auxiliary graphs for the cut detection on the log from Tables 1 and 2.

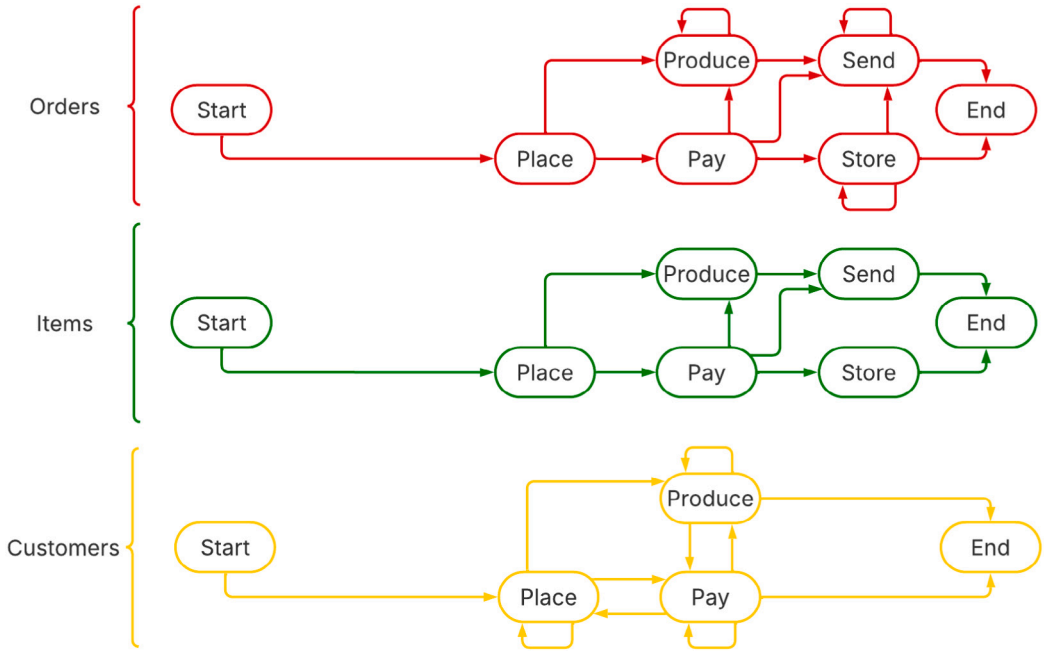


Fig. 9. Directly follows relation for the log in Tables 1 and 2, artificially filtered for a subset of activities and relations to showcase the fallthrough appliance.

directly-follow edges from produce and pay to send and store. As such, all activities are transitively connected by violation-representing edges, which means that there can be no exclusive choice cut in that log. The same effect can be observed for the concurrent and loop operator, which result in a transitively connected violation graph. In contrast, the sequence operator allows the detection of multiple connected components in the corresponding graph, as shown in Fig. 8. The only edges in that graph induced by the customer are between reject and identity, as those are bidirectionally connected. The placement activity is however not, as it is only connected by the directly-follows relation in one direction, which is allowed by the sequence cut conditions. The activities of produce and pay, are analogously connected, due to the bidirectional directly-follows relation of the order. Lastly, the activities of send and store are connected by the item type, since it does not have any directly-follows relation between the two. The resulting cut that would be detected based on this violation graph, corresponds to Fig. 4.

Note that Appendix 8 contains a detailed technical description of the detection strategy per type, and corresponding auxiliary proof work that we require for the fitness and rediscovery guarantee. We guarantee, that if a cut exists, it will be found and that if a cut is found, it is correct.

3.3. Fallthroughs definition

Unfortunately, it is not guaranteed that a cut exists for a given set of object-centric event logs. This might be due to incorrectness or incompleteness of the input logs. It can also be caused by an underlying business process that is outside the utilized representational bias of the trees. An example for such a situation can be seen in Fig. 9 which shows an alternative directly-follows relation for the log from Tables 1 and 2 with some alterations. Performing the cut detection will yield no results, since every partition and operator induce at least one violation of the respective cut conditions.

In this case, we determine a fallthrough $\oplus, \Sigma_1, \dots \Sigma_n$ that does not restrict the fitness of the resulting model by allowing more behavior than what is observed in the input log. While determining fallthroughs, we try to minimize this wrongfully allowed behavior to maximize the precision of the tree.

In the example in Fig. 9, the previously detected sequence cut between the first partition part with place, the second partition part with pay & produce and the final partition part with store & send is not valid, since the transitive closure of the graph does not contain paths from all activities to all other activities in succeeding partition parts. For example, producing items is not transitively followed by storing it. This means that the log does not perfectly reflect the behavior induced by the operator. However, the operator could still be suitable to at least provide a fitting, even when not completely precise, representation of the object-centric input logs.

For this purpose, we relax the definition of cuts from the previous sections, such that only those conditions are preserved that would impede fitness in case of a violation. Additionally, we introduce measures to estimate the precision decrease induced by this relaxation. Based on this measure, we can select the least precision reducing fallthrough that does not impede fitness.

For these precision estimates, we determine which directly-follows relations a fallthrough $\oplus, \Sigma_1, \dots \Sigma_n$ would allow in the resulting object-centric process tree, denoted as $P_{\oplus}(\Sigma_1, \dots \Sigma_n)$. Then, we check which of these relations cannot be found in the input logs, constituting the set of precision violations $P^{\not\in}_{\oplus}(\Sigma_1, \dots \Sigma_n)$. Lastly, we set the number of relations in these two sets into relation, giving us a quality estimate of the fallthrough, formally defined as $Q_{\oplus}(\Sigma_1, \dots \Sigma_n) = 1 - (|P^{\not\in}_{\oplus}(\Sigma_1, \dots \Sigma_n)| \cdot$

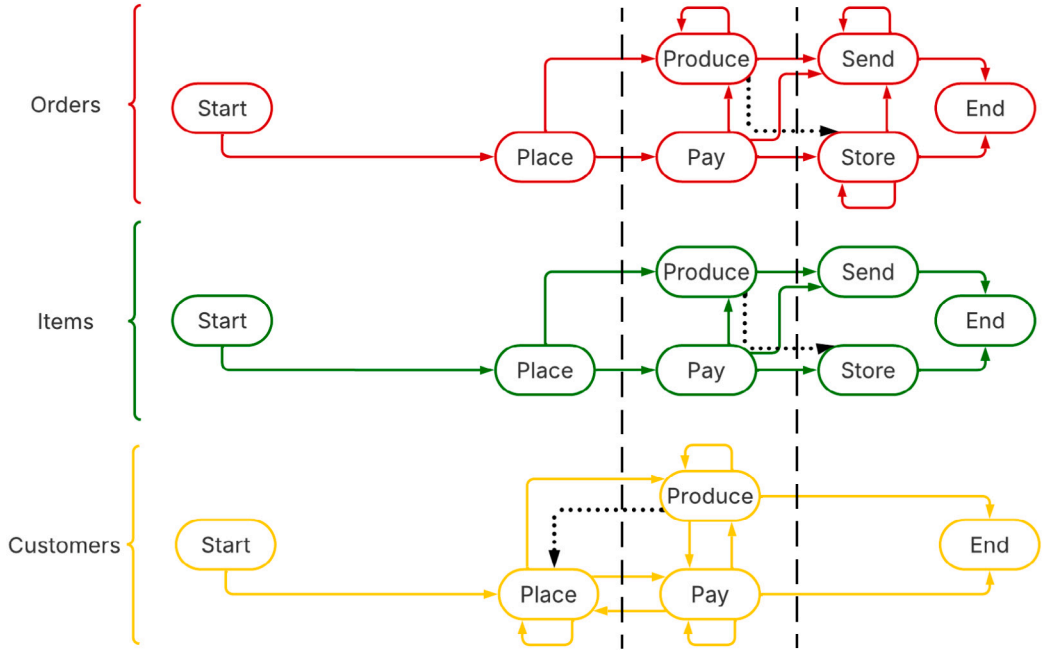


Fig. 10. Directly follows relation from Fig. 9 with a sequence fallthrough partition (dashed lines) and the corresponding precision violations (dotted arrows).

$|P_{\oplus}(\Sigma_1, \dots \Sigma_n)|^{-1}$). Then, we construct the fallthrough with the highest quality estimate.

3.3.1. Sequence fallthrough

The sequence operator can only restrict fitness if the order between partition parts is violated. As such, every partition is a sequence fallthrough, if the input logs include no such order violation (19). Regarding precision, the sequence operator allows bi-directional directly follows relations between activities in different partition parts by diverging object types (36). Additionally, it allows transitive directly-follows relations between activities by non-diverging object types in the correct order of the partition parts (37). Precision violations are all corresponding relations that are absent (39), (40).

$$P_{\rightarrow}(\Sigma_1, \dots \Sigma_n) = \{(a, b, ot, 1) \mid a \in \Sigma_i \wedge b \in \Sigma_j \tag{35}$$

$$\wedge ot \in rel_a \cap rel_b \cap (D(\Sigma_i, \dots, \Sigma_j) \cup D(\Sigma_j, \dots, \Sigma_i))\} \tag{36}$$

$$\cup \{(a, b, ot, 2) \mid (a \in \Sigma_i \wedge b \in \Sigma_j \wedge i < j \wedge ot \in rel_a \cap rel_b\} \tag{37}$$

$$P^{\not}_{\rightarrow}(\Sigma_1, \dots \Sigma_n) = \{(a, b, ot, 1) \mid a \in \Sigma_i \wedge b \in \Sigma_j \tag{38}$$

$$\wedge ot \in rel_a \cap rel_b \cap (D(\Sigma_i, \dots, \Sigma_j) \cup D(\Sigma_j, \dots, \Sigma_i)) \wedge a \not\rightarrow_{\mathbb{L},ot} b\} \tag{39}$$

$$\cup \{(a, b, ot, 2) \mid a \in \Sigma_i \wedge b \in \Sigma_j \wedge i < j \wedge ot \in rel_a \cap rel_b \wedge a \not\rightarrow^{+}_{\mathbb{L},ot} b\} \tag{40}$$

The previously discussed partition on the directly follows relation from Fig. 9 is shown in Fig. 10, including its respective precision violations.

3.3.2. Concurrent fallthrough

A concurrent cut cannot impede fitness, since the operator allows arbitrary interleavings in the resulting tree. As such, every partition with the concurrent operator is a concurrent fallthrough. However, for the very same reason of allowing arbitrary interleavings, the precision of the model suffers.

Condition (22) requires for a concurrent cut that activities in different partition parts are bi-directionally connected by the directly follows relations. These are also the directly-follows relations that the cut allows (42). All missing relations are precision violations of a corresponding fallthrough (44).

$$P_{||}(\Sigma_1, \dots \Sigma_n) = \{(a, b, ot) \mid \tag{41}$$

$$a \in \Sigma_i \wedge b \in \Sigma_j \wedge ot \in rel_a \cap rel_b\} \tag{42}$$

$$P^{\not}_{||}(\Sigma_1, \dots \Sigma_n) = \{(a, b, ot) \mid \tag{43}$$

$$a \in \Sigma_i \wedge b \in \Sigma_j \wedge ot \in rel_a \cap rel_b \wedge a \not\rightarrow_{\mathbb{L},ot} b\} \tag{44}$$

Fig. 11 shows a concurrent fallthrough for one of the partition parts that results from applying the sequence fallthrough in Fig. 10.

3.3.3. Exclusive choice fallthrough

The exclusive choice operator may restrict fitness, since it requires activities in different partition parts to not have any directly follows relation through a non-diverging object type (25). Hence, a partition with the exclusive choice operator is only a fallthrough, if this condition is fulfilled. The directly-follows relations required from an exclusive choice cut, are bidirectional connections by diverging types (26). These are the only relations the cut allows (46) and the only potential precision violations (48).

$$P_{\times}(\Sigma_1, \dots \Sigma_n) = \{(a, b, ot) \mid \tag{45}$$

$$a \in \Sigma_i \wedge b \in \Sigma_j \wedge ot \in rel_a \cap rel_b \cap D(\Sigma_i, \Sigma_j)\} \tag{46}$$

$$P^{\not}_{\times}(\Sigma_1, \dots \Sigma_n) = \{(a, b, ot) \mid \tag{47}$$

$$a \in \Sigma_i \wedge b \in \Sigma_j \wedge ot \in rel_a \cap rel_b \cap D(\Sigma_i, \Sigma_j) \wedge a \not\rightarrow_{\mathbb{L},ot} b\} \tag{48}$$

We again show an example exclusive choice fallthrough for a partition part of the sequence fallthrough in Fig. 10 in Fig. 12 with precision violations. Notice how the missing behavior in the log with respect to the exclusive choice fallthrough can only refer to the arbitrary behavior expected from locally fully divergent object types, such as the order here.

3.3.4. Loop fallthrough

The loop operator may restrict fitness by not allowing directly-follows relations between activities in different partition parts by non-diverging object types if they are not start or end activities. As such, every potential partition is a loop fallthrough, if the conditions (31)–(34) hold.

The loop operator allows all pairs of activities to transitively follow each other (50). Additionally, diverging object types exhibit directly

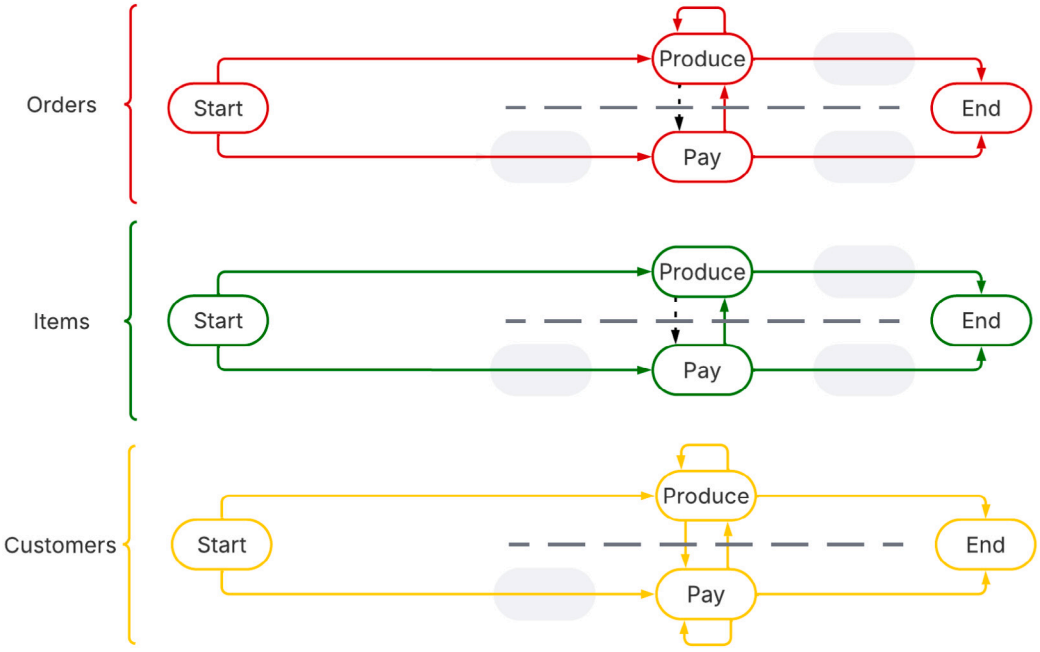


Fig. 11. Directly follows relation subset from Fig. 9 with a concurrent fallthrough partition (dashed lines) and the corresponding precision violations (dotted arrows).

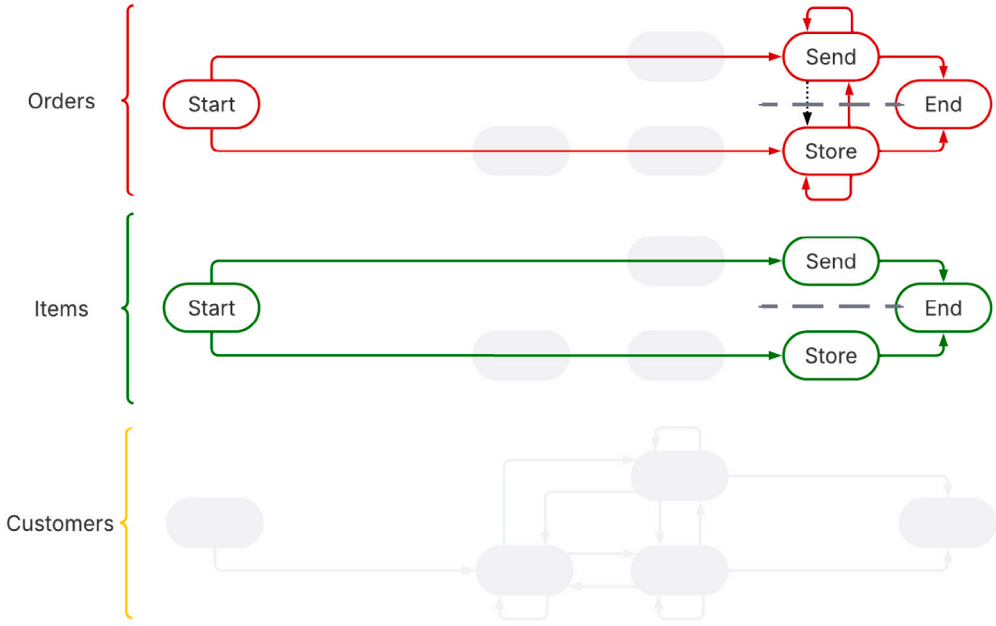


Fig. 12. Directly follows relation subset from Fig. 9 with a exclusive choice partition (dashed lines) and the corresponding precision violations (dotted arrows).

follows relation between all activities in different partition parts (51). All correspond relations that are not present in the log are counted as violations (53), (54).

$$P_{\circlearrowleft}(\Sigma_{body}, \Sigma_{redo}) = \{(a, b, ot, 1) \mid a \in \Sigma_{body} \neq b \in \Sigma_{body} \tag{49}$$

$$\wedge ot \in rel_a \cap rel_b \cap D(\Sigma_{body}, \Sigma_{redo})\} \tag{50}$$

$$\cup\{(a, b, ot, 2) \mid a, b \in \Sigma \wedge ot \in rel_a \cap rel_b\} \tag{51}$$

$$P^{\nexists}_{\circlearrowleft}(\Sigma_{body}, \Sigma_{redo}) = \{(a, b, ot, 1) \mid a \in \Sigma_{body} \neq b \in \Sigma_{body} \tag{52}$$

$$\wedge ot \in rel_a \cap rel_b \cap D(\Sigma_{body}, \Sigma_{redo}) \wedge a \not\Rightarrow_{\mathbb{L},ot} b\} \tag{53}$$

$$\cup\{(a, b, ot, 2) \mid a, b \in \Sigma \wedge ot \in rel_a \cap rel_b \wedge a \not\Rightarrow^{+}_{\mathbb{L},ot} b\} \tag{54}$$

Note that we intentionally exclude the loop fallthrough from our adjusted version of the running example, since an optimal loop fallthrough in comparison to the other operators would require an excessively complex setup.

3.4. Running example

For our running example, the optimal fallthroughs based on a brute force iteration through all possible ones, is very similar to the cuts discovered on the overall log for the same activity set. That is, a sequence between the partition parts of (1) place, (2) pay and produce and (3) store and send is discovered as shown in Fig. 10, since only very few relations are missing for this partition to constitute an actual cut. In subsequent recursion steps, the concurrency between pay and produce as shown in Fig. 11 and the exclusive choice between store and send as shown in Fig. 12 is discovered as well. Note that the most important conceptual role of the fallthroughs is to still provide reasonable precision in case of no cut being found on the input logs. For the sake of the fitness guarantee, one could simply resort to choosing a concurrent constructs between a self loop of each activity. That would preserve fitness as well, but lack precision to an extreme extent. The selection of fallthroughs based on the definition and quality estimates, allows us to chose the locally most precise one without compromising fitness.

3.5. Fallthrough detection

Next, we propose approximated methods to find a good fallthrough in polynomial runtime complexity. If runtime is not of concern, or the input log is relatively small, a brute force approach across all potential partitions can still be used for optimal results, by solving $\text{ARGMAX}_{\oplus, \Sigma_1, \dots \Sigma_n} Q_\oplus(\Sigma_1, \dots \Sigma_n)$.

For cases in which the brute force detection is not feasible, we restrict the investigated search space to fallthroughs with exactly two partition parts. For each operator, we first determine groups of activities that have to be in the same partition part. For this purpose, we reuse the detection of connected components via violation graphs from the detection of cuts. Then, we define distance measures between these groups of activities that correspond to the avoided precision loss, if these groups end up in the same partition part. As a result, groups of activities that would cause a lot of precision loss in case of being separated, are closer together. For the concurrent and exclusive choice operator, we then apply clustering based on these distance measures to determine an alphabet partition that minimizes the precision loss. For the sequence and loop operator this clustering is not needed, because only a linear number of fallthroughs exists after the detection of the connected component. After determining a fallthrough per operator, we return the best.

3.5.1. Concurrent fallthrough detection

All binary alphabet partitions constitute a concurrent fallthrough. As such, no groups of activities need to be identified that have to be in the same partition part. Each activity constitutes its own group for the subsequent clustering. The more precision violations a separation of activities induces, the closer they should be to each other. As such, we use the percentage of directly-follows relations between them for the concurrent operator distance (55). Based on that distance measure we cluster the activities into two partition parts $\Sigma_{\|,1} \Sigma_{\|,2}$.

$$dist_{\|}(a,b) = 1 - \frac{\cup_{ot \in rel_a \cap rel_b} \{(a,b,ot) \mid a \not\twoheadrightarrow_{\mathbb{L},ot} b\} \cup \{(b,a,ot) \mid b \not\twoheadrightarrow_{\mathbb{L},ot} a\}}{\cup_{ot \in rel_a \cap rel_b} \{(b,a,ot)\} \cup \{(b,a,ot)\}} \tag{55}$$

3.5.2. Exclusive choice fallthrough detection

For the exclusive choice operator, all activities that share any directly-follows relation by a non-diverging type need to be in the same partition part. We perform a connected component analysis on the activity graph with the corresponding edges to generate a first partition $\Sigma_1, \dots \Sigma_n$. Note that it might be, that only a single connected component is detected. In this case, no fallthrough with the exclusive choice operator exists. Then, we define a distance measure for these groups to account for the potential precision loss by diverging object types (56). Based on that distance measure we cluster the activities into two partition parts $\Sigma_{\times,1} \Sigma_{\times,2}$.

$$dist_{\times}(\Sigma_i, \Sigma_j) = 1 - \frac{\{(a,b,ot) | a \twoheadrightarrow_{\mathbb{L},ot} b \wedge ot \in rel_a \cap rel_b \cap D(\Sigma) \wedge ((a \in \Sigma_i \wedge b \in \Sigma_j) \vee (a \in \Sigma_j \wedge b \in \Sigma_i))\}}{\{(a,b,ot) | ot \in rel_a \cap rel_b \cap D(\Sigma) \wedge ((a \in \Sigma_i \wedge b \in \Sigma_j) \vee (a \in \Sigma_j \wedge b \in \Sigma_i))\}} \tag{56}$$

3.5.3. Sequence fallthrough detection

The sequence operator requires fallthroughs to not violate the order of activities. As such, all pairs of activities that have a bidirectional transitive order for a related non-diverging type need to be in the same partition part. Applying the connected component detection on the activity graph with the corresponding edges results in the partition $\Sigma_1, \dots \Sigma_n$. Note that it might be, that only a single connected component is detected. In this case, no fallthrough with the sequence operator can exist. We again use the partition-follows relation from the cut detection method to give them the correct order. All partitions that form a cycle in that relation are merged into one. Then, we check for which i the fallthrough $\rightarrow, \Sigma', \Sigma''$ with $\Sigma' = \cup_{1 \leq j \leq i} \Sigma_j$ and $\Sigma'' = \cup_{i < j \leq n} \Sigma_j$ has the highest quality estimate and return it.

3.5.4. Loop fallthrough detection

For a loop fallthrough, the conditions (31)–(34) need to hold. As such, we require all start and end activities of each object type to be in the same partition part. Similarly, activities that share any directly-follows relation by a non-diverging object type that are not start or end activities have to be in the same partition part as well. We again use the connected component detection to construct the first partition $\Sigma_1, \dots \Sigma_n$. Again, if only a single component is detected, no loop fallthrough can exist. Then, we iterate over partition parts and check if treating them as the body part of the loop and all others merged as the redo part leads to a fallthrough. For each fallthroughs, we determine the estimated precision loss and return the best.

3.6. Additional construct detection

The detection of cuts and fallthroughs exclude constructs with τ-leafs. For the sequence and concurrent operator, this is intended, since τ-leaves beneath them can be dropped without changing the language of the tree. However, τ-leaves beneath exclusive choice and loop operators cannot be ignored. They express optional behavior and loops with empty redo parts. This is of utmost importance for the fitness guarantee, as these concepts cannot be reflected in any language equivalent object-centric process trees without the use of τ-leaves below the two affected operators. Not explicitly accounting for them would as such limit the applicability of that guarantee.

For the optional behavior, we check if for any type in $\cup_{a \in \Sigma} rel_a$, any object exists in $\mathbb{O}$ that is not present in $\mathbb{L}$. This clearly indicates, that some objects of the respective types completely skip the execution of the remaining activities, which requires a τ-leaf to represent correctly If so, the τ-construct for optional behavior with the exclusive choice operator is returned. All respective objects are removed from $\mathbb{O}$ to avoid duplicating the same τ-construct in subsequent recursion steps, which would unnecessarily reduce precision without any alternative benefits.

For loops with empty redo parts we check if all object types exhibit repetitive behavior. This requires directly-follows relations from all end activities to all respective start activities per object type, i.e. $\forall_{ot \in \Omega} : \forall_{a \in start_{ot}(\mathbb{L})} : \forall_{b \in end_{ot}(\mathbb{L})} : b \twoheadrightarrow_{\mathbb{L},ot} a$. Additionally, all activities should be transitively connected in both directions with $\forall_{a,b \in \Sigma} : \forall_{ot \in rel_a \cap rel_b} : a \twoheadrightarrow^+_{\mathbb{L},ot} b \wedge b \twoheadrightarrow^+_{\mathbb{L},ot} a$. To ensure this is not caused by diverging object types, at least one type must be non-divergent, i.e. $\exists_{a \in \Sigma} : \exists_{ot \in \Omega} : ot \in rel_a \setminus div_a$. Should these conditions be met, we can conclude that the log

contains repetitive behavior that is not conditioned on any redo part, since it contains direct connections from end activities back to start activities. We use the method $\text{TAUCASE}_{oc}(\mathbb{L}, \mathbb{O})$ to detect the need for the corresponding τ-constructs, shown in Algorithm 2.

3.7. Log splitting & base case

Once a cut, fallthrough, or τ-construct is detected by the methods from the previous sections, the input logs are split and passed down into subsequent recursion steps. Relevant for the formal guarantees of our approach is that the splitting only removes information that can be deducted based on merging the sublogs based on the language definition of the detected operator. For a given set of object-centric event logs $\mathbb{L}$ and a cut or fallthrough $\oplus, \Sigma_1, \dots, \Sigma_n$ with the operator $\oplus \in \{\|, \times, \rightarrow\}$, we filter all input logs onto the activities in the partition part, i.e. $\mathbb{L}_i = \{L \downarrow_{\Sigma_i} \mid L \in \mathbb{L}\}$.

For the loop operator, additional processing is required to identify loop iterations. Let $\circlearrowleft, \Sigma_{body}, \Sigma_{redo}$ be a cut or fallthrough for $\mathbb{L}$. For each object-centric event log $L \in \mathbb{L}$, we determine the directly-follows relations for individual objects. This relation hold for two events $(a_i, O_i), (a_j, O_j) \in L$ and the object $o \in O_i \cap O_j$ if $i < j$ and $\forall i < k < j : o \notin O_k$. We write $(a_i, O_i) \twoheadrightarrow_{L,o} (a_j, O_j)$ in this case. Based on this relation, we construct a graph with nodes for each event of L and, potentially multiple, directed edges between them for each object for which this follows relation holds. Then we filter the edges in the graph based on the following criteria. An edge $(a_i, O_i) \twoheadrightarrow_{L,o} (a_j, O_j)$ is removed if the activities are in different parts of the loop cut, i.e. $a_i \in \Sigma_{body} \wedge a_j \in \Sigma_{redo} \vee a_i \in \Sigma_{redo} \wedge a_j \in \Sigma_{body}$. Edges are also removed, if they connect start activities with end activities of the same object type, and that type is exclusively related to activities in only one of the partition parts. Formally, these are the edges $(a_i, O_i) \twoheadrightarrow_{L,o} (a_j, O_j)$ for which $a_i \in end_{\omega(o)}(\mathbb{L}) \wedge a_j \in start_{\omega(o)}(\mathbb{L})$ and $(\forall b \in \Sigma_{body} : \omega(o) \notin rel_b) \vee (\forall b \in \Sigma_{redo} : \omega(o) \notin rel_b)$ holds. The removed edges are exactly the behavior that is induced by the loop operator, leaving us with only the behavior induced by the individual loop iterations. Then, we treat the remaining graph as undirected and detect all sets of events that correspond to a connected components in it. The events in each component are then ordered by their position in the original log, resulting in the spitted logs $L_1, \dots, L_n$. Based on whether the split logs contain activities from Σ_{body} or Σ_{redo} we sort each of them into the resulting sets $\mathbb{L}_{body}$ and $\mathbb{L}_{redo}$ respectively. Algorithm 3 shows the splitting strategy. Note that τ-constructs are covered by treating them as cuts with one empty partition part, i.e. $(\times, \Sigma, \emptyset)$ for optional behavior and $(\circlearrowleft, \Sigma, \emptyset)$ for loops with empty redo parts.

As soon as only a single activity a is left in the input logs $\mathbb{L}$, no further cutting is possible. Then, the recursion base case is returned, i.e. the leaf node $(a, rel_a, def_a, con_a, div_a \cup \{\omega(o) \mid \exists L \in \mathbb{L} : |\{j \mid o \in O_j\}| \neq 1\})$.

Algorithm 2: OCIM τ-Construct Detection

Function $\text{TAUCASE}_{oc}(\mathbb{L}, \mathbb{O})$
- **for** $o \in \mathbb{O}$ **do**
  - **if** $\forall L \in \mathbb{L} : \forall (a_i, O_i) \in L : o \notin O_i\}$ **then**
    - **return** $\times, \{o \mid \exists L \in \mathbb{L} : \exists (a_i, O_i) \in L : o \in O_i \wedge \exists a \in \Sigma : \omega(o) \in rel_a\}$
- **for do**
  - **if** $\forall_{ot \in \Omega} : \forall_{a \in start_{ot}(\mathbb{L})} : \forall_{b \in end_{ot}(\mathbb{L})} : b \twoheadrightarrow_{\mathbb{L},ot} a$ **then**
    - **if** $\forall_{a,b \in \Sigma} : \forall_{ot \in rel_a \cap rel_b} : a \twoheadrightarrow^{+}_{\mathbb{L},ot} b \wedge b \twoheadrightarrow^{+}_{\mathbb{L},ot} a$ **then**
      - **if** $\exists_{a \in \Sigma} : \exists_{ot \in \Omega} : ot \in rel_a \setminus div_a$ **then**
        - **return** $\circlearrowleft, \mathbb{O}$

Algorithm 3: OCIM Log Splitting

Function $\text{CHECK}_{oc}(\Sigma_{body}, \Sigma_{redo}, i, j, o)$
- **if** $i > j \vee o \notin O_i \cap O_j$ **then**
  - **return** False
- **if** $(a_i \in \Sigma_{body} \wedge a_j \in \Sigma_{redo}) \vee (a_i \in \Sigma_{redo} \wedge a_j \in \Sigma_{body})$ **then**
  - **return** False
- **if** $a_i \in end_{\omega(o)}(\mathbb{L}) \wedge a_j \in start_{\omega(o)}(\mathbb{L})$ **then**
  - **if** $\forall a \in \Sigma : ot \in rel_a \implies a \in \Sigma_{body}$ **then**
    - **return** False
  - **if** $\forall a \in \Sigma : ot \in rel_a \implies a \in \Sigma_{redo}$ **then**
    - **return** False
- **for** $k \in i \dots j$ **do**
  - **if** $o \in O_k$ **then**
    - **return** False
- **return** True

Function $\text{SPLITLOG}_{oc}(\mathbb{L}, \oplus, \Sigma_1, \dots \Sigma_n)$
- **if** $\oplus \in \{\|, \times, \rightarrow\}$ **then**
  - **for** $i \in 1 \dots |\mathbb{L}|$ **do**
    - $\mathbb{L}_i = \{\langle (a, O) \in L \mid a \in \Sigma_i \rangle \mid L \in \mathbb{L}\}$
  - **return** $\mathbb{L}_1 \dots \mathbb{L}_n$
- $\mathbb{L}_{body}, \mathbb{L}_{redo} \leftarrow \emptyset, \emptyset$
- **for** $L \in \mathbb{L}$ **do**
  - $edges \leftarrow \{(i, j) \mid \exists_o : (\text{CHECK}_{oc}(\Sigma_1, \Sigma_2, i, j, o) \vee \text{CHECK}_{oc}(\Sigma_1, \Sigma_2, j, i, o))\}$
  - $E_1, \dots E_n \leftarrow \text{CONNECTEDCOMPONENT}(\Sigma, edges)$
  - **for** $i \in 1 \dots n$ **do**
    - $L_i = \langle (a_j O_j) \mid (a_j, O_j) \in L \wedge j \in E_i \rangle$
    - **if** $\exists (a_j, O_j) \in L_i : a_i \in \Sigma_1$ **then**
      - $\mathbb{L}_{body} \leftarrow \mathbb{L}_{body} \cup \{L_i\}$
    - **else**
      - $\mathbb{L}_{redo} \leftarrow \mathbb{L}_{redo} \cup \{L_i\}$
- **return** $\mathbb{L}_{body}, \mathbb{L}_{redo}$

4. Discovery guarantees & complexity

In this section, we discuss the formal guarantees of our approach. First, we prove that every set of object-centric input logs fits the resulting object-centric process tree. Then, we show that under certain conditions, rediscovery can be guaranteed. Lastly, we present the complexity of our approach.

4.1. Fitness guarantee

We prove the fitness of the discovered object-centric process tree for every set of object-centric input logs by induction. We first show fitness for all logs that reach the base case. Then, we prove that fitness is preserved by all recursion steps. By induction, this guarantees fitness of the discovered tree.

4.1.1. Base cases

Once a recursion base case is reached, only a single activity is left in the input logs. The resulting base case leaf node is always fitting.

Lemma 1. *A set of object-centric logs $\mathbb{L}$ with $|\Sigma| = 1$ is always included in the language of the respective OCIM base case: $\forall L \in \mathbb{L} : L \in \mathcal{L}(\text{BASECASE}(\mathbb{L}))$*

The resulting leaf node in the object-centric process tree allows arbitrary executions of the respective activity and only prohibits divergence, convergence and deficiency for those object types for which these patterns are not observed in the input logs. As a result, the object-centric input logs are in the language of the corresponding leaf and as such fitting, proving Lemma 1.

4.1.2. Log splitting

All recursion steps that are not a base case cause a splitting of the input logs. First, we show, that our log splitting method preserves events:

Lemma 2. *Let $\mathbb{L}$ be a set of object-centric logs and $\oplus, \Sigma_1, \dots, \Sigma_n$ a cut or fallthrough for them. Additionally, let $\mathbb{L}_1 \dots \mathbb{L}_n = \text{SPLITLOG}(\mathbb{L}, \oplus, \Sigma_1, \dots \Sigma_n)$ be the sublog sets after splitting. Then, for each input log in $\mathbb{L}$, there exists a combination of sublogs that contain the exact same events, i.e. $\forall L \in \mathbb{L} : \exists \mathbb{L}' \subseteq \cup_{1 \leq i \leq n} \mathbb{L}_i : [(a, O) \mid (a, O) \in L] = [(a, O) \mid \exists L' \in \mathbb{L}' : (a, O) \in L']$.*

According to the log splitting shown in Algorithm 3, every event in the object-centric input logs is assigned to exactly one sublog. In case of the operators $\{\rightarrow, \times, ||\}$ this assignment is based on the activities of the events, which can only be in one partition part of the cut. In case of the loop operator, the application of connected components during the log splitting ensures that all events, represented as nodes, are assigned to exactly one component and subsequently to exactly one sublog. As a result, the combination of all sublogs contains exactly the events of all input logs. Furthermore, each sublog only contains events from exactly one input log. Therefore, for each input log, there is a combination of sublogs that contain the same events, proving Lemma 2. Furthermore, we can show that the splitting not only preserves the events, but also the order between them. That is, for every sublog that results from the log splitting, the contained events have the exact same order as they had in their input log they. As such, the original logs are an interleaving of a combination of sublogs after splitting.

Lemma 3. *Let $\mathbb{L}$ be a set of object-centric logs, $\oplus, \Sigma_1, \dots, \Sigma_n$ a cut or fallthrough for them and $\mathbb{L}_1 \dots \mathbb{L}_n = \text{SPLITLOG}(\mathbb{L}, \oplus, \Sigma_1, \dots \Sigma_n)$ the sublog sets after splitting. Then, for each input log, there is a sublog interleaving that includes the input log: $\forall L \in \mathbb{L} : \exists \mathbb{L}' \subseteq \cup_{1 \leq i \leq n} \mathbb{L}_i : L \in \sqcup_{L' \in \mathbb{L}'} L'$*

Regarding the log splitting in Algorithm 3, this is obvious, since every sublog is the result of a filtering operation that does not include any reordering of events. In combination with Lemma 2, we can conclude that every original input log can be reconstructed by interleaving a combination of sublogs, proving Lemma 3. Note that certain properties of $\mathbb{L}'$ are guaranteed by the operator specific input log splitting. That is, we know that for any operator in $\{\rightarrow, \times, ||\}$, there exists an $\mathbb{L}'$ with exactly one sublog from each split $|\mathbb{L}' \cap \mathbb{L}_i| = 1$. For the loop operator, multiple sublogs from each split might be included in each $\mathbb{L}'$ due to potentially multiple loop iterations.

4.1.3. Cut fitness

Next we show that for all recursion steps that result in a cut, the input logs are fitting the resulting tree, if all sets of sublogs after splitting are fitting their corresponding subtree.

Lemma 4. *Let $\mathbb{L}$ be a set of object-centric input logs with the objects $\mathbb{O}$. Let $\oplus, \Sigma_1, \dots, \Sigma_n$ be a cut for them and $\mathbb{L}_1 \dots \mathbb{L}_n = \text{SPLITLOG}(\mathbb{L}, \oplus, \Sigma_1, \dots \Sigma_n)$ the corresponding sublog sets after splitting. Assuming that sublogs fit their subtree $\forall_{1 \leq i \leq n} : \forall L_i \in \mathbb{L}_i : L_i \in \mathcal{L}(OCIM(\mathbb{L}_i, \mathbb{O}))$, it guarantees $\forall L \in \mathbb{L} : L \in \mathcal{L}((\oplus, OCIM(\mathbb{L}_1, \mathbb{O}), \dots, OCIM(\mathbb{L}_n, \mathbb{O})), \mathbb{O})$.*

Considering the language definition for object-centric process trees, each operator node first interleaves all possible combinations of sublogs in the language of the subtrees. According to Lemma 3, we know that all input logs are in those interleavings. Then, out of these interleavings, those that violate the operator are disallowed. However, if a cut is detected, we know by definition that the input logs do not contain any of the behavior disallowed by the operator. Our detection strategy is guaranteed to only find correct cuts, as discussed in the in-depth discussion of these techniques in Appendix 8. Together with Lemma 3, this guarantees the input logs to fit the operator node, if each sublog fits its subtree, proving Lemma 4.

4.1.4. Fallthroughs fitness

Similar to the fitness guarantee for the cut detection in the previous section, fallthroughs also preserve fitness throughout the recursion.

Lemma 5. *Let $\mathbb{L}$ be a set of input logs with the objects O. Let $\oplus, \Sigma_1, \dots, \Sigma_n$ a fallthrough for them and $\mathbb{L}_1 \dots \mathbb{L}_n = \text{SPLITLOG}(\mathbb{L}, \oplus, \Sigma_1, \dots \Sigma_n)$ the respective sublogs. Assuming that $\forall_{1 \leq i \leq n} : \forall L_i \in \mathbb{L}_i : L_i \in \mathcal{L}(OCIM(\mathbb{L}_i, O))$, it guarantees $\forall L \in \mathbb{L} : L \in \mathcal{L}((\oplus, OCIM(\mathbb{L}_1, O), \dots, OCIM(\mathbb{L}_n, O)), O)$.*

When comparing the definition of cuts and fallthroughs, it is clear that fallthroughs preserve exactly those cut conditions that disallow behavior. The conditions that are relaxed, all refer to required behavior, which does not influence model fitness. As such, the fitness proof for cuts from the previous section directly extends to fallthroughs, proving Lemma 5.

4.1.5. Additional construct fitness

Steps with τ-constructs can result in optional behavior or loops with empty redo parts. For optional behavior, fitness is preserved:

Lemma 6. *Let $\mathbb{L}$ be a set of input logs that include the object $\mathbb{O}'$ and let $\mathbb{O}$ be any superset with $\mathbb{O}' \subset \mathbb{O}$. If $\forall L \in \mathbb{L} : L \in \mathcal{L}(OCIM(\mathbb{L}, \mathbb{O}'), \mathbb{O}')$, it is guaranteed that $\forall L \in \mathbb{L} : L \in \mathcal{L}(\times, OCIM(\mathbb{L}, \mathbb{O}'), (\tau, \Omega, \Omega, \Omega, \Omega), \mathbb{O})$.*

Technically, in case of optional behavior, the set of objects in the input logs is smaller than the set of objects observed in the previous recursion steps. The corresponding log splitting passes all input logs without changes down into another recursion step and removes the missing objects from the observation set. As such, the language of the tree before and after this recursion step is exactly the same, except that additional objects can be included by arbitrary τ events. The input logs, with any extended set of previously observed objects, therefore fit the resulting exclusive choice operator node, if the input logs with the remaining objects fit the non-empty subtree, proving Lemma 6. For the second τ-construct of loops with empty redo parts, fitness is preserved as well.

Lemma 7. *Let $\mathbb{L}$ be a set of object-centric event logs with the objects $\mathbb{O}$. Let $\circlearrowleft, \Sigma, \emptyset$ be a τ-loop for them and $\mathbb{L}_1, \mathbb{L}_2 = \text{SPLITLOG}(\mathbb{L}, \circlearrowleft, \Sigma, \emptyset)$ the corresponding sublogs. Assuming that $\forall L_1 \in \mathbb{L}_1 : L_1 \in \mathcal{L}(OCIM(\mathbb{L}_1, \mathbb{O}))$, it is guaranteed that $\forall L \in \mathbb{L} : L \in \mathcal{L}(\circlearrowleft, OCIM(\mathbb{L}_1, \mathbb{O}), (\tau, \Omega, \Omega, \Omega, \Omega), \mathbb{O})$.*

For loops with empty redo parts, the proof for the fitness preservation of loop cuts applies. All behavior that could be limited in its fitness by the resulting operator node is prohibited by the corresponding conditions in Algorithm 2. Note that for empty redo parts, all sublogs in $\mathbb{L}_2$ are empty.

4.1.6. Proof by induction

Based on previous conclusions in this section, we can prove the construction of a fitting object-centric process tree by OCIM.

Theorem 1. *Let $\mathbb{L}$ be a set of object-centric input logs. Then, these logs are fitting the constructed object-centric process tree $\forall L \in \mathbb{L} : L \in \mathcal{L}(OCIM(\mathbb{L}))$.*

For any set of input logs, there is a finite number of recursion steps, since τ-constructs cannot occur indefinitely after each other and each cut and fallthrough reduces the alphabet size. Each of these recursion steps guarantees fitness, as long as all subsequent recursion steps guarantee fitness as well. Lemma 4 shows this for cuts, Lemma 5 for fallthroughs, Lemma 6 for optional behavior and Lemma 7 for loops with empty redo parts. Each of these recursion steps inevitably leads to a base case in finitely many steps. This base case is guaranteed to be fitting, irrespective of the set of object-centric input logs according to Lemma 1. As a result, the discovered object-centric process tree always fits, proving Theorem 1.

4.2. Rediscovery guarantee

Next, we show that under certain assumptions, our approach is able to rediscover a model that is language equivalent to the process that

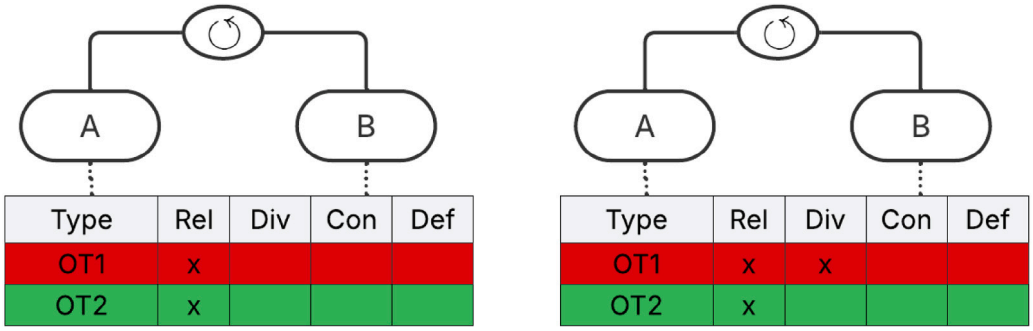


Fig. 13. Two object-centric process tree with different leaf configuration and different languages but the same abstraction properties with regards to object type divergence.

Table 6
Object-centric event log in the language of the left tree in Fig. 13. Note how for each loop iteration, there is no divergence, since the leaf specification does not allow it. However, in the overall log. across loop iterations, divergence occurs.

Event #	Activity	OT1	OT2
1	A	o_{11}	o_{21}
2	B	o_{11}	o_{21}
3	A	o_{12}	o_{22}
4	B	o_{12}	o_{22}
5	A	o_{12}	o_{21}
6	B	o_{12}	o_{21}
7	A	o_{11}	o_{22}
8	B	o_{11}	o_{22}
9	A	o_{11}	o_{21}
10	A	o_{12}	o_{22}

generated the input logs. For this purpose, we first define a subclass of object-centric process trees. Then, we introduce a syntactical normal form for this subclass and show that the language of any object-centric process tree in that normal form has a unique language abstraction with regards to the divergence, convergence, deficiency and directly-follows relations of object types. Lastly, we show that our approach is guaranteed to construct an object-centric process tree with an abstraction that is equivalent to the input logs. As a result, if the input logs are generated by an object-centric process tree within our normal form and complete and correct with regards to the utilized abstraction, rediscovery is guaranteed.

4.2.1. Model class

In the following we only consider the subclass of object-centric process trees that does not include any loop operator, as it impedes rediscovery. The key reason for that exclusion is that it is not possible to differentiate between shared looped behavior of all object types in which the identities of objects are not synchronized and divergence of the respective leafs. Fig. 13 shows two object-centric process trees to illustrate the problem. The tree on the left side does not allow divergence in the leaf nodes below the loop operator. However, since the loop iterations are independent of each other and can be merged, as long as each individual object follows the loop structure, the log in Table 6, for example, is in the language of that tree. Therefore, the divergence of object types will be observed in the trees language, despite the leaf nodes not allowing it for individual loop iterations. In contrast, the tree on the right side does allow divergence in the leaf nodes, which will hence also be observed in its language. As a result, these structures can lead to two trees with different languages having the same abstraction properties, impeding the rediscovery under the presence of the loop operator. Future work might address this problem by adding synchronizations between objects based on their identities to the representational bias of object-centric process trees to differentiate between the two scenarios, as proposed in [15,16]. Additionally, we assume each leaf node to have a unique label unequal to τ and at least one related object type that does not diverge. The work in [15] showed that if all object types diverge on an activity, the absence of relevant objects is likely. Lastly, we assume the leaf specification to reflect the convergence, divergence and deficiency correctly. That is, if divergence, convergence and deficiency are allowed by a leaf node, a log exists in the language of the tree that reflects this pattern. The strategy proposed in [17] can be used to check this in polynomial runtime complexity and adjust the object-centric process tree accordingly if needed.

4.2.2. Reduction rules

Given any object-centric process tree in the previously defined subclass, we introduce a set of reduction rules that can be recursively applied to achieve a syntactical normal form. These rules can be grouped into three categories with regards to the type of structure they address. The first category refers to object-centric process trees in which operator nodes have subtrees with the same operator. In these cases, the nodes can be merged, if the diverging object types do not cause any change in the language of the tree.

For the concurrent operator the reduction can always be done (57). We refer to this reduction rule as the concurrent operator associativity (**R1**).

$$\begin{aligned} S = \ &||\,(S_1 \dots S_n) \wedge \exists_{1\le i\le n} : S_i = \ ||\,(S'_1 \dots S'_m) \wedge n > 1 \\ &\implies \mathcal{L}(S) = \mathcal{L}(||\,(S_1 \dots S_n, S'_1 \dots S'_m)) \end{aligned} \tag{57}$$

For the exclusive choice operator, merging nodes might disrupt the behavior of object types that were previously fully divergent on some subtrees. To ensure this does not happen, the reduction can only be performed if the diverging types of the subtree agree with the diverging types of the higher level node (58). We refer to this rule as the choice operator associativity (**R2**).

$$\begin{aligned} S = \ &\times(S_1 \dots S_n) \wedge \exists_{1\le i\le n} : S_i = \ \times(S'_1 \dots S'_m) \wedge D(S_i) \cap D(S) = D(S_i) \\ &\wedge n > 1 \implies \mathcal{L}(S) = \mathcal{L}(\times(S_1 \dots S_n, S'_1 \dots S'_m)) \end{aligned} \tag{58}$$

For the sequence operator, the divergence of object types is sensitive to the order of the subtrees. As a result, the reduction is only allowed if the diverging object types of the lifted subtrees fit to the diverging object types of their new neighbors within the order of the higher level operator node (59). We refer to this rule as the sequence operator associativity (**R3**).

$$\begin{aligned} &S = \rightarrow(S_1 \dots S_n) \wedge \exists S_i = \rightarrow(S'_1 \dots S'_m) \wedge (\forall_{1\le j\le m} : D(S'_j) \cap D(S_i) \\ &= D(S'_j)) \wedge n > 1 \implies \mathcal{L}(S) = \mathcal{L}(\rightarrow(S_1 \dots S_{i-1}, S'_1 \dots S'_m, S_{i+1} \dots S_n)) \end{aligned} \tag{59}$$

The second category of reduction rules refers to operator nodes with subtrees that are independent because they do not share any related and non-diverging object types. In these case, the operator of the tree can be freely chosen. In case of the root node of the tree, we select the concurrent operator by convention. In case of non-root nodes, we choose the operator of the node in the level above, to potentially enable the application of the previously introduced rules. We refer to this reduction rule as the reduction of independent subtrees (**R4**).

$$\begin{aligned} &S = \oplus_1(S_1 \dots S_n) \wedge \exists_{1 \le i \le n} : S_i = \oplus_2(S'_1 \dots S'_m) \\ &\wedge \oplus_1 \neq \oplus_2 \wedge (\forall_{1 \le j \neq k \le m} : R(S'_j, S'_k) \setminus D(S'_j, S'_k) = \emptyset) \wedge n > 1 \\ &\implies \mathcal{L}(S) = \mathcal{L}(\oplus_1(S_1 \dots S_{i-1}, \oplus_1(S'_1 \dots S'_m), S_{i+1} \dots S_n)) \end{aligned} \tag{60}$$

The third category summarizes rules for redundant structures, such as nodes with only a single subtree. In these cases, the operator node can be completely removed (61), reducing the number of operator nodes (**R5**).

$$\begin{aligned} &S = \oplus(S_1 \dots S_n) \wedge \exists_{1 \le i \le n} : S_i = \oplus(S'_i) \\ &\implies \mathcal{L}(S) = \mathcal{L}(\oplus(S_1 \dots S_{i-1}, S'_i, S_{i+1} \dots S_n)) \end{aligned} \tag{61}$$

Additionally, we assume an ordering function to be present that provides a deterministic order for subtrees in cases where they can be arbitrarily ordered (**R6**). This is always the case for the concurrent and exclusive choice operator. For the sequence operator it can only occur when two subsequent subtrees do not share any non-diverging object type. In these cases, we apply the deterministic ordering function for the construction of the tree's normal form. By definition rule **R6** can be applied if none of the subtrees can be lifted (**R1, R2, R3**) or its operator adjusted (**R4**) and at least two subtrees are not correctly ordered yet.

4.2.3. Syntactical normal form

Next we show that the repeated application of the reduction rules (**R1**) - (**R6**) on any object-centric process tree in our subclass always terminates.

Lemma 8. *Given an object-centric process tree S, the application of the rules **R1, R2, R3, R4, R5, R6** terminates in finitely many steps.*

Lemma 8 can easily be proven, as most rules (**R1**, **R2**, **R3**, **R5**) reduce the number of nodes, ensuring termination in finitely many steps. All other rules can only be applied finitely many times per operator node (**R4, R6**).

Besides the termination itself, we can also guarantee that all applications of the reduction rules produce the same result, irrespective of the order in which they might be applied. For that purpose, we show for all pairs of reduction rules that they can either not both be applied to the same tree, or that they can both be applied in any order while leading to the same result.

Lemma 9. *Given an object-centric process tree S, the exhaustive application of the reduction rules **R1, R2, R3, R4, R5, R6**, irrespective or the order, always leads to the same object-centric process tree in normal form.*

Technically, we prove Lemma 9 by showing that all pairs of rules are locally confluent, i.e. they lead to the same result in both orders. First, we identify which pairs of rules can never be applied to the same nodes. Those are all combinations between **R1, R2, R3** as they require different operators. Additionally, rule **R5** requires operator nodes with only one subtree, meaning that it cannot be applied to the same nodes as any other rule. **R6** can by definition not be applied to the same nodes as **R1 - R5**.

For the remaining pairs of rules, we determine the structure of trees that match the input requirements of both rules. For those cases, we show that the order in which they are applied is irrelevant and leads to the same result. Table 7 shows the simultaneous or exclusive applicability of all rules.

Reduction rule **R4** can be applied to an operator node that also enable **R1**, **R2** or **R3** respectively, depending on the operator of the node. For the reduction rule **R1**, this respective node must be of the structure shown in (62), as all deviations from that structure impede the application of either **R4** or **R1**. In this case, the reduction rule **R1** can be applied to the subtree S_j, while rule **R4** is applicable to S_i. However, both application orders produce the same result.

Table 7
Simultaneous (✓) or exclusive (⊘) applicability of reduction rules.

	R1	R2	R3	R4	R5	R6
R1	–	⊘	⊘	✓	⊘	⊘
R2	⊘	–	⊘	✓	⊘	⊘
R3	⊘	⊘	–	✓	⊘	⊘
R4	✓	✓	✓	–	⊘	⊘
R5	⊘	⊘	⊘	⊘	–	⊘
R6	⊘	⊘	⊘	⊘	⊘	–

$$\begin{aligned} &S = \oplus_1(S_1 \dots S_n) \wedge \exists_{1 \le i \ne j \le n} : S_i = \oplus_2(S'_1 \dots S'_m) \wedge S_j = \oplus_1(S''_1 \dots S''_l) \\ &\wedge \oplus_1 = \,||\, \wedge \oplus_1 \neq \oplus_2 \wedge (\forall_{1 \le o \ne p \le m} R(S'_o, S'_p) \setminus D(S'_o, S'_p) = \emptyset) \end{aligned} \tag{62}$$

If **R1** is applied first, S_i will be merged into the higher level node without changing S_j or the operators $\oplus_1, \oplus_2$. As such, **R4** remains enabled, leading to the structure in (63). The application of the two rules in the reverse order, results in the same structure, showing local confluence of **R1** & **R4**.

$$\oplus_1(S_1 \dots S_{i-1}, \oplus_1(S'_1 \dots S'_m), S_{i+1} \dots S_{j-1}, S''_1, \dots, S''_l, S_{j+1} \dots S_n) \tag{63}$$

For the combination of the reduction rule **R4** with **R2** and **R3** we have analogous cases. We can again deduct the structure of the trees that are applicable to both rules at the same time. The structures in (64) and (65) are applicable to these pairs of reduction rules. Similarly to the previously discussed case, the order of application does not matter, as it results in the same structure (63). As such, we conclude the local confluence of **R2** & **R4** and **R3** & **R4**.

$$\begin{aligned} &S = \oplus_1(S_1 \dots S_n) \wedge \exists_{1 \le i \ne j \le n} : S_i = \oplus_2(S'_1 \dots S'_m) \\ &\wedge S_j = \oplus_1(S''_1 \dots S''_l) \wedge \oplus_1 = \times \wedge \oplus_1 \neq \oplus_2 \wedge D(S_j) \cap D(S) = D(S_j) \\ &\wedge (\forall_{1 \le o \ne p \le m} R(S'_o, S'_p) \setminus D(S'_o, S'_p) = \emptyset) \end{aligned} \tag{64}$$

$$\begin{aligned} &S = \oplus_1(S_1 \dots S_n) \wedge \exists_{1 \le i \ne j \le n} : S_i = \oplus_2(S'_1 .. S'_m) \wedge S_j = \oplus_1(S''_1 .. S''_l) \\ &\wedge \oplus_1 = \rightarrow \wedge \oplus_1 \neq \oplus_2 \wedge (\forall 1 \le o \le m : D(S'_o) \cap D(S_j) = D(S'_o)) \\ &\wedge (\forall_{1 \le p \ne q \le m} R(S'_p, S'_q) \setminus D(S'_p, S'_q) = \emptyset) \end{aligned} \tag{65}$$

As a result of the local confluence and exclusivity of all rule pairs and the termination of their applicability, we conclude that their exhaustive repeated application leads to the same result, in any order, proving Lemma 9.

4.2.4. Unique language abstraction

Each object-centric process trees in the normal form of the investigated subclass is unique with regards to its language abstraction, consisting of the combination of allowed directly-follows relations and the convergence, divergence and deficiency of object types. Every syntactical difference between two object-centric process trees in normal form corresponds to a difference in the abstraction. As a result, object-centric process trees in their normal form always have a unique language.

Assuming two object-centric process trees S, S' in normal form with the same language, i.e. $\mathcal{L}(S) = \mathcal{L}(S')$. Then these two trees also have the same language abstraction, written as $\mathcal{A}(\mathcal{L}(S)) = \mathcal{A}(\mathcal{L}(S'))$. We show by contradiction that the languages and abstractions of two object-centric process trees in normal form are equivalent if and only if they are syntactically identical. We first show that both trees must have the same leaf nodes with identical configurations

Lemma 10. *Let S', S be object-centric trees in normal form with $\mathcal{A}(\mathcal{L}(S)) = \mathcal{A}(\mathcal{L}(S'))$. Then, the set of leaves and their configuration is identical for both.*

We prove Lemma 10 by contradiction. Assuming that S contains a set of leaf nodes and $\mathcal{L}(S) = \mathcal{L}(S')$ then S' must contain the same set of leaf nodes as well. Otherwise, the alphabet of the language of the two trees would differ. Similarly, those leaf nodes must have the exact same combination of divergent, convergent and deficient types per activity, as those elements of the abstraction would otherwise be different. Next, we show that any difference in the choice of operators or the order and elements of the partitions of activities across subtrees leads to a change in the allowed directly follows relations of the language abstraction.

Lemma 11. *Let $S = \oplus, (S_1, \dots, S_n)$ and $S' = \oplus', (S'_1, \dots, S'_m)$ be two trees in normal form with $\mathcal{A}(\mathcal{L}(S)) = \mathcal{A}(\mathcal{L}(S'))$. Then $\oplus = \oplus' \wedge n = m$.*

Assuming that $S = \oplus(S_1, \dots, S_n)$ and $S' = \oplus'(S'_1 \dots S'_m)$ and $\mathcal{A}(\mathcal{L}(S)) = \mathcal{A}(\mathcal{L}(S'))$, it follows that $\oplus = \oplus'$ as the operators are mutually exclusive with regards to the allowed directly-follows relations. Considering that the directly-follows edges allowed by an operator are, by design, those that cuts require, it is trivial to see that they exclude each other pairwise. Note that fully divergent behavior of all object types is disallowed in the investigated subclass of object-centric process trees for this very reason. In case of exclusively diverging behavior, the original operator would not be deductible, as it does not influence the language of the tree. Additionally, we can show that any change in the order or the partition of activities across subtrees for an object-centric process tree in normal form results in an abstraction change.

Lemma 12. *Let $S = \oplus, (S_1, \dots, S_n)$ and $S' = \oplus', (S'_1, \dots, S'_m)$ be two trees in normal form with $\mathcal{A}(\mathcal{L}(S)) = \mathcal{A}(\mathcal{L}(S'))$. Then, the leafs of both tree are identically distributed across $S_1, \dots, S_n$ and $S'_1, \dots, S'_m$ with $m = n$.*

Assuming $S = \oplus(S_1, \dots, S_n)$ and $S' = \oplus'(S'_1 \dots S'_m)$ and $\mathcal{A}(\mathcal{L}(S)) = \mathcal{A}(\mathcal{L}(S'))$, we can show that the activities of the leaf nodes in each subtree have to be identical. According to Lemma 10 we already now that both trees contain the same leaf nodes. As such, only the partition across the subtrees and the order between them remains to be considered. If both trees contain the same partition in a different order, then one of them cannot be in normal form as rule **R6** would be applicable. Assuming that the partition itself is changed, the language of the tree would be different, except for activities that are unrelated to all other activities in that subtree. However, if this is the case, rule **R4** would be applicable to the corresponding subtree, which would hence not be in normal form. Therefore, and based on the previous three lemmas, we can guarantee that every syntactically unique object-centric process tree in normal form also has a unique language.

Lemma 13. *Let S, S' be object-centric process trees in normal form. They have the same language and language abstraction if and only if $S = S'$.*

Lemma 10 shows this for single leaves, while Lemma 11 and Lemma 12 ensure the preservation of the activities across subtrees and the choice of operators. This applies inductively to all nodes in a tree, as the allowed directly-follows relations for each type can only grow when going up the tree conceptually. As a result, the language for trees in normal form is unique.

4.2.5. Abstraction preserving discovery

Next, we show that if an object-centric process tree in the investigated subclass is discovered by OCIM without the use of fallthrough methods, the language abstraction of the resulting tree is equivalent to the language abstraction of the input logs and vice versa. As a result, if the input logs are complete and correct with regards to their language abstraction, the normal form of the tree that generated them is rediscovered.

We construct this proof in three steps. First, we show that if the input logs are generated by an object-centric process tree in normal form, the corresponding cut is detected by OCIM. Then, we proof that the corresponding log splitting removes exactly the directly-follows relations from the input logs that the detected operator allows. As a result, the directly-follows relations of the input logs are correctly preserved throughout the recursion. Lastly, we show that the interaction properties of convergence, divergence and deficiency are also preserved throughout the recursion. Therefore, the leaves of the tree will be configured correctly and the language abstraction of the tree will be identical to the language abstraction of the input logs and the tree that generated them.

Lemma 14. *Let $\mathbb{L}$ be a set of input logs that are in the language of the object-centric process tree in normal form $S = \oplus, (S_1, \dots, S_n)$. Additionally, let $\Sigma_1 \dots \Sigma_n$ be the activities in $S_1 \dots S_n$ and let the language abstraction of S be fully represented in $\mathbb{L}$. Then, the cut $\oplus, \Sigma_1 \dots \Sigma_n$ is detected.*

Let $\mathbb{L}$ be a set of object-centric input logs in the language of an object-centric process tree $S = \oplus(S_1 \dots S_n)$ in normal form that is complete with regards to the divergence, convergence deficiency and directly-follows relations of all object types. Additionally, let $\Sigma_1, \dots \Sigma_n$ be the activity labels of the leaves in each subtree $S_1 \dots S_n$ respectively. As the input logs are complete with regards to the directly-follows relation, a cut $\oplus, \Sigma_1, \dots \Sigma_n$ exists. As the tree is in normal form, there can be no cut with more partition parts, since otherwise one of the reduction rules **R1, R2, R3** would be applicable. As the detection methods for cuts guarantee the detection of maximal cuts if they exists for one of the operators, as discussed in Appendix 8, it is guaranteed that the cut $\oplus, \Sigma_1 \dots \Sigma_n$ will be found. Note again that the operators for cuts are mutually exclusive. After such a cut is detected, the subsequently applied log splitting correctly preserves the language abstraction.

Lemma 15. *Let $\mathbb{L}$ be a set of input logs that are in the language of the object-centric process tree in normal form $S = \oplus, (S_1, \dots, S_n)$. Additionally, let $\Sigma_1 \dots \Sigma_n$ be the activities in $S_1 \dots S_n$ and let the language abstraction of S be fully represented in $\mathbb{L}$. Then, the logs $\mathbb{L}_1 \dots \mathbb{L}_n$ based on the splitting with the cut $\oplus, \Sigma_1 \dots \Sigma_n$ have the same abstractions as $S_1 \dots S_n$ respectively.*

The language abstraction properties of divergence, convergence and deficiency are always accumulated across all input logs and hence do not change since the log splitting preserves the sum of all events (Lemma 2). The directly-follows relations removed from the input logs is preserved (Lemma 3), except for relations between different partition parts. As such, the abstraction of the split logs $\mathbb{L}_1, \dots \mathbb{L}_n$ is exactly the abstraction of the subtrees $S_1 \dots S_n$. Finally, we can conclude the rediscovery of abstraction complete input log sets.

Theorem 2. *Let $\mathbb{L}$ be a set of logs in the language of any object-centric tree S in normal form. If $\mathbb{L}$ is complete with regards to the language abstraction of S, it is guaranteed that S is rediscovered with $\mathcal{L}(OCIM(\mathbb{L})) = \mathcal{L}(S)$.*

By induction, all operator nodes of S will be correctly rediscovered. As the abstraction is preserved throughout the recursion, all leave nodes are configured correctly with regards to the object types that are allowed to exhibit divergence, convergence and deficiency. Operator nodes are rediscovered inductively as well, as shown by Lemmas 14 and 15. As a result, all object-centric process trees in the investigated subclass will be rediscovered from input logs that are correct and complete with regards to the used abstraction.

4.3. Runtime complexity

We show the theoretical runtime complexity of our approach, by determining the maximal number of recursion steps and the complexity of each of those steps. In total, the complexity is $\mathcal{O}(|\Sigma| \cdot (|L|^2 \cdot |\Theta| + |\Sigma|^2 \cdot |\Omega|))$

4.3.1. Recursion steps

Throughout the recursion, the alphabet size decreases until a base case is found. The only steps in which the alphabet size remains stable

is when a τ-construct is found. However, due to the log splitting, these constructs can never be found in subsequent recursion steps without a cut or fallthrough in between. Since each cuts or fallthrough reduces the alphabet size, the number of recursion steps is within the complexity $\mathcal{O}(|\Sigma|)$.

4.3.2. Auxiliary constructs

Each recursion step utilizes the directly follows relation and the properties of divergence, convergence and deficiency. The directly follows relation can be constructed by iterating through the event logs once per object, which results in a worst case complexity of $\mathcal{O}(|L| \cdot |\Theta|)$. The interaction properties of convergence and deficiency can be determined by iterating through the log once in $\mathcal{O}(|L|)$. Divergence requires a pairwise comparison between all events which share an object, which can be done in $\mathcal{O}(|L|^2 \cdot |\Theta|)$.

4.3.3. Additional construct detection

The check for optional behavior simply compares two object sets which can have a maximal runtime of $\mathcal{O}(|\Theta|)$. The detection of loops with empty redo parts iterates over all object types and all activity pairs in $\mathcal{O}(|\Sigma|^2 \cdot |\Omega|)$. Since we already accounted for the construction of all auxiliary constructs, not further runtime costs are caused by the detection of τ constructs.

4.3.4. Cut detection

The cut detection for each of the operators remains in polynomial runtime complexity. Each detection requires a pairwise comparison between all activities and object types in $\mathcal{O}(|\Sigma|^2 \cdot |\Omega|)$ Based on the resulting graph, the connected components are constructed, which can be done linearly in the number of nodes and edges in a graph. In our case, we have at most $|\Sigma|$ nodes and $|\Sigma|^2$ edges between them, resulting in a complexity of $\mathcal{O}(|\Sigma| + |\Sigma|^2)$. Between the operators there are only constant differences in runtime.

4.3.5. Fallthrough detection

The detection of fallthrough requires the creation of candidate partitions with the use of connected components, similar to the cut detection. As such, they share the associated runtime costs of $\mathcal{O}(|\Sigma| + |\Sigma|^2)$. Additionally, the fallthrough methods determine a clustering, which can be achieved in quadratic runtime of the number of elements, which in our case are the activities in the alphabet, i.e. $\mathcal{O}(|\Sigma|^2)$. Lastly, the precision estimates need to be determined, which iterate pairwise over all activities for every object type, resulting in a complexity of $\mathcal{O}(|\Sigma|^2 \cdot |\Omega|)$ for the fallthrough detection.

4.3.6. Log splitting

The log splitting requires the construction of a graph, based on a pairwise comparison between all events for each object in $\mathcal{O}(|L|^2 \cdot |\Theta|)$. Subsequently, the connected components are applied to the resulting graph with $|L|$ nodes and at most $|L|^2$ edges, resulting in a complexity of $\mathcal{O}(|L| + |L|^2)$.

4.3.7. Base case detection

The base case detection can be done in linear time, assuming that the interaction properties are already calculated. They only require one pass over the input logs to ensure the alphabet size of one in $\mathcal{O}(|L|)$.

5. Evaluation

In this section, we evaluate the runtime feasibility of our approach, the quality of the discovered process models and the suitability of the utilized representational bias. Additionally, we compare the obtained results to existing approaches, using a variety of real-life and synthetic object-centric event logs. Lastly, we verify the rediscovery guarantee of our approach on small artificial logs from known ground truth models and investigate its deterioration under incompleteness.

Table 8
Characteristics of utilized object-centric event logs.

ID	Name	Activities	Events	Objects	Types
01	Container Logistics	14	35 372	13 882	7
02	Order Management	11	21 008	10 840	6
03	Procure-To-Pay	10	14 671	9054	7
04	LRMS O2C	22	28 278	8819	9
05	LRMS P2P	12	8300	4493	6
06	LRMS HR	24	18 119	3768	6
07	LRMS Hospital	9	14 642	3688	8
08	Hinge Production	11	38 528	23 771	12
09	Recruiting	16	6980	1505	6
10	Order Transfer	3	10 319	2500	5
11	Order Management	11	22 367	11 521	5
12	PM4PY Github	20	1798	532	3
13	BPIC 2015 M1	289	52 217	1269	4
14	BPIC 2015 M2	304	44 354	859	4
15	BPIC 2015 M3	277	59 681	1465	4
16	BPIC 2015 M4	272	47 293	1084	4
17	BPIC 2015 M5	285	59 083	1202	4
18	Angular Repository	67	27 842	28 317	2
19	BPIC 2017 Loans	26	1 202 267	106 162	4
20	Windows Registry	337	98 128	404	6

Table 9
Total runtime in seconds and relative runtime distribution in percentages across relevant algorithmic steps for the object-centric input logs from Table 8.

ID	Total	Cuts	FTs	τ	Split	Auxiliary	Rest
01	7.78s	31.5%	0.8%	1.98%	23.44%	32.59%	9.66%
02	35.7s	17.6%	0.3%	0.4%	65.94%	7.55%	8.23%
03	37.59s	0.6%	0.1%	0.14%	86.36%	3.44%	9.41%
04	115.93s	23.5%	0.1%	0.1%	71.35%	2.13%	2.85%
05	61.32s	16.7%	0.1%	0.06%	79.25%	1.18%	2.67%
06	8.53s	61.6%	1.6%	0.67%	16.7%	15.39%	4.03%
07	1.95s	0.1%	0.0%	1.98%	12.89%	68.82%	16.23%
08	34.48s	66.2%	0.3%	1.04%	16.62%	12.16%	3.69%
09	3.97s	34.5%	1.4%	1.32%	39.32%	17.62%	5.8%
10	1.86s	20.5%	1.6%	0.72%	30.24%	35.64%	11.3%
11	46.7s	12.6%	0.1%	0.19%	69.92%	6.98%	10.29%
12	6.08s	25.0%	3.6%	0.53%	61.26%	5.21%	4.36%
13	1114.2s	31.4%	27.0%	0.39%	37.78%	1.3%	2.14%
14	604.68s	45.1%	32.2%	0.45%	18.33%	1.78%	2.06%
15	566.73s	46.8%	26.4%	0.55%	22.5%	1.81%	2.04%
16	2356.95s	19.7%	20.2%	0.25%	58.58%	0.34%	0.88%
17	1215.86s	31.8%	18.5%	0.27%	46.63%	1.35%	1.42%
18	4779.62s	14.7%	10.2%	0.15%	68.9%	0.45%	5.51%
19	483.29s	63.3%	0.1%	1.00%	20.66%	11.48%	3.49%
20	644.57s	86.8%	0.2%	0.98%	8.01%	3.1%	0.89%

We implemented our approach in python and run all experiments on a consumer grade laptop with 64 GB of RAM and an Intel Core I9-13980-HX processor. The object-centric event logs used throughout all experiments and their characteristics are shown in Table 8. They stem from the public library of the OCEL file format for object-centric event logs and the Business Process Intelligence Challenge 2015 and 2017. The implementation of our approach can be found here (https://github.com/Nik314/OCIM).

5.1. Computational feasibility

First, we applied our technique to all object-centric input logs and measured the runtime. To enable a detailed analysis, we tracked the runtime for each of the steps in OCIM separately, i.e. the construction of auxiliary constructs, the cut detection, the fallthrough detection, the τ-construct detection and the log splitting. Table 9 shows the results, including the implementation specific overhead that cannot be attributed to any of the mentioned steps.

The total runtime for the discovery of object-centric process trees ranged from two seconds to eighty minutes. The construction of auxiliary constructs had the most relevant effect on smaller event logs and

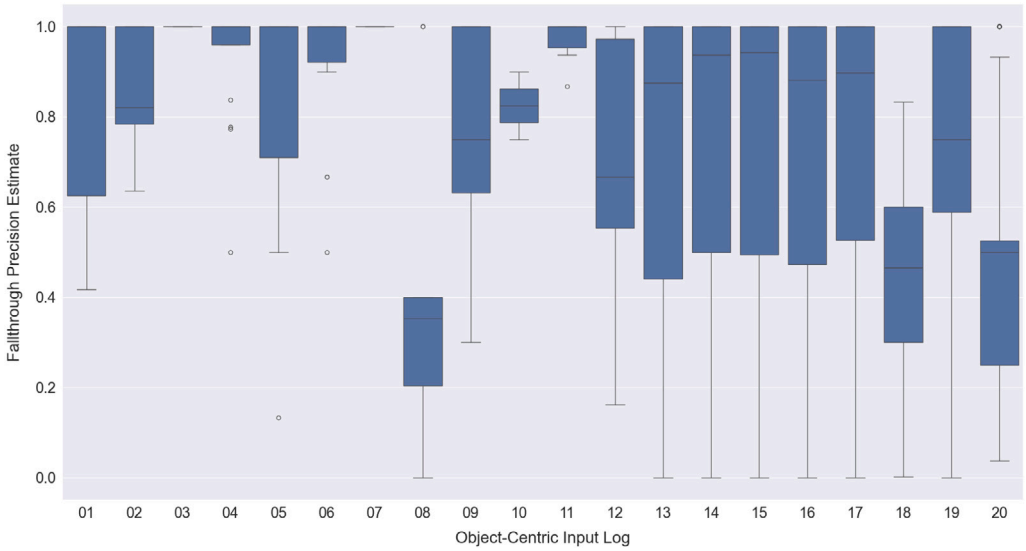


Fig. 14. Distribution of the precision estimates for cut and fallthroughs. Every cut has a score of 1.0, while all other values correspond to the estimates from Section 3.3.

became proportionally less relevant when the total runtime increased on larger logs. The implementation specific overhead was most significant for logs with lower total runtime as well. For the logs with longer overall runtimes, the cut detection and log splitting accounted for the majority of time being spent. This was to be expected, as both methods are performed in every recursion step except for the base cases. We inspected those logs for which the percentage of time being spent on the log splitting was the highest and noticed that this correlates with the detection of a loop cut and the application of the corresponding log splitting strategy. In contrast, the detection of τ-constructs and fallthroughs both had a negligible influence on the runtime for the majority of event logs. The runtime spent on the detection of both constructs consistently remained well below three percent of the overall runtime, for all logs except (13)–(18). This is an important insight, as it indicates that our approximation measures to avoid exponential runtime complexity by trying out all potential fallthroughs, is working well with regards to the intended runtime decrease. The detection of fallthroughs on the outlier logs (13)–(18) caused significant runtime contributions of up to thirty percent for those logs because of large alphabets that include up to four-hundred activities. In these case, the clustering applied for the fallthroughs lead to the runtime increase, with a runtime complexity that is quadratic in the alphabet size. Overall, the runtime demands of our approach on the investigated object-centric input logs were feasible and only showed scalability issues for extremely large alphabet sizes, with several hundreds of included activities.

5.2. Representational bias

Next, we investigate to which extent the object-centric input logs adhere to the utilized representational bias of the object-centric process trees. For this purpose, we track how often cuts or fallthroughs are detected. Cuts correspond to the representational bias of the respective operator nodes, as they enforce the language definition, while fallthroughs allow some level of deviation with regards to the additionally allowed behavior. We apply our approach to all input logs and count the number of detected cuts and fallthroughs. Additionally, we observe the estimated precision deviations for all fallthroughs. Fig. 14 shows the results.

Across the investigated event logs, only two logs did not require any application of the fallthrough methods. For the remaining input logs, we observed fallthroughs to be frequently applied. We identified multiple reasons for this fallthrough application, which are in line with common problems observed in traditional, single object type, process discovery algorithms. We recognized varying levels of incompleteness, especially with regards to the full connectivity in the directly-follows relation for activities in different partition parts by diverging object types. For increasing partition sizes, the number of relations one would need to observe for a cut quadratically increases, making it much less likely for the log to be complete, even if the original process might fit into the representational bias of the tree. Similar problems have been observed in traditional process discovery when large parts of a process are performed concurrently. Additionally, we could observe multiple cases in which directly-follows relations deviated from the allowed behavior of all available operators. Those can be caused by infrequent or incorrectly recorded behavior or due to the underlying process being outside the utilized representational bias. In those situations, our approach resorted to highly imprecise parallel fallthroughs since no other operator would preserve fitness. For practical applications, it might be beneficial to give up the fitness guarantees for such logs and instead opt for constructs which sacrifice some fitness for a disproportional increase in the estimated precision. This remains future work as it is outside the scope of this paper. In principle one could further relax the conditions of fallthroughs formulated in Section 3.3.

We show one specific example in which this behavior lead to an exceptional decrease in precision. For the procure-to-pay process (05) the recursion, on average, resulted in various perfect cuts and good fallthroughs with precision estimates above 0.75. However, one rather early recursion step lead to the selection of a concurrent fallthrough across a rather larger set of activities, dropping the overall precision of the model. Fig. 15 shows the accumulated directly-follows relation across all object types and the respective activities. Visually, we can clearly observe that the most present behavior in that part of the process is at least partially ordered. However, since no perfectly fitness preserving sequence is available due to divergent types, a concurrent fallthrough with an estimate of only 0.13 is used instead.

5.3. Model quality comparison

Next, we compare our approach against existing techniques for the automated discovery of imperative object-centric process models with compatible conformance checking techniques. In particular, we use the DF^2-Miner for object-centric process trees from [3] and the perspective

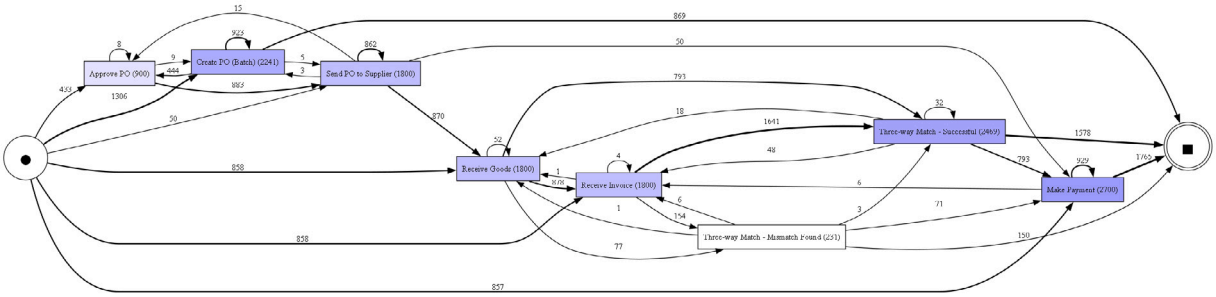


Fig. 15. Accumulated directly-follows relation across all object type for a subset of activities from the procure-to-pay log (05) form the LRMS collection.

Table 10
Conformance checking results for OCIM, the DF^2 Miner and the perspective aggregating OCPN approach [4] on the object-centric input logs from Table 8.

ID	Fitness			Precision		
	OCIM	DF^2 [3]	OCPNs [4]	OCIM	DF^2 [3]	OCPNs [4]
01	1.00	0.88	0.96	0.60	0.74	0.95
02	1.00	1.0	1.00	0.7	0.64	0.80
03	1.00	0.97	1.00	0.78	0.74	0.88
04	1.00	1.0	0.94	0.56	0.36	1.00
05	1.00	0.95	1.00	0.58	0.60	0.81
06	1.00	0.89	0.95	0.64	0.68	0.96
07	1.00	0.98	0.96	0.83	0.86	1.00
08	1.00	0.97	0.99	0.52	0.54	0.94
09	1.00	0.95	0.99	0.64	0.62	0.84
10	1.00	0.24	0.94	0.84	0.96	0.86
11	1.00	1.00	1.00	0.72	0.67	0.79
12	1.00	0.88	0.99	0.45	0.44	0.51
13	1.00	1.00	(-)	0.34	0.31	(-)
14	1.00	0.97	(-)	0.33	0.36	(-)
15	1.00	(-)	(-)	0.34	(-)	(-)
16	1.00	(-)	(-)	0.33	(-)	(-)
17	1.00	(-)	(-)	0.33	(-)	(-)
18	1.00	0.78	(-)	0.26	0.29	(-)
19	1.00	(-)	0.99	0.59	(-)	0.66
20	1.00	0.97	0.99	0.42	0.37	0.38

aggregating approach for traditional process trees that results in object-centric Petri nets (OCPN) from [4] for comparison. We discover models from each input log with each approach and measure fitness and precision. As these approaches all result in object-centric process trees, or collections of traditional, single type, process trees, the conformance checking approach from [17] can be applied. Note that context-based and alignment-based object-centric conformance checking techniques are infeasible for the investigated logs, as they are of exponential runtime complexity and would require excessive filtering, resulting in trivial models with few activities. The resulting conformance values for all approaches and logs can be seen in Table 10. Note that for several input logs, we encountered memory and recursion-depth errors for the DF^2 Miner and the approach from [4], indicated by (-).

Overall, OCIM performed best with regards to fitness, which was expected given the fitness guarantee, resulting in perfect fitness for the models discovered on all logs. Due to these optimal results, the compared approaches from [3,4] performed worse on that quality dimension. The DF^2 Miner resulted in object-centric process trees with an average fitness of 0.90, with one outlier ranging as low as 0.24 in fitness. We inspected this log in particular and found that the way divergence is handled in [3], leads to an edge case in which looping behavior of multiple object types is not detected correctly. For the remaining logs, with only slightly impeded fitness, minor differences between the construction of operator nodes in [3] and their language definition seem to be responsible. The perspective aggregating approach of traditional process trees from [4] performed only slightly worse than our approach when it comes to fitness, with an average value of 0.98 across the logs for which models where successfully discovered.

With regards to precision, the perspective aggregating approach performed the best, with 0.81 being the average precision value. Note that this excludes the six logs for which no model could be found due to apparent implementation errors. OCIM and the DF^2 Miner had an average precision of 0.63 and 0.61 for the same logs respectively. We inspected the resulting models again and noticed that the formalism of object-centric process trees requires some level of agreement between the control flows of different object types to guarantee identifier-soundness. The approach from [4] allows more deviations between object types, enabling a more precise representation of individual object types that might however induce deadlocks in return.

5.4. Discussion

Our evaluation showed that OCIM discovers object-centric process trees in feasible runtime. Our strategy to avoid an exponential runtime complexity in case of the application of fallthrough methods successfully lead to a negligible contribution of the detection to the overall runtime of our approach, except for input logs with extremely large alphabet sizes. The utilized representational bias seems to be applicable to the object-centric event logs used in our evaluation, with some notable exceptions outside the utilized representational bias. In comparison to existing techniques, we found the handling of interaction properties of divergence and deficiency to provide improvements. In particular we found our approach to outperform existing discovery techniques for object-centric process trees. Perspective aggregating approaches showed better precision than our approach, at the cost of potential deadlocks.

Some observed limitations may be attributable to data quality issues, such as incomplete and infrequent behavior. Those factors should be addressed in future work, potentially by applying corresponding measures for the discovery of traditional tree-structured model, such as [11–14]. In that context, the option to balance fitness and precision by giving up the guarantees of our approach could be investigated. Further limitations became apparent due to the inability of object-centric process trees to capture certain control flow structures, such as long term dependencies and partial orders. In the traditional process discovery setting, some extensions exist to account for them, even under the preservation of behaviorally soundness properties [18] and tree-structuredness [8]. It remains an open research question whether this can also be done in the object-centric setting without impeding the provided guarantees of our approach.

The rediscovery guarantee further restricts the representational bias and assumes the absence of any incorrect recordings. However, the presented work hopefully serves as a foundation for further investigations into the algorithmic rediscovery of object-centric processes and an alternative to the mostly compositional approaches for object-centric process discovery so far [4,19]. Extending the formalism of object-centric process trees with identity-based relations between objects could provide an opportunity to extend the provided rediscovery guarantee under the presence of loop structures for all object types. First proposal have been made [16], but it remains to be investigated if the included relations are complete enough to guarantee rediscovery. Further investigations into missing object types, as started in [15], might eliminate the assumption of at least one non-diverging type per activity.

6. Related work

We summarize related work on imperative discovery techniques with formal guarantees for processes with interacting business objects. For a general overview on object-centric process mining techniques, we refer to [20]. A summary of all existing object-centric approaches known to us that provide any of the guarantees discussed in this paper is shown in Table 11. To the best of our knowledge, our approach is the first discovery technique that provides the fitness and rediscovery guarantee for object-centric event logs. By using object-centric process trees, we provide additional behavioral soundness guarantees by construction and remain compatible to existing object-centric conformance checking approaches.

Various implementations of the Inductive Miner framework have been proposed for the discovery of process models in the traditional, single object type, setting [10]. The original version and the extension proposed in [13] both guarantee the input logs to fit the resulting process model. Some attempts have been made to lift the framework into the object-centric setting [4]. The key idea of those approaches is to discover individual models for each object type and then merge them into an object-centric Petri net [4]. However, the fitness guarantee provided by the discovery techniques for each individual object type, does not hold under composition. Hence, such frameworks do not guarantee fitness of the resulting models without making restrictive assumptions on the input logs, especially on the absence of interactions of certain types, such as deficiency. This was also reflected in our evaluation.

The rediscovery of process models has been investigated for traditional event logs in the Inductive Miner framework [10] and other traditional discovery approaches [21]. Various extensions exist that preserve this guarantee [11–14] for traditional logs. However, only limited research has been conducted in the rediscovery of process models from object-centric event logs. The work in [19] shows that for typed Jackson nets, the rediscovery of process models is closed under composition if all interactions between object types are accounted for. However, this insight has not been exploited by object-centric discovery algorithms yet. While we utilize a different representational bias, our work also relies on assuming completeness with regards to the potential interactions between objects, captured by the patterns of divergence, convergence and deficiency. Existing object-centric discovery techniques, such as [4] are limited to individual perspectives when it comes to the rediscovery of processes. That is, it might be that each individual perspective in isolation is rediscovered but the overall process is not, due to a neglect for the interactions between object types.

The utilized representational bias of object-centric process trees guarantees identifier-soundness, i.e. the absence of problematic behavioral constructs such as deadlocks [22]. As a consequence, all models discovered by our approach are guaranteed to be identifier-sound [3]. Similar guarantees are only provided by the modeling formalisms of typed Jackson nets [22]. Other object-centric modeling formalism based on graphs [23], artifacts [24], proclets [25] and constraints [26] either do not provide any soundness notions, or do not integrate them into the discovery of models. Other Petri net-based formalism do have well defined soundness notions, but lack algorithmic support for discovery or conformance [27,28].

Previous work on the discovery of object-centric process trees does not provide any of the two guarantees, as the interaction property of divergence is not accounted for [3], but simply filtered. As such, our approach is the first to provide fitness and rediscovery guarantees for object-centric logs.

Table 11
Formal guarantees and algorithmic support for object-centric approaches, indicated by (✓, available), (∼, only for single object types) and (⊘, unavailable).

Approach	*Soundness*	*Fitness*	*Rediscovery*	*Discovery*	*Conformance*
Knowledge Graphs [23]	⊘	⊘	⊘	✓ [29]	⊘
Artifact-Centric Models [24]	⊘	⊘	⊘	✓ [30,31]	✓ [32]
Object-Centric Petri Nets [4]	∼	∼	∼	✓ [4]	✓ [7,33]
Typed Jackson Nets [22]	✓	⊘	✓	⊘	⊘
Object-Centric Trees [3]	✓	⊘	⊘	✓ [3]	✓ [17,33]
OCIM (Our Approach)	✓	✓	✓	✓	✓

7. Conclusion

In this paper, we introduced a new object-centric discovery technique, based on the representational bias of object-centric process trees. We showed that our technique remains in polynomial runtime complexity and guarantees the discovered models to perfectly fit their respective input logs. Additionally, we proved that for a certain subclass of object-centric process trees, the original processes are rediscovered from correctly recorded and sufficiently complete input logs. We evaluated our approach on a range of publicly available object-centric event logs. We investigated its runtime feasibility, the suitability of the utilized representational bias, and the quality of the discovered process models in comparison to existing techniques. We observed our approach to produce fitting models with competitive precision in feasible runtime. Additionally, we found the utilized representational bias of object-centric process trees to be applicable to the object-centric event logs in our evaluation with some notable exceptions. Those reflect known limitations of tree-structured discovery techniques, in particular the lacking support for long-term dependencies between multiple choices and partial ordered control flow structures. Future work will be focused on mitigating this effect by lifting corresponding extensions to the object-centric setting. Additionally, we intend to investigate if the guarantees provided by our approach can also be guaranteed under the presence of common data quality issues, such as the incompleteness or incorrectness of the object-centric input logs. As discussed in Section 5.4, we also consider extending the formalism of object-centric process trees to provide a foundation for stronger rediscovery guarantees.

CRediT authorship contribution statement

Jan Niklas van Detten: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Pol Schumacher:** Writing – review & editing, Validation, Supervision, Project administration. **Sander J.J. Leemans:** Writing – review & editing, Validation, Supervision, Project administration.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Jan Niklas van Detten reports financial support was provided by Celonis. Jan Niklas van Detten reports a relationship with Celonis that includes: employment. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix 8. Cut detection

In this appendix we provide the in-depth technical description on the cut detection steps and corresponding auxiliary proofs for fitness and rediscovery.

8.1. Concurrent cut detection

For concurrent cuts, we exploit the conditions to decide which activities must be in the same partition part as follows: Condition (22) states that object types related to activities in different partition parts have to be bi-directionally connected by the directly-follows relation of that type. This means that activities that share a related object type without that bi-directional connection have to be in the same partition part of a concurrent cut. Formally, these are all activity pairs (a, b) for which $\exists_{ot \in rel_a \cap rel_b} : a \not\twoheadrightarrow_{\mathbb{L},ot} b \vee b \not\twoheadrightarrow_{\mathbb{L},ot} a$ holds, which would violate condition (22).

The conditions (23) and (24) require that the start and end activities of each individual partition part are also start and end activities for the entire alphabet. Any activity that violates these conditions, i.e. all activities which are in $start_{ot}(\{L\!\downarrow_{\Sigma_i} \mid L \in \mathbb{L}\})$ or $end_{ot}(\{L\!\downarrow_{\Sigma_i} \mid L \in \mathbb{L}\})$ but not in $start_{ot}(\mathbb{L})$ or $end_{ot}(\mathbb{L})$ respectively, are wrongfully separated from an actual start or end activity. For each non-start and non-end activity we check which start and end activities cannot be separated from them without violating the conditions (23) and (24). For a concurrent cut, such pairs of activities need to be in the same part. Formally, these are all activity pairs $(a, b) \in \Sigma \times \Sigma$ for which an object type $ot \in rel_a \cap rel_b$ exists such that (66) or (67) holds.

$$a \in start_{ot}(\mathbb{L}) \wedge b \notin start_{ot}(\mathbb{L}) \wedge b \in start_{ot}(\{L\!\downarrow_{\Sigma\setminus\{a\}} \mid L \in \mathbb{L}\}) \tag{66}$$

$$a \in end_{ot}(\mathbb{L}) \wedge b \notin end_{ot}(\mathbb{L}) \wedge b \in end_{ot}(\{L\!\downarrow_{\Sigma\setminus\{a\}} \mid L \in \mathbb{L}\}) \tag{67}$$

Then, we construct a graph with nodes for $a \in \Sigma$ and undirected edges $(a, b) \in \Sigma \times \Sigma$ between activity pairs that cannot be separated without violating one of the cut conditions. Finally, we check if this graph can be partitioned into multiple connected components. Algorithm 4 shows our detection. Based on this detection strategy, we can guarantee that every detected partition, constitutes a concurrent cut for the set of input logs.

Lemma 16. *Given a set of object-centric event logs $\mathbb{L}$ and the non-empty partition $\Sigma_1, \ldots, \Sigma_n = \text{FINDCUT}_{||}(\mathbb{L})$, the tuple $|| \Sigma_1, \ldots, \Sigma_n$ is a cut for $\mathbb{L}$.*

We proof by contradiction that if the method returns any alphabet partition $\Sigma_1, \ldots \Sigma_n = \text{FINDCUT}_{||}(\mathbb{L})$, it constitutes a concurrent cut for the set of object-centric input logs $\mathbb{L}$. If we assume the partition $\Sigma_1, \ldots \Sigma_n$ to not correspond to a concurrent cut, one of the conditions (22), (23) and (24) needs to be violated. However condition (22) cannot be violated, because all pairs of activities that are not bi-directionally connected by the projected directly-follows relation of a shared related object type, are in the same connected component. Similarly, the separation of activities that would create an invalid start or end activity for individual partition parts, and thereby causing a violation of the conditions (23) or (24), is prohibited by forcing these activities to be in the same partition part. As a result Lemma 16 holds. Reversely, we show that if any concurrent cut exists, our method finds it.

Lemma 17. *If the set of concurrent cuts for a set of object-centric logs $\mathbb{L}$ is not empty, it includes $||, \Sigma_1, \ldots \Sigma_n$ with $\Sigma_1, \ldots, \Sigma_n = \text{FINDCUT}_{||}(\mathbb{L})$.*

Let the set of concurrent cuts for the set of object-centric event logs $\mathbb{L}$ be not empty, and let $||, \Sigma_1, \ldots, \Sigma_n$ be one of them with a maximal number of partition parts out of all cuts. Due to the fulfilled condition (22) the connected component analysis in Algorithm 4 guarantees that all activities in a detected connected component, are in the same partition part of the maximal cut. As such, the partition detected by the connected component analysis $\Sigma'_1 \ldots \Sigma'_m$ guarantees $m \geq n$ and $\forall_{1 \leq i \leq m} : \exists_{1 \leq j \leq n} : \Sigma'_i \subseteq \Sigma_j$. If $m > n$, some of the projected start or end activities of the logs in $\mathbb{L}$ have to be wrongfully separated from other activities. However, this is not possible, since these activity pairs are guaranteed to be in the same connected component, prohibiting wrongful separations. As such, we can guarantee $m = n$, meaning the detected partition has to correspond to a permutation of the maximal concurrent cut and is hence a maximal concurrent cut as well.

Due to Lemmas 16 and 17, we can conclude that if and only if there is no cut for $\mathbb{L}$ with the concurrent operator, nothing is returned by $\text{FINDCUT}_{||}(\mathbb{L})$. Additionally, if and only if there is at least one cut for $\mathbb{L}$, the method returns a cut with the maximal number of partition parts. We assume a deterministic ordering function to be given to select the order of the partition parts.

8.2. Exclusive choice cut detection

The exclusive choice operator shares condition (23) and (24) with the concurrent operator. We therefore reuse the exploit for these two conditions for the exclusive choice operator. That is, activity pairs have to be in the same partition part if separating them leads to a violation.

Additionally, condition (25) states that all activities in different partition parts that share a related non-divergent object type, are not allowed to be connected by the directly-follows relation. If there is any such relation, the two activities have to be in the same partition part. Formally, these are all activity pairs $(a, b) \in \Sigma \times \Sigma$ for which an object type $ot \in (rel_a \cap rel_b) \setminus (div_a \cap div_b)$ exists such that $a \twoheadrightarrow_{\mathbb{L},ot} b \vee b \twoheadrightarrow_{\mathbb{L},ot} a$ holds. Based on the activity pairs identified above we again perform a connected component analysis to construct an alphabet partition $\Sigma'_1 \ldots \Sigma'_m$ that adheres to (23), (24) and (25).

However, exclusive choice cuts additionally require, with condition (26), that object types that are fully divergent on two partition parts, exhibit bi-directional directly-follows relation between activities across these two partition parts. If this is not the case for two parts of the identified partition $\Sigma'_1 \ldots \Sigma'_m$, all activities in those parts need to be in the same partition part of a cut. Formally, these are the activity pairs

Algorithm 4: Concurrent Cut Detection

```
Function CHECK_||(𝕃, a, b)
    for ot ∈ rel_a ∩ rel_b do
        if a ↛_{𝕃,ot} b ∨ b ↛_{𝕃,ot} a then
            return True
        if a ∈ start_ot(𝕃) ∧ b ∉ start_ot(𝕃) ∧ b ∈
          start_ot({L↓_{Σ\{a}} | L ∈ 𝕃}) then
            return True
        if a ∈ end_ot(𝕃) ∧ b ∉ end_ot(𝕃) ∧ b ∈ end_ot({L↓_{Σ\{a}} | L ∈ 𝕃})
          then
            return True
    return False

Function FINDCUT_||(𝕃)
    edges ← {(a, b) | a, b ∈ Σ ∧ CHECK_||(𝕃, a, b)}
    Σ_1, ... Σ_n ← CONNECTEDCOMPONENT(Σ, edges)
    if n > 1 then
        return Σ_1, ... Σ_n
```

$(a,b) \in \Sigma \times \Sigma$ with $a \in \Sigma'_i \wedge b \in \Sigma'_j \wedge i \neq j$ for which an object type $ot \in D(\Sigma'_i, \Sigma'_j)$ exists, such that $\exists c \in \Sigma'_i, d \in \Sigma'_j : c \not\twoheadrightarrow_{\mathbb{L},ot} d \vee d \not\twoheadrightarrow_{\mathbb{L},ot} c$. The last condition (27) requires that all pairs of partition parts share at least one non-diverging object type with each other. If this is not the case, all activities in these two parts should be in the same partition part for a cut. Formally, these are the activity pairs $a, b \in \Sigma$ with $a \in \Sigma'_i \wedge b \in \Sigma'_j \wedge i \neq j$ for which $\forall a, b \in \Sigma'_i \times \Sigma'_j : rel_a \cap rel_b \setminus D(\Sigma'_i, \Sigma'_j) = \emptyset$ holds. With these newly added information on activity pairs that have to be in the same partition part for a cut, we can determine the final connected components of the activity graph. If multiple connected components exist, the corresponding alphabet partition is returned. Algorithm 5 shows this detection strategy. Note that the two sequential detections of the connected components is required, since we first need initial partition parts to check for the directly-follows relation of diverging object types. We again prove that the detected partition is a cut.

Lemma 18. *Given a set of object-centric event logs $\mathbb{L}$ and the non-empty partition $\Sigma_1, \ldots, \Sigma_n = \text{FINDCUT}_\times(\mathbb{L})$, the tuple $\times \Sigma_1, \ldots, \Sigma_n$ is a cut for $\mathbb{L}$.*

The proof strategy is the same as for the concurrent operator. Building a graph with edges for the separation of activities that would violate a condition ensures that the partition that corresponds to the connected components in that graph adheres to these conditions, in this case (23), (24), (25), (26) and (27). Hence, Lemma 18 holds. Reversely, we can again show that if any maximal exclusive choice cut $\times, \Sigma_1, \ldots \Sigma_n$ exists, our method finds it.

Lemma 19. *If the set of exclusive choice cuts for a set of object-centric logs $\mathbb{L}$ is not empty, it includes $\times, \Sigma_1, \ldots \Sigma_n$ with $\Sigma_1, \ldots, \Sigma_n = \text{FINDCUT}_\times(\mathbb{L})$.*

Similar to the concurrent cut detection, we utilize the fact that the first connected component detection will result in a partition $\Sigma'_1 \ldots \Sigma'_m$ that guarantees $m \geq n$ and $\forall_{1 \leq i \leq m} : \exists_{1 \leq j \leq n} : \Sigma'_i \subseteq \Sigma_j$. If $m = n$, we again have a permutation of the maximal cut. If $m > n$, one of the conditions (26) and (27) have to be violated. However, all of these violations are removed by the second application of the connection component detection, guaranteeing $m = n$ afterwards. A permutation of the partition for the maximal cut is therefore guaranteed to be found by our method, i.e. Lemma 19 holds. Again, we assume an ordering function to be given to ensure a deterministic order.

Algorithm 5: Exclusive Choice Cut Detection

Function $\text{CHECK}_{\times^1}(\mathbb{L}, a, b)$
- **for** $ot \in (rel_a \cap rel_b)$ **do**
  - **if** $(a \twoheadrightarrow_{\mathbb{L},ot} b \vee b \twoheadrightarrow_{\mathbb{L},ot} a) \wedge ot \notin (div_a \cap div_b)$ **then**
    - **return** True
  - **if** $a \in start_{ot}(\mathbb{L}) \wedge b \notin start_{ot}(\mathbb{L}) \wedge b \in start_{ot}(\{L|_{\Sigma \setminus \{a\}} \mid L \in \mathbb{L}\})$ **then**
    - **return** True
  - **if** $a \in end_{ot}(\mathbb{L}) \wedge b \notin end_{ot}(\mathbb{L}) \wedge b \in end_{ot}(\{L|_{\Sigma \setminus \{a\}} \mid L \in \mathbb{L}\})$ **then**
    - **return** True
- **return** False

Function $\text{CHECK}_{\times^2}(\mathbb{L}, \Sigma'_i, \Sigma'_j)$
- **for** $(a,b) \in (\Sigma'_i \times \Sigma'_j)$ **do**
  - **for** $ot \in rel_a \cap rel_b \cap D(\Sigma_i, \Sigma_j)$ **do**
    - **if** $a \not\twoheadrightarrow_{\mathbb{L},ot} b \vee b \not\twoheadrightarrow_{\mathbb{L},ot} a$ **then**
      - **return** True
- **if** $\forall a, b \in \Sigma'_i \times \Sigma'_j : rel_a \cap rel_b \setminus D(\Sigma'_i, \Sigma'_j) = \emptyset$ **then**
  - **return** True
- **return** False

Function $\text{FINDCUT}_\times(\mathbb{L})$
- $edges \leftarrow \{(a,b) \mid a, b \in \Sigma \wedge \text{CHECK}_{\times^1}(\mathbb{L}, a, b)\}$
- $\Sigma'_1, \ldots \Sigma'_m \leftarrow \text{CONNECTEDCOMPONENT}(\Sigma, edges)$
- $edges \leftarrow edges \cup \{(a,b) \mid \exists_{1 \leq i \neq j \leq n} : (a,b) \in \Sigma'_i \times \Sigma'_j \wedge \text{CHECK}_{\times^2}(\mathbb{L}, \Sigma'_i, \Sigma'_j)\}$
- $\Sigma_1, \ldots \Sigma_n \leftarrow \text{CONNECTEDCOMPONENT}(\Sigma, edges)$
- **if** $n > 1$ **then**
  - **return** $\Sigma_1 \ldots \Sigma_n$

8.3. Sequence cut detection

For the sequence operator, we detect the connected components three times. Each time, we determine additional pairs of activities that have to be in same part of the partition to constitute a cut. Condition (19) requires that if activities share a related non-diverging object type, the transitive closure of the directly-follows relation should only connect them in one direction. If this connection does not exist at all, or is bi-directionally, the two activities have to be in the same partition part. Formally, these are all activity pairs for which an object type $ot \in (rel_a \cap rel_b) \setminus (div_a \cap div_b)$ exists, such that $(a \twoheadrightarrow^+_{\mathbb{L},ot} b \wedge b \twoheadrightarrow^+_{\mathbb{L},ot} a) \vee (a \not\twoheadrightarrow^+_{\mathbb{L},ot} b \wedge ab \not\twoheadrightarrow^+_{\mathbb{L},ot} a)$ holds. Detecting the connected components on the corresponding activity graph with edges for these activity pairs, results in a first alphabet partition $\Sigma''_1, \ldots \Sigma''_l$.

Next, we determine the correct order between these partition parts. For this purpose we define the partition follows relations $\Sigma''_i \twoheadrightarrow_{\mathbb{L}} \Sigma''_j$ which holds if an activity pair $a \in \Sigma_i, b \in \Sigma_j$ and an object type $ot (\in rel_a \cap rel_b) \setminus D(\Sigma''_j, \Sigma''_i)$ exists such that $a \twoheadrightarrow_{\mathbb{L},ot} b$. We write $\Sigma'_i \twoheadrightarrow^+_{\mathbb{L}} \Sigma'_j$ for the transitive closure of that relation. Condition (21) requires partition parts that follow each other to share a non-diverging object type. Therefore, if two partition parts do not have any distinct transitive order, all activities in them should be in the same partition part for a cut. Formally, these are the activity pairs $(a,b) \in \Sigma''_i \times \Sigma''_j$ for which $\Sigma''_i \not\twoheadrightarrow_{\mathbb{L}} \Sigma''_j \wedge \Sigma''_j \not\twoheadrightarrow_{\mathbb{L}} \Sigma''_i$ or $\Sigma''_i \twoheadrightarrow_{\mathbb{L}} \Sigma''_j \wedge \Sigma''_j \twoheadrightarrow_{\mathbb{L}} \Sigma''_i$ holds. Detecting the connected components on the activity graph with added

Algorithm 6: Sequence Cut Detection

Function $\text{CHECK}_{\rightarrow^1}(\mathbb{L}, a, b)$
- **for** $ot \in (rel_a \cap rel_b) \setminus (div_a \cap div_b)$ **do**
  - **if** $a \twoheadrightarrow^+_{\mathbb{L},ot} b \wedge b \twoheadrightarrow^+_{\mathbb{L},ot} a$ **then**
    - **return** True
  - **if** $a \not\twoheadrightarrow^+_{\mathbb{L},ot} b \wedge b \not\twoheadrightarrow^+_{\mathbb{L},ot} a$ **then**
    - **return** True
- **return** False

Function $\text{CHECK}_{\rightarrow^2}(\mathbb{L}, \Sigma''_1, \ldots, \Sigma''_l, i, j)$
- **if** $(\Sigma''_i \not\twoheadrightarrow^+_{\mathbb{L}} \Sigma''_j \wedge \Sigma''_j \not\twoheadrightarrow^+_{\mathbb{L}} \Sigma''_i) \vee (\Sigma''_i \twoheadrightarrow^+_{\mathbb{L}} \Sigma''_j \wedge \Sigma''_j \twoheadrightarrow^+_{\mathbb{L}} \Sigma''_i)$ **then**
  - **return** True
- **return** False

Function $\text{CHECK}_{\rightarrow^3}(\mathbb{L}, \Sigma'_1, \ldots, \Sigma'_m, i, j)$
- **for** $(a,b) \in (\Sigma'_i \times \Sigma'_j)$ **do**
  - **for** $ot \in rel_a \cap rel_b \cap D(\Sigma'_i, \ldots, \Sigma'_j)$ **do**
    - **if** $a \not\twoheadrightarrow_{\mathbb{L},ot} b \vee b \not\twoheadrightarrow_{\mathbb{L},ot} a$ **then**
      - **return** True
- **return** False

Function $\text{FINDCUT}_\rightarrow(\mathbb{L})$
- $edges \leftarrow \{(a,b) \mid a, b \in \Sigma \wedge \text{CHECK}_{\rightarrow^1}(\mathbb{L}, a, b)\}$
- $\Sigma''_1, \ldots \Sigma''_l \leftarrow \text{CONNECTEDCOMPONENT}(\Sigma, edges)$
- $edges \leftarrow edges \cup \{(a,b) \mid (a,b) \in \Sigma''_i \times \Sigma''_j \wedge \text{CHECK}_{\rightarrow^2}(\mathbb{L}, \Sigma''_1 \ldots \Sigma''_l, i, j)\}$
- $\Sigma'_1, \ldots \Sigma'_m \leftarrow \text{ORDERED}(\text{CONNECTEDCOMPONENT}(\Sigma, edges))$
- $edges \leftarrow edges \cup \{(a,b) \mid (a,b) \in \Sigma'_i \times \Sigma'_j \wedge \text{CHECK}_{||^3}(\mathbb{L}, \Sigma'_1 \ldots \Sigma'_m, i, j)\}$
- $\Sigma_1, \ldots \Sigma_n \leftarrow \text{ORDERED}(\text{CONNECTEDCOMPONENT}(\Sigma, edges))$
- **if** $n > 1$ **then**
  - **return** $\Sigma_1, \ldots \Sigma_n$

edges for these activity pairs and ordering them according to $\twoheadrightarrow_L^+$ results in $\Sigma'_1 \dots \Sigma'_m$.

Condition (21) requires bi-directional connections between activities in different partition parts, by object types that diverge in these partition parts and those in between them. If this connectivity is missing, the corresponding activities should be in the same partition part for the cut. Formally, these are the activity pairs $a \in \Sigma'_i, b \in \Sigma'_j$ for which an object type $ot \in rel_a \cap rel_b \cap D(\Sigma'_i \dots \Sigma'_j)$ exists such that $a \not\twoheadrightarrow_{\mathbb{L},ot} b \vee b \not\twoheadrightarrow^+_{\mathbb{L},ot} a$. If the activity graph can be split into multiple connected components after adding edges for these pairs, the resulting partition is returned, as shown in Algorithm 6. Note that the three staged approach is necessary, because we first need to have a minimal partition to then determine a valid order between the parts before we can finally account for the directly-follows relation of divergent object types. We can again guarantee the resulting partition to constitute a cut.

Lemma 20. *Given a set of object-centric event logs $\mathbb{L}$ and the non-empty partition $\Sigma_1, \dots, \Sigma_n = \text{FINDCUT}_{\rightarrow}(\mathbb{L})$, the tuple $\rightarrow \Sigma_1, \dots, \Sigma_n$ is a cut for $\mathbb{L}$.*

We again apply a similar strategy as for the previous operators to prove that all detected partitions correspond to cuts. For each application of the connected components, the inclusion of edges that would violate the conditions (20), (19) and (21) ensures that they are fulfilled, guaranteeing Lemma 20. As for previous operators, we show that if any sequence cut exists, it is found.

Lemma 21. *If the set of sequence cuts for a set of object-centric logs $\mathbb{L}$ is not empty, it includes $\rightarrow, \Sigma_1, \dots \Sigma_n$ with $\Sigma_1, \dots, \Sigma_n = \text{FINDCUT}_{\rightarrow}(\mathbb{L})$.*

We again have the first two detection of connected components result in a partition $\Sigma'_1 \dots \Sigma'_m$ that guarantees $m \geq n$ and $\forall_{1 \leq i \leq m} : \exists_{1 \leq j \leq n} : \Sigma'_i \subseteq \Sigma_j$. If $m = n$, we again have the maximal cut. If $m > n$, the condition (21) has to be violated, which will be addressed by the third application of the connected component detection, guaranteeing $m = n$, and hence Lemma 21 holds.

8.4. Loop cut detection

For the loop operator, we first explicitly check condition (30), since it is independent of the chosen partition. If it does not hold, no loop cut can exist. Next, we determine for each fully divergent object type, which activities are not bi-directionally connected with each other by the directly-follows relation. These pairs of activities have to be in the same partition part due to condition (29). Formally, these are $(a, b) \in \Sigma \times \Sigma$ for which an object type $ot \in D(\Sigma)$ exists, such that $a \not\twoheadrightarrow_{\mathbb{L},ot} b \vee b \not\twoheadrightarrow_{\mathbb{L},ot} a$. Additionally, condition (31) and (32) require start and end activities of all non-diverging object types to be in the same partition part. Formally, these are $(a, b) \in \Sigma \times \Sigma$ for which an object type $ot \in rel_a \cap rel_b \setminus D(\Sigma)$ exists such that $a, b \in start_{ot}(\mathbb{L}) \cup end_{ot}(\mathbb{L})$.

Furthermore, condition (33) and (34) require activities with a directly-follows relation between them by a non-diverging object type to be in the same partition part, if the relation does not start from an end activity or ends on a start activity of that type. Formally, these are all $(a, b) \in \Sigma \times \Sigma$ for which an object type $ot \in rel_a \cap rel_b \setminus D(\Sigma)$ exists with $a \twoheadrightarrow_{\mathbb{L},ot} b$ and $a \notin end_{ot}(\mathbb{L}) \wedge b \notin start_{ot}(\mathbb{L})$. We perform a connected component analysis to find an alphabet partition $\Sigma_1, \dots \Sigma_n$.

Lastly, according to condition (28), there must be at least one non-diverging object type that is present in the body part and the redo part of the loop. Therefore, we iterate over all non-diverging object types and check if treating Σ_i with $\Sigma_i \cap (start_{ot}(\mathbb{L}) \cup end_{ot}(\mathbb{L}) \neq \emptyset)$ as the body part of the loop and $\cup_{j \neq i} \Sigma_j$ as the redo part of the loop, results in a cut. If so, we return the corresponding alphabet partition. Algorithm 7 shows this loop cut detection strategy.

Algorithm 7: Loop Cut Detection

Function $\text{CHECK}_{\circlearrowleft}(\mathbb{L}, a, b)$
- **for** $ot \in rel_a \cap rel_b$ **do**
  - **if** $a \not\twoheadrightarrow_{\mathbb{L},ot} b \vee b \not\twoheadrightarrow_{\mathbb{L},ot} a \wedge ot \in D(\Sigma)$ **then**
    - **return** True
  - **if** $a, b \in start_{ot}(\mathbb{L}) \cup end_{ot}(\mathbb{L}) \wedge ot \notin D(\Sigma)$ **then**
    - **return** True
  - **if** $a \twoheadrightarrow_{\mathbb{L},ot} b \wedge a \notin end_{ot}(\mathbb{L}) \wedge b \notin start_{ot}(\mathbb{L}) \wedge ot \notin D(\Sigma)$ **then**
    - **return** True
- **return** False

Function $\text{FINDCUT}_{\circlearrowleft}(\mathbb{L})$
- **if** $\forall a, b \in \Sigma : \forall ot \in rel_a \cap rel_b : a \twoheadrightarrow^+_{\mathbb{L},ot} b \wedge b \twoheadrightarrow^+_{\mathbb{L},ot} a$ **then**
  - $edges \leftarrow \{(a, b) \mid a, b \in \Sigma \wedge \text{CHECK}_{\circlearrowleft}(\mathbb{L}, a, b)\}$
  - $\Sigma_1, \dots \Sigma_n \leftarrow \text{CONNECTEDCOMPONENT}(\Sigma, edges)$
  - **for** $ot \in \Omega \setminus D(\Sigma)$ **do**
    - **for** $i \in 1 \dots n$ **do**
      - **if** $\Sigma_i \cap (start_{ot}(\mathbb{L} \cup end_{ot}(\mathbb{L})) \neq \emptyset$ **then**
        - $\Sigma_{body} = \Sigma_i$
    - **for** $j \in 1 \dots n$ **do**
      - **if** $\exists a \in \Sigma_j : ot \in rel_a \wedge \Sigma_j \cap \Sigma_{body} = \emptyset$ **then**
        - $\Sigma_{redo} = \Sigma \setminus \Sigma_{body}$
        - **return** $\Sigma_{body}, \Sigma_{redo}$

Data availability

All used data sets and the code for experiments are publicly available.

References

[1] W.M.P. van der Aalst, J. Carmona (Eds.), Process mining handbook, in: Lecture Notes in Business Information Processing, vol. 448, Springer, 2022, http://dx.doi.org/10.1007/978-3-031-08848-3.

[2] W.M.P. van der Aalst, Object-centric process mining: An introduction, in: A. Cerone (Ed.), Formal Methods for an Informal World - ICTAC 2021 Summer School, Virtual Event, Astana, Kazakhstan, September 1-7, 2021, Tutorial Lectures, in: Lecture Notes in Computer Science, vol. 13490, Springer, 2021, pp. 73–105, http://dx.doi.org/10.1007/978-3-031-43678-9_3.

[3] J.N. van Detten, P. Schumacher, S.J.J. Leemans, Discovering compact, live and identifier-sound object-centric process models, in: 6th International Conference on Process Mining, ICPM 2024, Kgs. Lyngby, Denmark, October 14-18, 2024, IEEE, 2024, pp. 113–120, http://dx.doi.org/10.1109/ICPM63005.2024.10680659.

[4] W.M.P. van der Aalst, A. Berti, Discovering object-centric Petri nets, Fund. Inform. 175 (1–4) (2020) 1–40, http://dx.doi.org/10.3233/FI-2020-1946.

[5] G. Park, J.N. Adams, W.M.P. van der Aalst, Conformance checking and performance analysis using object-centric directly-follows graphs, in: A. Marrella, M. Resinas, M. Jans, M. Rosemann (Eds.), Business Process Management Forum - BPM 2024 Forum, Krakow, Poland, September 1-6, 2024, Proceedings, in: Lecture Notes in Business Information Processing, vol. 526, Springer, 2024, pp. 179–196, http://dx.doi.org/10.1007/978-3-031-70418-5_11.

[6] J. Carmona, B.F. van Dongen, M. Weidlich, Conformance checking: Foundations, milestones and challenges, in: W.M.P. van der Aalst, J. Carmona (Eds.), Process Mining Handbook, in: Lecture Notes in Business Information Processing, vol. 448, Springer, 2022, pp. 155–190, http://dx.doi.org/10.1007/978-3-031-08848-3_5.

[7] L. Liss, J.N. Adams, W.M.P. van der Aalst, Object-centric alignments, in: J.P.A. Almeida, J. Borbinha, G. Guizzardi, S. Link, J. Zdravkovic (Eds.), Conceptual Modeling - 42nd International Conference, ER 2023, Lisbon, Portugal, November 6-9, 2023, Proceedings, in: Lecture Notes in Computer Science, vol. 14320, Springer, 2023, pp. 201–219, http://dx.doi.org/10.1007/978-3-031-47262-6_11.

[8] H. Kourani, S.J. van Zelst, POWL: partially ordered workflow language, in: Business Process Management - 21st International Conference, BPM 2023, Utrecht, the Netherlands, September 11-15, 2023, Proceedings, in: Lecture Notes in Computer Science, vol. 14159, Springer, 2023, pp. 92–108, http://dx.doi.org/10.1007/978-3-031-41620-0_6.

[9] G. Li, E.G.L. de Murillas, R.M. de Carvalho, W.M.P. van der Aalst, Extracting object-centric event logs to support process mining on databases, in: J. Mendling, H. Mouratidis (Eds.), Information Systems in the Big Data Era - CAiSE Forum 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings, in: Lecture Notes in Business Information Processing, vol. 317, Springer, 2018, pp. 182–199, http://dx.doi.org/10.1007/978-3-319-92901-9_16.

[10] S.J.J. Leemans, D. Fahland, W.M.P. van der Aalst, Discovering block-structured process models from event logs - A constructive approach, in: J.M. Colom, J. Desel (Eds.), Application and Theory of Petri Nets and Concurrency - 34th International Conference, PETRI NETS 2013, Milan, Italy, June 24-28, 2013. Proceedings, in: Lecture Notes in Computer Science, vol. 7927, Springer, 2013, pp. 311–329, http://dx.doi.org/10.1007/978-3-642-38697-8_17.

[11] J.N. van Detten, P. Schumacher, S.J.J. Leemans, An approximate inductive miner, in: 5th International Conference on Process Mining, ICPM 2023, Rome, Italy, October 23-27, 2023, IEEE, 2023, pp. 129–136, http://dx.doi.org/10.1109/ICPM60904.2023.10271971.

[12] S.J.J. Leemans, D. Fahland, W.M.P. van der Aalst, Discovering block-structured process models from event logs containing infrequent behaviour, in: N. Lohmann, M. Song, P. Wohed (Eds.), Business Process Management Workshops - BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers, in: Lecture Notes in Business Information Processing, vol. 171, Springer, 2013, pp. 66–78, http://dx.doi.org/10.1007/978-3-319-06257-0_6.

[13] S.J.J. Leemans, D. Fahland, W.M.P. van der Aalst, Discovering block-structured process models from incomplete event logs, in: G. Ciardo, E. Kindler (Eds.), Application and Theory of Petri Nets and Concurrency - 35th International Conference, PETRI NETS 2014, Tunis, Tunisia, June 23-27, 2014. Proceedings, in: Lecture Notes in Computer Science, vol. 8489, Springer, 2014, pp. 91–110, http://dx.doi.org/10.1007/978-3-319-07734-5_6.

[14] D. Brons, R. Scheepens, D. Fahland, Striking a new balance in accuracy and simplicity with the probabilistic inductive miner, in: C.D. Ciccio, C.D. Francescomarino, P. Soffer (Eds.), 3rd International Conference on Process Mining, ICPM 2021, Eindhoven, the Netherlands, October 31 - Nov. 4, 2021, IEEE, 2021, pp. 32–39, http://dx.doi.org/10.1109/ICPM53251.2021.9576864.

[15] J.N. van Detten, P. Schumacher, S.J.J. Leemans, Object synchronizations and specializations with silent objects in object-centric Petri nets, in: A. Marrella, M. Resinas, M. Jans, M. Rosemann (Eds.), Business Process Management - 22nd International Conference, BPM 2024, Krakow, Poland, September 1-6, 2024, Proceedings, in: Lecture Notes in Computer Science, vol. 14940, Springer, 2024, pp. 57–74, http://dx.doi.org/10.1007/978-3-031-70396-6_4.

[16] J.N. Van Detten, P. Schumacher, S.J.J. Leemans, Modeling and discovering dynamic identity relations in object-centric process mining, in: 2025 7th International Conference on Process Mining (ICPM), 2025, pp. 1–8, http://dx.doi.org/10.1109/ICPM66919.2025.11220672.

[17] J.N. van Detten, C. Rennert, A. Küsters, W.M.P.v. der Aalst, S.J.J. Leemans, Footprint-based conformance checking for object-centric process trees, in: F.M. Maggi, H. Leopold, F. Santoro, A. Gal (Eds.), Business Process Management, Springer Nature Switzerland, Cham, 2027, pp. 202–219.

[18] L.L. Mannel, W.M.P. van der Aalst, Discovering process models with long-term dependencies while providing guarantees and filtering infrequent behavior patterns, Fund. Inform. 190 (2–4) (2024) 109–158, http://dx.doi.org/10.3233/FI-242168.

[19] D. Barenholz, M. Montali, A. Polyvyanyy, H.A. Reijers, A. Rivkin, J.M.E.M. van der Werf, There and back again - on the reconstructability and rediscoverability of typed jackson nets, in: L. Gomes, R. Lorenz (Eds.), Application and Theory of Petri Nets and Concurrency - 44th International Conference, PETRI NETS 2023, Lisbon, Portugal, June 25-30, 2023, Proceedings, in: Lecture Notes in Computer Science, vol. 13929, Springer, 2023, pp. 37–58, http://dx.doi.org/10.1007/978-3-031-33620-1_3.

[20] A. Berti, M. Montali, W.M.P. van der Aalst, Advancements and challenges in object-centric process mining: A systematic literature review, 2023, http://dx.doi.org/10.48550/ARXIV.2311.08795, CoRR abs/2311.08795, arXiv:2311.08795.

[21] W.M.P. van der Aalst, T. Weijters, L. Maruster, Workflow mining: Discovering process models from event logs, IEEE Trans. Knowl. Data Eng. 16 (9) (2004) 1128–1142, http://dx.doi.org/10.1109/TKDE.2004.47.

[22] J.M.E.M. van der Werf, A. Rivkin, A. Polyvyanyy, M. Montali, Data and process resonance - identifier soundness for models of information systems, in: L. Bernardinello, L. Petrucci (Eds.), Application and Theory of Petri Nets and Concurrency - 43rd International Conference, PETRI NETS 2022, Bergen, Norway, June 19-24, 2022, Proceedings, in: Lecture Notes in Computer Science, vol. 13288, Springer, 2022, pp. 369–392, http://dx.doi.org/10.1007/978-3-031-06653-5_19.

[23] D. Fahland, Process mining over multiple behavioral dimensions with event knowledge graphs, in: W.M.P. van der Aalst, J. Carmona (Eds.), Process Mining Handbook, in: Lecture Notes in Business Information Processing, vol. 448, Springer, 2022, pp. 274–319, http://dx.doi.org/10.1007/978-3-031-08848-3_9.

[24] D. Fahland, Artifact-centric process mining, in: S. Sakr, A.Y. Zomaya (Eds.), Encyclopedia of Big Data Technologies, Springer, 2019, http://dx.doi.org/10.1007/978-3-319-63962-8_93-1.

[25] W.M.P. van der Aalst, P. Barthelmess, C.A. Ellis, J. Wainer, Proclets: A framework for lightweight interacting workflow processes, Int. J. Coop. Inf. Syst. 10 (4) (2001) 443–481, http://dx.doi.org/10.1142/S0218843001000412.

[26] G. Li, R.M. de Carvalho, W.M.P. van der Aalst, Automatic discovery of object-centric behavioral constraint models, in: W. Abramowicz (Ed.), Business Information Systems - 20th International Conference, BIS 2017, Poznan, Poland, June 28-30, 2017, Proceedings, in: Lecture Notes in Business Information Processing, vol. 288, Springer, 2017, pp. 43–58, http://dx.doi.org/10.1007/978-3-319-59336-4_4.

[27] F. Rosa-Velardo, D. de Frutos-Escrig, Decidability and complexity of Petri nets with unordered data, Theoret. Comput. Sci. 412 (34) (2011) 4439–4451, http://dx.doi.org/10.1016/J.TCS.2011.05.007.

[28] S. Ghilardi, A. Gianola, M. Montali, A. Rivkin, Petri nets with parameterised data - modelling and verification, in: D. Fahland, C. Ghidini, J. Becker, M. Dumas (Eds.), Business Process Management - 18th International Conference, BPM 2020, Seville, Spain, September 13-18, 2020, Proceedings, in: Lecture Notes in Computer Science, vol. 12168, Springer, 2020, pp. 55–74, http://dx.doi.org/10.1007/978-3-030-58666-9_4.

[29] C.O. Back, J.G. Simonsen, Discovering order-inducing features in event knowledge graphs, in: M. Comuzzi, D. Grigori, M. Sellami, Z. Zhou (Eds.), Cooperative Information Systems - 30th International Conference, CoopIS 2024, Porto, Portugal, November 19-21, 2024, Proceedings, in: Lecture Notes in Computer Science, vol. 15506, Springer, 2024, pp. 382–390, http://dx.doi.org/10.1007/978-3-031-81375-7_25.

[30] M.L. van Eck, N. Sidorova, W.M.P. van der Aalst, Multi-instance mining: Discovering synchronisation in artifact-centric processes, in: F. Daniel, Q.Z. Sheng, H. Motahari (Eds.), Business Process Management Workshops - BPM 2018 International Workshops, Sydney, NSW, Australia, September 9-14, 2018, Revised Papers, in: Lecture Notes in Business Information Processing, vol. 342, Springer, 2018, pp. 18–30, http://dx.doi.org/10.1007/978-3-030-11641-5_2.

[31] V. Popova, D. Fahland, M. Dumas, Artifact lifecycle discovery, Int. J. Coop. Inf. Syst. 24 (1) (2015) 1550001:1–1550001:44, http://dx.doi.org/10.1142/S021884301550001X.

[32] D. Fahland, M. de Leoni, B.F. van Dongen, W.M.P. van der Aalst, Behavioral conformance of artifact-centric process models, in: W. Abramowicz (Ed.), Business Information Systems - 14th International Conference, BIS 2011, Poznan, Poland, June 15-17, 2011. Proceedings, in: Lecture Notes in Business Information Processing, vol. 87, Springer, 2011, pp. 37–49, http://dx.doi.org/10.1007/978-3-642-21863-7_4.

[33] J.N. Adams, W.M.P. van der Aalst, Precision and fitness in object-centric process mining, in: C.D. Ciccio, C.D. Francescomarino, P. Soffer (Eds.), 3rd International Conference on Process Mining, ICPM 2021, Eindhoven, the Netherlands, October 31 - Nov. 4, 2021, IEEE, 2021, pp. 128–135, http://dx.doi.org/10.1109/ICPM53251.2021.9576886.