Uncovering Object-Centric Data in Classical Event Logs for the Automated Transformation from XES to OCEL

Adrian Rebmann[1(✉)], Jana-Rebecca Rehse[2], and Han van der Aa[1]

[1] Data and Web Science Group, University of Mannheim, Mannheim, Germany
{rebmann,han}@informatik.uni-mannheim.de
[2] Management Analytics Center, University of Mannheim, Mannheim, Germany
rehse@uni-mannheim.de

Abstract. Object-centric event logs have recently been introduced as a means to capture event data of processes that handle multiple concurrent object types, with potentially complex interrelations. Such logs allow process mining techniques to handle multi-object processes in an appropriate manner. However, event data is often not yet available in this new format, but is rather captured in the form of classical, "flat" event logs. This flat representation obscures the true interrelations that exist between different objects and associated events, causing issues such as the well-known convergence and divergence of event data. This situation calls for support to transform classical event logs into object-centric counterparts. Such a transformation is far from straightforward, though, given that the information required for object-centric logs, such as explicitly indicated object types, identifiers, and properties, is not readily available in flat logs. In this paper, we propose an approach that automatically uncovers object-related information in flat event data and uses this information to transform the flat data into an object-centric event log according to the OCEL format. We achieve this by combining the semantic analysis of textual attributes with data profiling and control-flow-based relation extraction techniques. We demonstrate our approach's efficacy through evaluation experiments and highlight its usefulness by applying it to real-life event logs in order to mitigate the quality issues caused by their flat representation.

Keywords: Process mining · Object-centric event logs · Semantic analysis

1 Introduction

Process mining focuses on the analysis of event data recorded by information systems in order to gain insights into the true behavior of organizational processes [1]. This behavior is captured in event logs, i.e., sequences of events that denote the execution of activities in the process. Traditional process mining techniques assume that each event in the log refers to exactly one case, represented by a single unambiguous case notion. To define this case notion, researchers commonly choose the main object type that is handled by the process, e.g., a request or an application.

However, defining a single unambiguous case notion becomes problematic if the process handles multiple concurrent object types, with potentially complex interrelations. For example, in an order handling process, multiple items can be part of one

© Springer Nature Switzerland AG 2022
C. Di Ciccio et al. (Eds.): BPM 2022, LNCS 13420, pp. 379–396, 2022.
https://doi.org/10.1007/978-3-031-16103-2_25

order and multiple orders can be shipped in one package. For such a process, there is no main object type to serve as an unambiguous case notion. Instead, one is forced to select an imperfect object type for this purpose, such as an order, and represent the event data accordingly. Once the log is recorded from this perspective, the information related to the other object types is lost, though. It is hence impossible to switch perspectives between object types or to analyze the relations between different objects in the process. Moreover, this "flat" recording leads to spurious behavioral relations in the log [2], which, for example, distort the results of automated process discovery techniques [12].

To overcome these issues, researchers recently introduced object-centric event logs, which can capture multiple types of concurrent objects in the process [2]. These object-centric logs allow process mining techniques to handle multi-object processes in a more appropriate manner. However, there is an abundance of event data captured in the form of classical, "flat" event logs, without access to the original data source from which these logs were extracted (cf. [8,9]). In this case, the only option is to transform the flat event logs into object-centric ones. Such a transformation is far from straightforward, because it requires knowledge about which object types occur in the event log, which object instances exist with which properties, and how these instances relate to the events. This information is to a certain extend contained in flat event logs, but in an unstructured, i.e., hidden way. Uncovering this information manually is a tedious and time-consuming task, considering the complexity of real-life logs, with dozens of attributes and thousands of events. Hence, the transformation from flat into object-centric event logs needs to be supported automatically.

Therefore, we propose an approach that automatically uncovers object-related information in flat event data and uses this information to transform the flat data into an object-centric event log according to the OCEL format [13]. For this purpose, our approach combines the semantic analysis of textual attributes in the flat event log with data profiling and control-flow-based relation extraction. In the following, Sect. 2 first illustrates the challenges that our approach needs to address, before we define preliminaries in Sect. 3. Our approach itself is presented in Sect. 4. Section 5 describes our evaluation, which shows that our approach is able to accurately rediscover flattened OCEL logs and can effectively mitigate quality issues in real-life logs. Section 6 summarizes related work; Sect. 7 discusses limitations and concludes the paper.

2 Problem Illustration

In this section, we illustrate the problems caused by recording object-centric event data in flat event logs and the challenges that must be overcome when transforming these logs into object-centric counterparts. For this, we use an established running example of an order handling process [2], which involves four types of objects: *customers*, *orders*, *items*, and *packages*. As visualized in Fig. 1, a customer can place multiple orders, an item belongs to exactly one order and one package, a package can contain multiple orders, and an order can be split over multiple packages.

Problems of Flat Event Logs. We illustrate the problems of recording a multi-object process in a flat format using the following trace, with an order as the case notion:

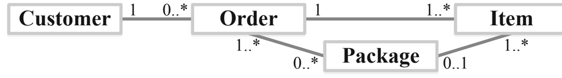

Fig. 1. UML data model of the running example.

t_{order} : ⟨Create order, Reorder item, Pick item, Send package, Pick item, Send package⟩.

The events in t_{order} indicate the picking of two items and the creation of two packages. Although their ordering suggests that these activities occur in an interleaving fashion, there is a clear relation between first picking an item and then sending it in a package. This clear precedence relation on the item level is lost, because there can be several items and packages per order, which we cannot distinguish on the trace level. This phenomenon, called *divergence*, is unavoidable when recording processes with object relations beyond 1:1 in the form of flat event logs [2]. It often occurs together with another unavoidable issue, called *convergence*. Convergence emerges when we use an individual item as the case notion to represent the events from trace t_{order}, which results in the following traces:

$t_{item}(1)$: ⟨Create order, Reorder item, Pick item, Send package⟩,
$t_{item}(2)$: ⟨Create order, Pick item, Send package⟩.

Because both items belong to the same order, the "*Create order*" event is duplicated across the traces. As a result, the information that both items relate to the same order is no longer captured at the trace level. Due to the m:n relations in the process at hand, the impact of this issue is amplified, given that also multiple orders can relate to the same package, and vice versa.

To overcome these issues and their associated information loss, a flat event log needs to be transformed into an object-centric counterpart, as discussed next.

From Flat to Object-Centric Logs. To illustrate the transformation of flat into object-centric event logs, consider the example in Table 1, which provides a flat event log with

Table 1. Flat event log of an order handling process with the *order* as the case notion.

CaseID	Event	Activity	Timestamp	PackageID	Weight	Customer
o1	e1	Create order	05-20 09:07			Pete
o1	e2	Reorder item	05-23 10:40		12.5	Pete
o1	e3	Pick item	05-23 14:20		70.8	Pete
o1	e4	Send package	05-23 17:26	p1	70.8	Pete
o1	e6	Pick item	06-04 15:20		12.5	Pete
o1	e9	Send package	06-06 16:20	p2	20.4	Pete
o2	e5	Create order	06-03 19:17			Pete
o2	e7	Update order	06-04 18:11			Pete
o2	e8	Pick item	06-05 11:48		7.9	Pete
o2	e10	Send package	06-06 16:20	p2	20.4	Pete

two orders. The log captures information on the events related to each order, as well as attributes that associate events with a PackageID, a Weight, and the Customer.

As shown in Table 2 and Table 3, constructing an object-centric version of this event log requires information about: object types (customers, orders, items, and packages), their instances and associated properties (e.g., that package *p1* has a weight of 70.8), and the relations between object instances and events (e.g., that event *e1* creates order *o1*, which relates to items *i1_1* and *i1_2*). However, such crucial information is not explicit in the flat version of the event log, but rather needs to be uncovered in order to transform flat data into an object-centric log. This results in four main transformation tasks:

Table 2. Object-centric event log of the running example.

Event	Activity	Timestamp	Orders	Packages	Items	Customer
e1	Create order	05-20 09:07	{*o1*}	∅	{*i1_1, i1_2*}	{*Pete*}
e2	Reorder item	05-23 10:40	{*o1*}	∅	{*i1_1*}	{*Pete*}
e3	Pick item	05-23 14:20	{*o1*}	∅	{*i1_2*}	{*Pete*}
e4	Send package	05-23 17:26	{*o1*}	{*p1*}	{*i1_2*}	{*Pete*}
e5	Create order	06-03 19:17	{*o2*}	∅	{*i2_1*}	{*Pete*}
e6	Pick item	06-04 15:20	{*o1*}	∅	{*i1_1*}	{*Pete*}
e7	Update order	06-04 18:11	{*o2*}	∅	{*i2_1*}	{*Pete*}
e8	Pick item	06-05 11:48	{*o2*}	∅	{*i2_1*}	{*Pete*}
e9	Send package	06-06 16:20	{*o1,o2*}	{*p2*}	{*i1_1, i2_1*}	{*Pete*}

Table 3. Objects of the object-centric event log.

Type	Instances
Customer	{*Pete* ()}
Order	{*o1* (), *o2* ()}
Package	{*p1* (Weight: 70.8), *p2* (Weight: 20.4)}
Item	{*i1_1* (Weight: 12.5), *i1_2* (Weight: 70.8), *i2_1* (Weight: 7.9)}

1. **Detect object types.** Object types in a process are not explicitly indicated in flat event logs. Rather, transformation requires these types to be extracted from unstructured activity labels, such as the *order* type in "*Create order*", and from certain event attributes, such as Customer in Table 1.
2. **Identify object instances.** Due to divergence and convergence, a transformation approach needs to identify distinct object instances within cases, e.g., that case *o1* deals with two items and two packages, and relate object instances across cases, e.g., that package *p2* appears in both *o1* and *o2*. This involves identifying event attributes that represent identifiers of a specific object, e.g., that PackageID defines individual packages. Furthermore, because such identifier attributes may not exist for all object types, it also requires inferring certain object instances from the event log itself, e.g., that events *e3* and *e6* yield two different items (*i1_1* and *i1_2*).

3. **Relate objects to their properties.** Flat event logs do not distinguish between attributes that relate to a specific event, such as a resource performing it, and attributes that provide information about the object handled in the event, such as the `Weight` attribute, which captures information about an individual package or item. When establishing an object-centric log, such relations must thus be derived by separating event attributes from object properties, in order to have a comprehensive view on the instances involved in the process, as captured in Table 3.
4. **Associate object instances with events.** Finally, instead of referring to a specific case, each event in an object-centric log must be mapped to the object instances it relates to. Obtaining a complete mapping requires a thorough analysis of the inter-relations that exist between object instances. For example, this requires the recognition that package *p2* relates to orders *o1* and *o2*, as well as items *i1_1* and *i2_1*, and associating all these objects with event *e9*, even though the objects originally stem from a range of different events and cases in the flat log.

In this paper, we propose an approach that tackles these tasks by combining the semantic analysis of the textual attributes of flat events logs with data profiling and control-flow-based relation extraction. In this manner, our approach uncovers object types, their instances and properties, as well as the relations between instances and events.

3 Preliminaries

We define objects, events, flat event logs, and object-centric event logs as follows based on the definitions by Van der Aalst [2].

Objects. An object is a tuple $o = (oi, ot, vmap)$, with oi as its identifier, ot its type, and $vmap$ a value map, which captures the assignment of values to o's properties. For instance, object *i1_1* has the identifier *i1_1*$.oi$ = "i1_1", the type *i1_1*$.ot$ = *item*, and has a value map assigning it a weight, *i1_1*$.vmap = \{$`Weight` : "12.5"$\}$.

Events. An event is a tuple $e = (a, ts, omap, vmap)$, with a its activity label, ts its timestamp, $omap$ the object map, which captures the objects that e relates to, and $vmap$ the value map, which assigns values to e's attributes. For instance, event *e1* in Table 2 has an activity label $e1.a$ = *Create order*, a timestamp $e1.ts$ = 05-20 09:07, an object map *e1*$.omap = \{$*o1,i1_1,i1_2*, *Pete*$\}$, and a value map *e1*$.vmap = \{$`Event` : "e1"$\}$.

Flat Event Logs. A flat event log L is a set of events that all have exactly one case identifier in their value map, i.e., $\forall_{e \in L}$`CaseID` $\in dom(e.vmap) \wedge |e.vmap($`CaseID`$)| = 1$, whereas their object maps are empty. Events that have the same `CaseID` are said to be part of the same *case*. Events belonging to the same case are assumed to have a total order, following, e.g., from their timestamps. For instance, event *e1* in Table 1 has a value map *e1*$.vmap = \{$`CaseID` : "o1", `Event` : "e1", `Customer` : "Pete"$\}$, whereas, all events in this log have an empty object map, e.g., *e1*$.omap = \emptyset$. Note that, for instance, *e9* in Table 2 would be part of two cases of a flat event log, when the *order* type serves as the case notion (cf. *e9* and *e10* in Table 1), because each event must have exactly one `CaseID`, yet, *e9* refers to two orders, *o1* and *o2* resulting in two case identifiers.

Finally, we define $Att = \bigcup_{e \in L} dom(e.vmap)$ as the set of attributes in L. For the log in Table 1, we get: $Att = \{\texttt{CaseID}, \texttt{Event}, \texttt{PackageID}, \texttt{Weight}, \texttt{Customer}\}$.

Object-Centric Event Logs. An object-centric event log O is a set of events that have populated object maps. For instance, event *e9* in Table 2 has an object map $e9.omap = \{o1, o2, p2, i1_1, i2_1, Pete\}$ and a value map $e9.vmap = \{\texttt{Event} : \text{"e9"}\}$. The events in O are assumed to have a known partial order, following, e.g., from their timestamps.

4 Approach

As visualized in Fig. 2, our approach for the transformation of a flat into an object-centric event log consists of five main steps. Step 1 extracts the *object types and actions* from the activity label and other textual attributes of events, which yields object types and the applied actions per event. Steps 2 and 3 jointly establish a set of object *instances*: Step 2 first matches extracted object types to attributes that capture identifiers to recognize distinct instances of object types, whereas Step 3 aims to discover instances for object types for which no such identifier attribute was found. Afterwards, Step 4 aims to assign *properties* to object types by identifying attributes that represent object properties. Finally, Step 5 assigns the discovered *object instances to events* by exploiting behavioral relations among object types and instances discovered in previous steps. Based on the result of Step 5, we create an object-centric event log according to the *OCEL* format [13]. In the remainder, we describe each of these five steps in detail.

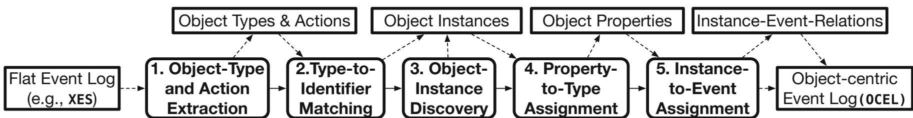

Fig. 2. Overview of the approach.

4.1 Step 1: Object-Type and Action Extraction

The first step of our approach extracts the object types and actions from an event log. As illustrated in Sect. 2, object types need to be derived from unstructured textual attribute values, such as activity labels, and attribute names of a flat event log. An *action* is applied to an object, incurring a change in its state [14]. For instance, the "*Create order*" activity label indicates that a *create* action is applied to an *order*. We extract actions along with the object types since these can contain information about the creation of new object instances, which we will exploit in a later step.

To achieve this, we use a semantic extraction technique from our earlier work [22]. This technique extracts parts of textual attribute values that correspond to different semantic roles, such as *object types* and *actions* in two ways:

1. *Instance-level labeling*: The extraction technique labels parts of unstructured textual attribute values with semantic roles. The parts that correspond to the desired roles are then extracted. For instance, for the "*Send package*" activity label of event *e9*, the technique labels "*package*" as an object type and "*send*" as an action.
2. *Attribute-level classification*: The extraction technique also identifies event attributes that in their entirety correspond to a certain semantic role. It does so based on an attribute's name and its value range. This, e.g., applies to the `Customer` attribute in Table 1, which allows us to also identify *customer* as an object type contained in the event log, assigning this type to any event that has a value for the attribute.

By taking the output of this extraction technique, Step 1 instantiates a function $extract$, which, given an event $e \in L$, extracts the object types and the actions applied to them (if any) from e. The result maps the object types to a (possibly empty) set of actions, e.g., $extract(e9) = \{package \rightarrow \{send\}, customer \rightarrow \emptyset\}$. Each event's object map is then initialized with its object types, e.g., $e9.omap = \{package \rightarrow \emptyset, customer \rightarrow \emptyset\}$. Finally, we establish a set of identified object types $T = \bigcup_{e \in L} dom(e.omap)$ and move to Step 2, which aims to match these types to identifier attributes.

4.2 Step 2: Type-to-Identifier Matching

In this step, our approach tries to associate identifier attributes with the extracted object types to be able to recognize distinct object instances. For our running example, we can differentiate between the two packages *p1* and *p2* by recognizing that the `PackageID` is an identifier for *package* objects. Such identifier attributes are not explicitly given, meaning that we need to match object types to attributes. To establish these matches, our approach first identifies a set Att_{ID} of potential object identifiers, by categorizing attributes according to their domain. Then, we match these attributes to object types in T, resulting in a mapping M, consisting of (ot, att) pairs with $ot \in T$ and $att \in Att_{ID}$.

Finding Potential Identifier Attributes. To identify the set $Att_{ID} \subseteq Att$, we recognize that identifiers generally use alphanumeric domains, i.e., `string` or `int`, such as `PackageID` ("*p1*" and "*p2*") and `Customer` ("*Pete*") in Table 1. Therefore, we categorize attributes according to their domain's data type and add those with `string` and `int` domains to Att_{ID}. In this manner, we discard attributes corresponding to, e.g., timestamps, boolean values, and floats, such as the `Weight` attribute.

Matching Identifier Attributes and Object Types. Next, we aim to identify matches between the object types in T and potential identifier attributes in Att_{ID}, resulting in the set M. Some object types and attributes can be directly matched. For others, we first establish candidate matches, and then verify the validity of these candidates.

Direct Matching. Object types in T that stem from the attribute-level classification of Step 1 reflect objects that correspond to the name of a specific event attribute, such as the *customer* object type corresponding to the `Customer` attribute. Because these types were identified in this manner, we know that their identifiers are captured in the corresponding attributes, if indeed these are part of Att_{ID}. Therefore, we can directly add such pairs, e.g., (*customer*, `Customer`), to the matches in M.

Establishing Candidate Matches. For object types that cannot be directly matched, such as the *package* type extracted from activity labels, we first establish candidate matches, collected in a set M_C, using two strategies, considering attribute names and values.

First, we establish candidate matches by checking if the name of an unmatched object type encompasses the name of an unmatched attribute, or vice versa. In this manner, we recognize a candidate match between the *package* type and the `PackageID` attribute, or between the *item* type and a, hypothetical, `order_item` attribute.

Then, for attributes in Att_{ID} that are not yet in a candidate match with an object type, we apply a strategy inspired by *data profiling* [4], which checks if an attribute exclusively *co-occurs* with an object type. For the running example, all events associated with the *package* type (*e4*, *e9*, *e10*) have values for the `PackageID` attribute, but this attribute does not apply to any other events. Therefore, even if `PackageID` was named `pID` and hence not a name-based candidate match, our approach would still be able to recognize it as a potential identifier for the *package* type and add it to M_C.

Validating Candidate Matches. Although name-based similarity and co-occurrence are useful indicators to identify relations between object types and attributes, there is no guarantee that the candidate matches actually capture proper identifiers. Therefore, we next validate each candidate match $(ot, att) \in M_C$ by determining if each unique value of att is indeed associated with a specific instance of ot, and vice versa.

This validation task is complex, though, given that multiple events in a case can relate to the same object instance (e.g., creating and updating an order) or multiple instances of the same object type (e.g., shipping multiple packages for one order), and that, due to duplication issues, the same event can essentially appear in two cases (cf. *e9* and *e10*). For an object type *ot*, we deal with these issues by aiming to establish a set of events $E'(ot)$ that should each relate to a different instance of *ot*. Given $E(ot) \subseteq L$ as the events related to *ot* (i.e., that have *ot* in their *omap* after Step 1), we obtain $E'(ot)$ by avoiding duplicate events and by selecting only a single event per case. Our approach avoids duplicates by only selecting events from $E(ot)$ that have a unique combination of an activity label, timestamp, and event attributes (aside from their `CaseID` and event ID, if available). In this manner, we detect *e9* and *e10* as duplicates. Given the identified duplicates, we select a single event per case related to *ot*, in a manner that maximizes the size of $E'(ot)$. For instance, given $E(package) = \{e4, e9, e10\}$, we select *e10*, because *e4* and *e9* stem from the same case, and obtain $E'(package) = \{e4, e10\}$.

Finally, if the attribute values of att are unique for the events in $E'(ot)$, we consider att as a valid identifier of *ot* and add (ot, att) to M. For instance, we consider (*package*, `PackageID`) a valid match, given that the two events in $E'(package)$ have unique values for the attribute, "*p1*" and "*p2*". If there are multiple valid candidates for the same object type, we match the type to the attribute with the largest number of unique values and discard the other candidates.

Object-Instance Creation. For all matches $(ot, att) \in M$, our approach creates an object instance o with its type *ot* for each unique value of att, i.e., its identifier oi, and adds these instances to the object maps of the events that refer to this instance. For example, we add package *p1* to event *e4* and package *p2* to events *e9* and *e10*.

4.3 Step 3: Object-Instance Discovery

Next, we aim to discover instances for those object types for which no explicit identifier attribute was found in the previous step, such as the *item* type in the running example. For this, we try to find activities that indicate the instantiation of objects, either based on their activity labels or based on the life cycle of an object.

Instance Discovery Based on Creation Actions. We first aim to identify activities whose meaning hints at the instantiation of an object, such as "*Create order*". To this end, we use the action classification framework of the MIT Process Handbook [19], which defines a set of 15 *creation actions* (see Table 4) describing the creation of some output.

Given an event, we check if any of its actions, extracted in Step 1, corresponds to an action in this set. If so, the occurrence of this event implies the instantiation of a new object. For instance, we recognize that events *e1* and *e5*, corresponding to the "*Create order*" activity label, result in two new orders.

Table 4. Creation actions [19] used by Step 3.

build	*compute*	*construct*	*copy*	*create*
design	*develop*	*document*	*duplicate*	*generate*
make	*manufacture*	*perform*	*produce*	*record*

Although we here identify creation actions based on the 15 actions from the MIT Process Handbook, our work is independent of this specific resource. It can be replaced or enhanced with alternatives, such as the *build verbs* from the classification framework by Levin [15], multilingual resources, such as *ConceptNet* [23], or a self-defined set.

Instance Discovery Based on Object Life Cycles. Although creation actions are a reliable indicator for the creation of new objects, they are not always available for an object type. Therefore, our approach next analyzes the life cycles of object types in terms of the applied activities per case. To illustrate this, consider the life cycles in Fig. 3.

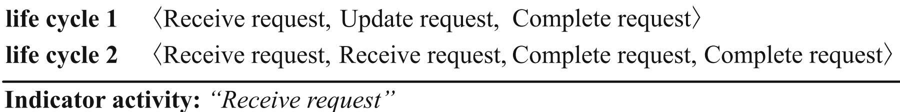

Fig. 3. Recognizing activities that indicate new object instances.

For an object type *ot*, we aim to identify an *indicator activity*, which corresponds to a new object instance. We look for such an indicator by checking if there are any activities related to *ot* that occur for every case of this type. For example, assuming only the two depicted cases relate to requests in the process at hand, both "*Receive request*" and "*Complete request*" are candidate activities, since they occur in both life cycles. In case of such a tie, we select the activity that most commonly occurs first among the candidates—"*Receive request*" in the example—as the activity that we use

to identify new object instances. Therefore, we recognize that the cases in Fig. 3 relate to three distinct *request* objects: one in the first life cycle and two in the second.

Object-Instance Creation. For each event that indicates a new object instance, based on a creation action or indicator activity, our approach establishes an object instance, for which we generate a unique identifier oi, and add it to the event's object map. Duplicate events, as identified in Step 2, form an exception here. Since they correspond to the creation of the same object instance, which is why we assign the same instance to them.

Note that we discard all object types for which neither Step 2 nor Step 3 identified any instances, by removing the type from T as well as from any event's object map.

4.4 Step 4: Property-to-Type Assignment

In this step, our approach tries to associate properties to object types, which are attributes that capture information about an object instance associated with an event, rather than relate to the event itself. For instance, although event *e3* ("*Pick item*") has a `Weight` attribute with a value of "70.8", it is clear that this refers to the weight of the item, not of the event. Therefore, in this step we aim to establish a mapping between a log's attributes Att and the object types in T.

To establish this mapping, we first select all attributes that were not recognized as object identifiers in Step 2. Then, we consider an attribute att to be a property of an object type *ot* if (1) events related to *ot* have a value for att and (2) all events related to the same object instance have the same value for att. The former ensures co-occurrence, ascertaining that att indeed relates to *ot*, whereas the latter ensures that object properties are immutable per object instance, in line with their definition in the OCEL format. In this manner, we identify that `Weight` is an attribute of both the *item* and *package* types, whereas attributes such as a `timestamp` or `employee` are not identified as properties, because they change across the events related to the same object instance.

Finally, we avoid assigning an attribute att as a property to multiple object types if the attribute name indicates a clear relation to one of the types. For example, we avoid assigning an `item_category` attribute to the *package* type, given that this property clearly relates to items, irrespective of the co-occurrence of the attribute and packages.

4.5 Step 5: Instance-to-Event Assignment

Finally, our approach aims to complete the mapping between events and object instances, which is necessary to account for missing *instance-to-event* and *instance-to-instance* relations. The former involves events that correspond to a particular object type, but for which no particular instance has yet been discovered. For example, event *e2* ("*Reorder item*") is already recognized as relating to the *item* type, yet we still need to identify that this event refers to the same item that is later handled by event *e6* ("*Pick item*"). The latter refers to the inter-relations that can exist among object instances, which need to be reflected in the object maps of the corresponding events. For example, since package *p1* relates to order *o1*, event *e4*, which creates this package, should also be associated with that order. We identify these missing relations as follows.

Finding Missing Instance-to-Event Relations. To find missing instance-to-event relations, we first identify the events that are associated with an object type (through Step 1), but for which no instance was discovered in Step 2 or 3. This applies, e.g., to event *e2* ("*Reorder item*") and *e7* ("*Update order*"). Then, given such an event, we search within the case for other events that are associated with an object instance of the same type and verify that the object's properties match across the events. For instance, since event *e2* has a `Weight` of 12.5, we do not want to associate it with the item of event *e3*, which has a weight of 70.8, but rather with the same item as event *e6*, which also relates to an item weighing 12.5 kg. Should multiple object instances satisfy this requirement, we associate the event to the instance of its nearest predecessor or successor.

Finding Missing Instance-to-Instance Relations. We look for instance-to-instance relations by (1) considering relations between instances and cases, (2) identifying strict orders among object types, and (3) consolidating cross-case relations.

Discovering Case Objects. We first exploit that, commonly, each case in a flat event log corresponds to an instance of a particular object type, such as an *order* in our running example. If such a *case object* can be identified, we know that any other object instance handled in the same case also relates to that object, e.g., that the items and packages handled in the first case all relate to order *o1* as well.

However, to recognize such inter-relations, we need to identify the case object type, if any, for a particular event log. Given that instances of this object type must, by definition, be in a 1:1 relation with the cases in a log, we first discount all object types for which this does not apply, i.e., which are affected by convergence and divergence issues. Given an object type ot, we thus ensure that (1) no instance of ot is associated with multiple cases in the log, such as the *package* type in the example, and (2) that no case in the log is associated with multiple instances of ot, such as the *item* type.

If these checks yield a single case object type, ot_c, then we add each object instance of that type to the object map of all events e in their case, using `CaseID` as the instance's identifier. For the example, *order* is the only object type that passes the checks, which means that we assign orders *o1* and *o2* to all events in their respective cases.

Strict Order Between Object Types. We next identify instance-to-instance relations by looking for the existence of strict orders between object types. Here, we consider an object type ot_1 to be in a strict order with type ot_2 if every time an event related to ot_2 occurs, an event related to ot_1 (directly or indirectly) precedes it. In this manner, we, for example, observe a strict order between items and packages in the running example.[1]

Given such a strict order between ot_1 and ot_2, we relate an instance o_1 of type ot_1 to an instance o_2 of ot_2 if the life cycle of o_1 completes before the life cycle of o_2 begins, i.e., if the last event related to o_1 comes before the first event related to o_2. For example, we relate item *i1_1*, which last occurs in *e6*, to package *p2*, which first occurs in *e9*.

Consolidating Cross-Case Relations. Last, we consolidate inter-relations across cases by ensuring that duplicate events are associated with the same sets of object instances.

[1] Note that these object types can still occur in an interspersed manner, as e.g., seen in case *o1*, where events related to items also occur in between packages.

Given two duplicates, e and e', we achieve this by associating both events with all object instances stemming from the union of their object maps. For example, having recognized that events *e9* and *e10* are duplicates, we add all object instances stemming from case *o1* (associated with *e9*) to the object map of *e10*, and vice versa. In this manner, we, e.g., recognize that package *p2*, which is created by these duplicate events, deals with item *i1_1* (stemming from case *o1*) as well as item *i2_1* (stemming from *o2*).

Having associated events with all object instances, our approach has uncovered the necessary information to construct its output, an object-centric event log.

4.6 Output

Our approach returns an object-centric event log according to the OCEL format [13], which, at a high level, consists of an *objects* and an *events* map.

The *objects* map relates object identifiers to instances, which are in turn associated with their type and property values. To populate this map, we add all instance identifiers, either detected in Step 2 or generated in Step 3, to the map and associate these instances with their properties identified in Step 4. Simultaneously, we also disassociate any object property from the events that they were associated with in the flat event log, e.g., rather than having `Weight` as an attribute of event *e4*, we represent it as a property of the respective package: $objects[p1] = \{package, \{\texttt{Weight:}\text{“70.8”}\}\}$.

The *events* map associates identifiers with events, which are associated with object instances through their $omap$. It is important to recognize that these events are no longer grouped per case. As a result, we can omit any duplicate event from consideration, e.g., by removing event *e10* and preserving *e9*. The map is then populated with the remaining events, which are each associated with the identifiers of their respective object instances, as assigned in Steps 2, 3, and 5, e.g., $e1.omap = \{\text{“o1”}, \text{“i1_1”}, \text{“i1_2”}\}$.

Based on the established maps, we return the object-centric log, which can directly be used by object-centric process mining techniques [3,17].

5 Evaluation

We implemented our approach as a Python prototype[2], using the PM4Py library [7] for event log handling. Based on this prototype, we perform evaluation experiments to assess our approach's capability to rediscover artificially flattened object-centric logs (Sect. 5.1). Then, we illustrate its practical value by showing that it can resolve divergence and convergence in real-life scenarios (Sect. 5.2). Finally, we discuss the main insights from our evaluation and its limitations (Sect. 5.3).

5.1 Rediscovering Object-Centric Event Logs

We assess whether our approach is able to rediscover an artificially flattened object-centric log by comparing its output with the original OCEL log.

[2] https://gitlab.uni-mannheim.de/processanalytics/uncovering-object-centric-data.

Data. For our evaluation experiments, we use a publicly available OCEL log of an order handling process.[3] Currently, this is the only available log suitable for this evaluation. It contains 22,367 events and 11,522 object instances of five object types: 2,000 orders, 8,159 items, 20 products, 17 customers, and 1,325 packages. From this original OCEL log, we create three flattened logs, using the *item*, *order*, or *package* as the case notion. The resulting logs capture 1:n, n:1, and n:m relationships between objects and include object types both in attribute names and activity labels. Thus, all relation types are covered, meaning that all strategies employed by our approach can be assessed.

Setup. To assess the ability of our approach to correctly discover relevant object-centric information in the flat event log, we conduct experiments using two settings:

(1) All attributes. In this setting, we use all information from the flattened event logs as input for the rediscovery task.

(2) Masked ID attributes. To assess the robustness of our approach, we also purposefully reduce the information that is available by masking object ID attributes in the flattened event logs. This increases the dependency of our approach on its instance discovery techniques employed in Step 3. Specifically, we mask each ID attribute once for each of the three flattened logs. Since the *item* and *order* logs include identifiers for all four other types, and the *package* log captures only a customer identifier, we obtain nine *masked* logs, one with *package*, four with *item*, and four with *order* as the case notion.

We measure the performance of our approach in terms of the well-known *precision*, *recall*, and F_1*-score* metrics with respect to the original OCEL log per type of element, i.e., object types, object instances, properties, and instance-to-event assignments. Using A to denote the set of elements uncovered by our approach and G for the set of elements in the ground truth, i.e., the OCEL log, precision is the fraction of elements uncovered by our approach that are actually correct ($|A \cap G|/|A|$), recall is the fraction of elements in the OCEL log that were also correctly uncovered by our approach ($|A \cap G|/|G|$), and the F_1-score is the harmonic mean of precision and recall. Because flattening the log causes a loss of information about entire object types, we only include object types in G that are actually contained in a particular flattened log. To avoid propagating false positives from object-type extraction (Step 1), we only include elements in A that relate to object types actually present in the original OCEL log for the other steps.

Results. Table 5 reports on the results of our rediscovery experiments, micro-averaged over the logs for the respective settings. In the following, we discuss the results for the different tasks that our approach addresses.

Object-Type Extraction. For the extraction of object types, our approach achieves a recall of 1.00 and a precision of 0.71, yielding an F_1-score of 0.82. We thus accurately identify all object types from the original log. The lower precision is caused by the extraction of two additional object types, *payment reminder* and *delivery*. Although not contained in the original OCEL log, their extraction is not problematic and can even enable additional insights, e.g., on the number of payment reminders sent per order.

Object-Instance Identification. Our approach identifies object instances with perfect accuracy in both the regular and masked settings. This highlights its ability to find and

[3] http://ocel-standard.org/1.0/running-example.jsonocel.zip.

match ID attributes to object types (Step 2) and the usefulness of our instance-discovery strategies (Step 3), which can identify instances for types with masked ID attributes.

Property-to-Type Assignment. When assigning properties to object types, our approach achieves a perfect recall, but a rather low precision of 0.37. An in-depth look reveals that these different assignments are not problematic, though. For example, the attribute `cost` is assigned to both *product* and *item*, whereas in the original it is only associated with products. However, given that also items have costs, such assignments are redundant, but not wrong. Similarly, our approach associates attributes such as `price` and `weight` with orders, items, packages, and products. While these are realistic assignments, the attributes are not considered as properties in the original OCEL log, but are associated with events. Thus, our approach actually provides a more complete mapping.

Instance-to-Event Assignment. For instance-to-event assignment, we achieve an excellent recall (0.998, rounded in Table 5) and a high precision (0.94) in the all-attributes setting. Thus, our approach assigns relevant object instances to events they relate to. An in-depth look into the constructed OCEL logs reveals that the superfluous assignments of instances to events are mainly assignments of packages to events that relate to items shipped in the respective package. Such assignments are not considered in the original log, but can enable insights into the packaging process in a post-hoc analysis.

When masking identifier attributes, precision and recall decrease slightly, which indicates that our approach occasionally makes incorrect assignments. This is especially the case for 1:n relationships between object types and the case notion. For example, in the *order* event log, where one order may contain many different items, items with the same properties may be assigned incorrectly. However, it is important to recognize that such assignments are simply not possible based on the information in the masked log, whether done by an automated approach or manually.

Table 5. Results of the evaluation experiments averaged over flattened logs.

Element	All attributes				Masked ID attribute			
	Count	Precision	Recall	F_1-score	Count	Precision	Recall	F_1-score
Object types	12	0.71	1.00	0.82	42	0.71	1.00	0.82
Object instances	24k	1.00	1.00	1.00	94k	1.00	1.00	1.00
Object properties	10	0.37	1.00	0.54	34	0.39	1.00	0.56
Instance-to-event	411k	0.94	1.00	0.97	1,559k	0.93	0.97	0.95

5.2 Real-Life Application Cases

We next demonstrate the practical value of our approach by showing that it is capable of resolving convergence and divergence issues in well-known real-life event logs. The full results and OCEL logs obtained by our approach can be found in our repository (see Page 13). In the following, we use individual cases and events from these logs to illustrate in detail how our approach mitigates divergence and convergence.

Divergence. We use the BPI17 application log [8] to show how our approach mitigates divergence issues. The log captures a loan application process, containing 1,202,267 events, 31,509 cases, and 26 distinct activities. Divergence is particularly frequent, because the log uses the *application* as the case notion and one application can have multiple offers. This means that cases in the log often contain multiple events that denote execution of the same activity for distinct offers (divergence). Applying process discovery to the log leads to loop-backs, as visualized in Fig. 4. This shows the directly-follows graph (DFG) discovered for one case of the log, which is already quite complex.

When applying our approach to mitigate the divergence issue, we discover that 42,995 offers are handled in the 31,509 applications and that offers have several properties, such as an offered amount and a monthly cost. For the particular case in Fig. 4, we find that four distinct offers are handled in this application, that these all have different properties, and that the process is linear with respect to a single offer, e.g., ⟨*Create Offer*, *O Created*, *O Sent*, *O Canceled*⟩. It is important to stress that this information on the sub-case level is not readily available in the flat log and has to be uncovered by identifying the distinct offers handled in a single case. Our approach achieves this by extracting the *offer* type, finding an identifier attribute for it, and assigning it, among others, the `MonthlyCost` and `OfferedAmount` properties.

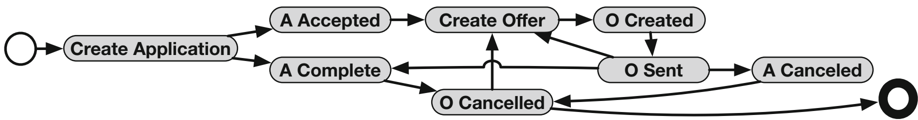

Fig. 4. Directly-follows graph of application 196483749 of the BPI17 log.

Convergence. To illustrate how our approach can mitigate convergence issues, we chose the BPI19 event log [9], which captures data on the purchasing process of a multinational company and contains 1,595,923 events across 251,734 cases with 42 distinct activities. Each event relates to a single *purchase order item* and multiple purchase order items can belong to the same *purchasing document*. Consequently, events on the purchasing-document level are duplicated across cases (convergence). For example, the duplication of "*Vendor creates invoice*" events suggests the creation of invoices per purchase order item, whereas in reality invoices can cover multiple such items.

When applying our approach to mitigate convergence, we discover, among others, 251,734 purchase order items, 76,349 purchasing documents, 86,868 invoices, 1,975 vendors, and 4 companies. The resulting OCEL log reveals the relationships between object types, as shown in Fig. 5: Purchasing documents consist of any number of purchase order items, a vendor creates multiple invoices, and each invoice is associated with one purchasing document. Notably, in contrast to the input log, events related to purchasing documents and invoices, such as "*Vendor creates invoice*" or "*Document created*", are not captured at the level of individual purchase order items, but at the level of purchasing documents, thus eliminating duplicate events. This demonstrates

that our approach is able to reveal actual relationships among objects and mitigate the convergence issue present in real-life logs.

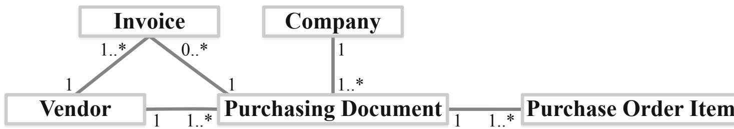

Fig. 5. UML data model with relations between object types found in the BPI19 log.

5.3 Discussion

Our evaluation experiments show that our approach is capable of accurately uncovering object-centric information from an artificially flattened object-centric event log, using different settings and case notions. We also observe that our approach even uncovers more information than originally captured in the OCEL log. This includes additional object types, properties, and relations, which allow for deeper insights into the process. The main difficulty for our approach was the recognition of object inter-relations for objects in a 1:n relation with the case object, which resulted in several incorrect instance-to-event assignments. Despite their promise, the evaluation results must be considered with care, given that only one original OCEL log was available as a basis.

The real-life application cases demonstrate that our approach can mitigate divergence and convergence in real-life event logs. Although, due to a lack of a ground truth, the completeness of uncovered object-centric information cannot be quantified, the results nevertheless show that our approach provides considerable practical value by extending the analysis potential for flat event logs of multi-object processes.

6 Related Work

Our work primarily relates to research on object-centric representations of event data and discovering object-centric information from event logs.

After storing event data in flat formats like XES [5] for many years, the first data format proposed for object-centric event logs was XOC [16], which does not require a case notion and therefore avoids flattening multi-dimensional data. More recently, researchers introduced the OCEL format [13], which allows for more efficient storage and processing than its predecessor. Beyond log formats, another proposed option for storing multi-dimensional object-centric event data are event graphs, which enable the analysis of behavior of different objects handled in a process [11]. For our approach, we adopt OCEL as the output format, which, among others, enables the subsequent application of techniques for discovering object-centric process models, such as object-centric behavioral constraint models [17] and object-centric Petri nets [3].

Approaches for the discovery of object types and their behavioral relations from event data usually require relational data or rich logs that cover multiple perspectives of a process as input. This includes approaches for the discovery of artifact (i.e., object)

life cycles from raw logs of artifact-centric systems [10,21] as well as the discovery of behavioral dependencies between object types based on such logs [20] or based on data extracted from ERP systems [18]. Compared to these approaches, our approach takes flat event logs, where no explicit relations between objects are given, and transforms them into object-centric logs. The approach by Bano et al. [6] also uses flat event logs as input data, but their goal is to discover UML models from activity labels and attribute names to provide analysts with domain-specific context information.

7 Conclusion

In this paper, we proposed an approach to uncover object-centric data from flat event logs to automatically transform them into object-centric logs according to the OCEL format. To this end, our approach combines the semantic analysis of textual attributes with data profiling and control-flow-based relation extraction. It extracts object types, discovers object instances and their properties, and assigns these instances to events they relate to. We demonstrated our approach's efficacy in an evaluation by showing that it is able to rediscover an artificially flattened object-centric log and that it can mitigate convergence and divergence issues in real-life event logs.

Our approach is subject to certain limitations. First, object types must at least be mentioned in the flat log for our approach to extract them. However, once an object type is extracted, instances, properties, and relations can be identified through the use of diverse strategies that include and go beyond the semantic analysis of events. Second, to accurately handle n:1 and m:n relations with respect to the case notion, our approach relies on duplicate detection, which requires (non-duplicate) events to have discriminative timestamps or attribute values. Finally, because the assignment of objects to events often depends on domain knowledge about inter-object relations, our approach can currently not handle all scenarios. For example, it is not clear without domain knowledge that items shall relate to packages but not vice versa. Nevertheless, as our evaluation shows, our approach achieves promising results and thus provides an important contribution towards the applicability of object-centric process mining.

In future work, we aim to give domain experts the option to provide input regarding the higher-level relations between different object types in the form of rules. For example, they could state that, if an event results in the creation of an order, it also relates to the items of that order. Then, only assignments that adhere to these rules could be made. Moreover, we want to integrate common-sense knowledge into our approach, which could help to derive relations between object types through their meaning.

Reproducibility: The implementation, employed data, and obtained OCEL logs are available through the repository linked in Sect. 5.

References

1. van der Aalst, W.: Process Mining: Data Science in Action. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49851-4
2. van der Aalst, W.M.P.: Object-centric process mining: dealing with divergence and convergence in event data. In: Ölveczky, P.C., Salaün, G. (eds.) SEFM 2019. LNCS, vol. 11724, pp. 3–25. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30446-1_1

3. van der Aalst, W., Berti, A.: Discovering object-centric petri nets. Fundamenta Informaticae **175**(1–4), 1–40 (2020)
4. Abedjan, Z., Golab, L., Naumann, F.: Profiling relational data: a survey. VLDB J. **24**(4), 557–581 (2015). https://doi.org/10.1007/s00778-015-0389-y
5. Acampora, G., Vitiello, A., Di Stefano, B., van der Aalst, W., Günther, C., Verbeek, E.: IEEE 1849tm: the XES standard. IEEE Comput. Intell. Mag. 4–8 (2017)
6. Bano, D., Weske, M.: Discovering data models from event logs. In: Dobbie, G., Frank, U., Kappel, G., Liddle, S.W., Mayr, H.C. (eds.) ER 2020. LNCS, vol. 12400, pp. 62–76. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-62522-1_5
7. Berti, A., van Zelst, S., van der Aalst, W.: Process mining for python (PM4Py): bridging the gap between process-and data science. In: ICPM Demo Track, pp. 13–16. CEUR-WS (2019)
8. van Dongen, B.: BPI Challenge (2017). https://doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b
9. van Dongen, B.: BPI Challenge (2019). https://doi.org/10.4121/uuid:d06aff4b-79f0-45e6-8ec8-e19730c248f1
10. van Eck, M., Sidorova, N., van der Aalst, W.: Guided interaction exploration in artifact-centric process models. In: Business Informatics, pp. 109–118. IEEE (2017)
11. Esser, S., Fahland, D.: Multi-dimensional event data in graph databases. J. Data Semant. **10**(1), 109–141 (2021)
12. Fahland, D.: Artifact-centric process mining. In: Sakr, S., Zomaya, A.Y. (eds.) Encyclopedia of Big Data Technologies, pp. 108–117. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-77525-8_93
13. Ghahfarokhi, A.F., Park, G., Berti, A., van der Aalst, W.M.P.: OCEL: a standard for object-centric event logs. In: Bellatreche, L., et al. (eds.) ADBIS 2021. CCIS, vol. 1450, pp. 169–175. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85082-1_16
14. Leopold, H., van der Aa, H., Offenberg, J., Reijers, H.A.: Using hidden Markov models for the accurate linguistic analysis of process model activity labels. Inf. Syst. **83**, 30–39 (2019)
15. Levin, B.: English Verb Classes and Alternations: A Preliminary Investigation. University of Chicago Press, Chicago (1993)
16. Li, G., de Murillas, E.G.L., de Carvalho, R.M., van der Aalst, W.M.P.: Extracting object-centric event logs to support process mining on databases. In: Mendling, J., Mouratidis, H. (eds.) CAiSE 2018. LNBIP, vol. 317, pp. 182–199. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92901-9_16
17. Li, G., de Carvalho, R.M., van der Aalst, W.M.P.: Automatic discovery of object-centric behavioral constraint models. In: Abramowicz, W. (ed.) BIS 2017. LNBIP, vol. 288, pp. 43–58. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59336-4_4
18. Lu, X., Nagelkerke, M., Van De Wiel, D., Fahland, D.: Discovering interacting artifacts from ERP systems. IEEE Trans. Serv. Comput. **8**(6), 861–873 (2015)
19. Malone, T., Crowston, K., Herman, G.: Organizing Business Knowledge: The MIT Process Handbook. MIT Press, Cambridge (2003)
20. Popova, V., Dumas, M.: Discovering unbounded synchronization conditions in artifact-centric process models. In: Lohmann, N., Song, M., Wohed, P. (eds.) BPM 2013. LNBIP, vol. 171, pp. 28–40. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06257-0_3
21. Popova, V., Fahland, D., Dumas, M.: Artifact lifecycle discovery. Int. J. Coop. Inf. Syst. **24**(01), 1550001 (2015)
22. Rebmann, A., van der Aa, H.: Extracting semantic process information from the natural language in event logs. In: Advanced Information Systems Engineering, pp. 57–74 (2021)
23. Speer, R., Chin, J., Havasi, C.: Conceptnet 5.5: an open multilingual graph of general knowledge. In: Thirty-First AAAI Conference on Artificial Intelligence (2017)