Discovering Interacting Artifacts from ERP Systems (Extended Version)

Xixi Lu[1], Marijn Nagelkerke[2], Dennis van de Wiel[2], and Dirk Fahland[1]

[1] Eindhoven University of Technology, The Netherlands
[2] KPMG IT Advisory N.V., Eindhoven, The Netherlands.
(x.lu,d.fahland)@tue.nl
(Nagelkerke.marijn,vandewiel.dennis)@kpmg.nl

Abstract. The omnipresence of using Enterprise Resource Planning (ERP) systems to support business processes has enabled recording a great amount of (relational) data which contains information about the behaviors of these processes. Various process mining techniques have been proposed to analyze recorded information about process executions. However, classic process mining techniques generally require a linear event log as input and not a multi-dimensional relational database used by ERP systems. Much research has been conducted into converting a relational data source into an event log. Most conversion approaches found in literature usually assume a clear notion of a case and a unique case identifier in an isolated process. This assumption does not hold in ERP systems where processes comprise the life-cycles of various interrelated data objects, instead of a single process. In this paper, a new semi-automatic approach is presented to discover from the plain database of an ERP system the various objects supporting the system. More precisely, we identify an *artifact-centric process model* describing the system's objects, their life-cycles, and detailed information about how the various objects synchronize along their life-cycles, called *interactions*. In addition, our artifact-centric approach helps to eliminate ambiguous dependencies in discovered models caused by the *data divergence* and *convergence* problems and to identify the exact "abnormal flows". The presented approach is implemented and evaluated on two processes of ERP systems through case studies.

Keywords: Process Discovery, Artifact-Centric Processes, Outlier Detection, Relational Data, Log Conversion, ERP Systems

1 Introduction

Information systems (IS) not only store and process data in an organization but also record *event data* about how and when information changed. This "historical event data" is often used to analyze, for instance, whether information processing in the past conformed to the processes in the organization or to compliance regulations. For example, has each order by a gold customer been delivered with priority shipping, or have all delivery documents been created before creating the invoice? The manual analysis of historic event data is time consuming and error-prone as often hundreds of thousands of records need to be checked.

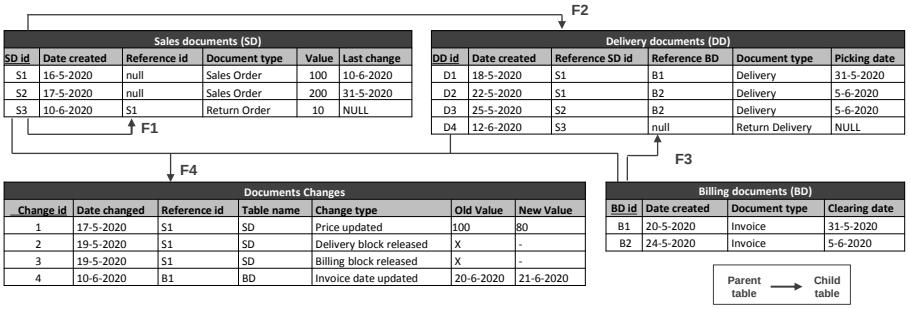

Fig. 1: The tables of the simplified OTC example

Process mining [1] offers automated techniques for this task. The most prominent technique is to discover from historical event data a graphical process model describing all historic behaviors; the discovered model can be visually explored to identify the main flows and the unusual flows of the process. Process analyst and domain expert can then for instance identify the historic events that correspond to unusual flows, investigate circumstances and possible causes for this behavior, and devise concrete measures to improve the process [2]. The success of the analysis often depends on whether unusual behavior is easy to distinguish visually from normal behavior. Prerequisite to this analysis is a *process event log* that describes how all information changes occurred from the perspective of a particular process; its underlying assumption is that each event can unambiguously be mapped to a particular case of the process.

1.1 Problem Description

In the more general case, information access is not tied to a particular case of a process; rather the same information can be accessed and changed from various processes and applications. A typical example are *Enterprise Resource Planning* (ERP) systems which organize all information in *documents* related to each other through one-to-many and many-to-many relations; information changes occur in *transactions* and the completion of a transaction is logged as an event also called *transactional data*. All data relevant for the analysis is stored in a relational database.

Figure 1 shows a simplified example of the transactional data of an *Order to Cash* (OTC) process supported by SAP systems; Fig. 2 visualizes the events stored in these tables related to the creation of documents. There are two sales orders S1 and S2; creation of S1 is followed by creation of a delivery document D1, an invoice B1, another delivery document D2, and another invoice B2 which also contains billing information about S2. Creation of S2 is also followed by creation of another delivery document D3. Further, there is a return order S3 related to S1 with its own return delivery document D4. The many-to-many relations between documents surface in the transactional data: a sales document can be related to multiple billing documents (S1 is related to B1 and B2) and a billing document can be related to multiple sales document (B2 is related to S1 and S2). This behavior already contains an *unusual flow*: two times delivery doc-

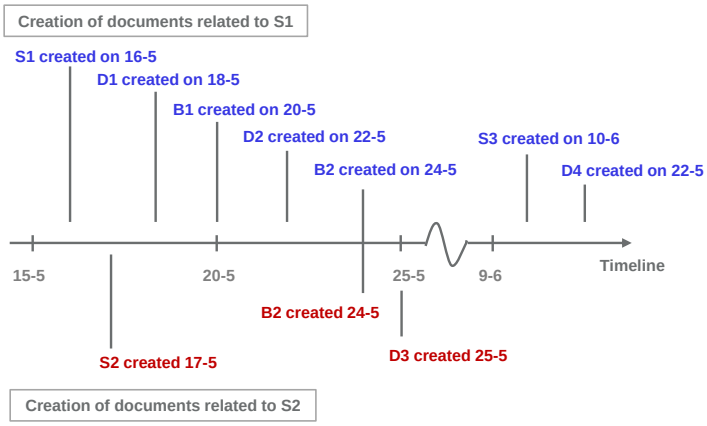

Fig. 2: A time-line regarding the creation of documents of the OTC example.

uments were created before the billing document (main flow), but once the order was reversed (B2 before D3).

The main research problem addressed in this paper is to provide (semi-)automated techniques to

(1) discover from the relational transactional data of an ERP system an accurate graphical model describing all transactions and their order, and
(2) identify main flows and unusual flows and highlight the latter ones.

Classical process mining techniques cannot be applied directly. Many previous studies have shown that an attempt to cast transactional data over objects with many-to-many relations into a single process event log and discovering a single process model describing all transactional data is bound to fail. It leads to false dependencies between events and duplicate events which obscures the main flow and hinder the detection of unusual flows [3] [4] [5] [6] [7].

1.2 Proposed Solution

We propose to approach the problem under the "conceptual lens" of *artifact-centric models* [8, 9]. An *artifact* is a data object over an information model; each artifact instances exposes *services* that allow changing its informational contents; a *life-cycle model* governs when which service of the artifact can be invoked; the invocation of a service in one artifact may trigger the invocation of another service in another artifact. Information models of different artifacts can be in one-to-many and many-to-many relations, which allows to describe behavior over complex data in terms of multiple objects interacting via service invocations. We apply the artifact-centric view to our problem as follows: each document of an ERP system can be seen as an artifact; transactions on the document are service calls on the artifacts; behavioral dependencies between transactions of documents can be seen as life-cycle behavior and dependencies of service calls. With these concepts, the transactional data of Fig. 1 can be described as the

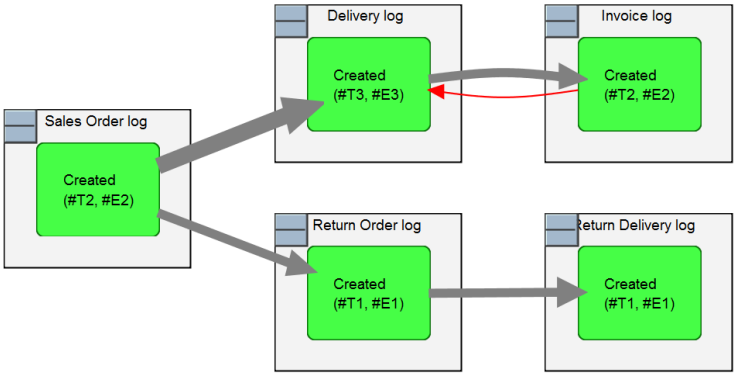

Fig. 3: Artifact-centric model of the behavior in Fig. 1

artifact-centric model of Fig. 3. The model visualizes the order in which objects are created and also highlights the unusual flow of invoice B2 being created before delivery D2.

The problem of discovering an artifact-centric process model from relational ERP data decomposes into two sub-problems:

(1) Given a relational data source, identify a set of artifact types on the database level, extract for each artifact type an event log and discover its life-cycle.
(2) Given a relational data source, a set of artifact types and their corresponding set of logs, identify interactions between the artifact types, between their instances, between their event types and between their events. As a result, obtain a complete artifact-centric process model.

Figure 4 shows the overview of our approach. The flow of our approach that addresses the first problem of discovering artifact's life-cycles is shown by the filled arcs, whereas the second problem of discovering interactions between artifacts is addressed by the flow shown by the dashed arcs. In a nutshell, (1.1) we use the data schema to discover *artifact schemas* from which (1.2) we discover *artifact types*; each artifact type describes the information model of one artifact in terms of the attributes found in the data source. Each record in the data source defines an *artifact instance*. For each artifact type, (1.3) we extract its instances and all related events; all events related to one artifact instance are grouped together into a *case* of this instance and ordered by time. The case describes how the artifact instance evolved over time. All cases together yield the *event log* of the artifact type. (1.4) We feed the event log of an artifact type to existing process discovery algorithms to obtain the life-cycle model of the artifact type. In parallel, (2.1) we use the foreign key relations in the data schema to discover interactions between artifact types and instances. (2.2) This information about interactions between artifact instances is also added to the respective cases in the extracted event logs. (2.3) We then propose two different techniques to derive from the interactions between cases interactions between the events of the different cases. (2.4) Interactions between events are then generalized to interactions between life-cycle models.

We implemented our approach and conducted two case studies. In both case studies the discovered process models were assessed as accurate graphical representations of

the source data; insights about unusual flows could be obtained significantly faster than with existing best practices. Thereby, we also learned that the steps of (1.1-1.2) of identifying artifact types and steps (2.1-2.4) are tightly related due to relations in the original relational data source. By choosing whether a relation is contained inside one artifact type or between two artifact types, one also chooses whether there is an interaction between artifacts or not. In this paper, we will show that by moving all one-to-many and many-to-many relations between artifact types in (1.1-1.2), the life-cycle models discovered in (1.4) have higher quality and the interactions discovered in (2.1-2.4) are meaningful to business users.

The remainder of this paper is structured as follows. In Section 2, we provide a detailed problem analysis using a running example, showing the limitations of classical log conversion approaches and motivating the use of an artifact-centric approach instead. Section 3 discusses related work. Section 4 illustrates our extended approach to identify artifacts and their life-cycles from a given relational data source. In Section 5, we discuss interactions between artifacts on different levels and show how to identify these interactions to obtain a complete artifact-centric model. The methodology used to conduct artifact-centric process analysis is presented in Section 6. We implemented our technique and report on two case studies in Section 7. Section 8 concludes the paper.

2 Problem Analysis

In this section, we first introduce a running example that is used throughout the paper to demonstrate the concepts used in this paper and our approach. Using the running example, we then discuss why classical log conversions and process discovery techniques fail to analyze ERP data sets. Then we introduce the artifact-centric approach and show that it is better suited to describe ERP data sets allowing for a variety of results depending on user choices.

2.1 Running Example

To illustrate the problem of process discovery from ERP data, we consider a simplified variant of the Order to Cash process supported by SAP systems and use this as our running example throughout the paper. In short, the OTC process starts with customers placing orders. Then, the organization fulfills the orders by delivering the goods and sending invoices to bill the cost and receive payments from customers. Organizations use an ERP system to store documents of sales orders, deliveries, invoices and payments that are related to the OTC process in tables similar to those shown in Figure 1. We briefly explain the process executions that have led to the data in Figure 1, focusing only on the creation of documents for the sake of brevity. First, a customer placed a sales order S1, which is created in the system on May 16th. Then a partial delivery D1 is done on May 18th, and the related invoice B1 is created two days later. On May 22th, another part of the sales order S1 is delivered according to the delivery document D2 which is invoiced with document B2 on May 24th. On May 17th, the same customer places another sales order S2, which is also invoiced within the same billing document B2. However, the delivery D3 related to the sales order S2 is executed after the billing

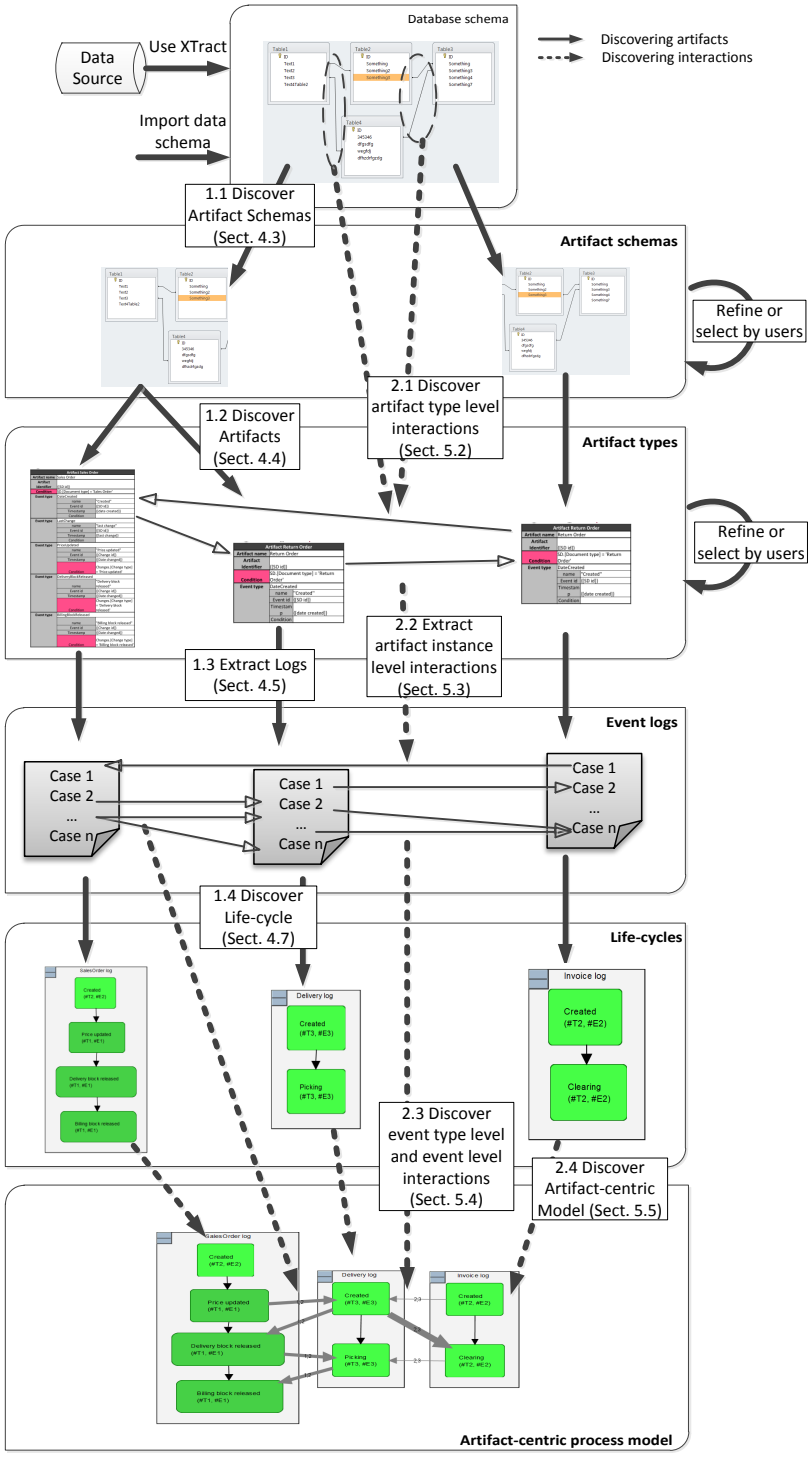

Fig. 4: An overview of our approach.

document B2 on May 25th. Days later, a return order S3 is placed for the sales order S1 and return delivery D4 is executed. A time-line of the events related to the creations is shown in Figure 2, in which a distinction is made between the creation of documents that is related to the sales order S1 (above the line in Figure 2) or to S2 (below the line in Figure 2).

The data related to these executions are stored in four tables *Sales Documents*(*SD*), *Delivery Documents*(*DD*), *Billing Documents*(*BD*), and *Document Changes*(*DC*) shown in Figure 1. The table *SD* contains the two sales order documents S1 and S2 and the return order document S3. The foreign key F1 relates the sales documents to each other. The *DD* contains the three delivery documents D1, D2 and D3 and the return delivery document D4. The delivery and return delivery documents are related to the sales documents via foreign key F2. The two invoices B1 and B2 are stored in the table *BD* have relations with the delivery documents via foreign key F3. Any changes related to the documents are stored in the table *Document Changes*.

2.2 Classical Log Conversion and Process Discovery

Process mining is a set of techniques to "*discover, monitor and improve real processes (i.e., not assumed processes) by extracting knowledge from event logs*" [10]. In this paper, we focus on process discovery, which aims to discover a process model from a given event log.

In general, an event log comprises a list of traces of which each *trace* contains all events that occurred in a case, i.e., an execution of the process. Each event may be characterized by various *attributes*, e.g., a timestamp, correspond to an activity, is executed by a particular person. Therefore, to be able to apply process discovery techniques, relational data sources have to be converted into an event log.

Classical log conversion and extraction approaches [3–7] tend to extract an event log from a relational data source based only on one notion of a case. These approaches first (try to) identify or define one notion of a case. After specifying or selecting the event types related to the defined case notion, the approaches collect the events found in the data source that are associated with the defined cases. The extracted events are finally sorted by cases and time and written into one event log. These approaches only extract one log for one process definition at a time, while assuming the process is isolated and has no interaction with other processes or its system environment. For example, if we consider the sales orders in the OTC example as our cases of the OTC process, we can obtain the event log of Figure 5, by relating each creation event to one of the two sales order cases. For example, Figure 5 shows that the sales order S1 trace has seven events, each event has four attributes. While this method is straight forward, it leads to two special problems arising from one-to-many relationships between the source tables.

Data Divergence The *data divergence* problem is defined as the situation when a case is related to multiple events of the same event type. Figure 5 shows that the case sales order S1 has two *Delivery Created* events D1 and D2 and two *Invoice Created* events B1 and B2. If we draw a simple causality net by only using the trace S1, we obtain the model shown in Figure 6 (left). Business users immediately notice the edge from

Event Log

Trace : Sales Order S1			
Event Id	**Event type**	**Event timestamp**	**Resource**
S1	Order Created	16-5-2020	Dirk
D1	Delivery Created	18-5-2020	Dirk
B1	Invoice Created	20-5-2020	Dennis
D2	Delivery Created	22-5-2020	Marijn
B2	Invoice Created	24-5-2020	Marijn
S3	Return order Created	10-6-2020	Xixi
D4	Return Delivery created	12-6-2020	Xixi

Trace : Sales Order S2			
Event Id	**Event type**	**Event timestamp**	**Resource**
S2	Order Created	17-5-2020	Dirk
B2	Invoice Created	24-5-2020	Dennis
D3	Delivery Created	25-5-2020	Dirk

Fig. 5: An conceptual event log of the OTC example

Invoice to *Delivery* and find this edge strange as they think the edge indicates that there are invoices created before the related deliveries. However, this edge actually means that there is an invoice B1 created before a delivery D2, both of which are related to the sales order S1 but not related to each other. The *complexity and ambiguity of the process model* increase when more deliveries and invoices are linked to the case, as the divergence problem also introduces self-loops. Now, if we include the trace S2, a similar model shown in Figure 6 (right) is discovered, in which the same abnormal edge from *Invoice Created* to *Delivery Created* appears. However, this time there really is an invoice B2 created before its related delivery document D3, which is an outlier and might indicate risks or faulty configurations in the process.

The aim of conducting process analysis by business users is to produce rather simple process models that can be used to communicate with stakeholders and to identify exactly the abnormal process executions that happened in reality. As the running example shows, this aim is disturbed by the divergence problem. Solving data divergence is therefore one of the goals of this paper.

Data Convergence The problem of *data convergence* is defined as the situation when one event is related to multiple cases. For example, when considering the sales orders as the notion of a case and the creation of invoices as events, the *Invoice Created* event of the invoice B2, which is related to two different sales orders S1 and S2, is extracted twice, as illustrated by the event log in Figure 5. Traditional process mining techniques consider the event *Invoice Created* B2 as two different events. Together with the creation of invoice B1, we obtain three *Invoice Created* events as shown in Figure 6 (right), whereas there are actually only two invoices B1 and B2. Thus, the data convergence problem leads to extracting duplicate events and biased statistics.

Choosing different notions of a case for the process definition is proposed in [5] [6] [7] as a solution to the divergence and convergence problem in traditional log extraction approaches. However, while this might avoid some issues of converge and divergence, it cannot solve these problems completely. Taking the OTC example, and choosing the invoices as the case definition, the many-to-many relation between the invoices and sales orders yields an event log suffering from divergence and convergence. Choosing the deliveries as case definition solves the divergence problem, but worsens the convergence problem. It is also very difficult to define or to retrieve an optimal definition of a case from all possible case definitions found in relational data.

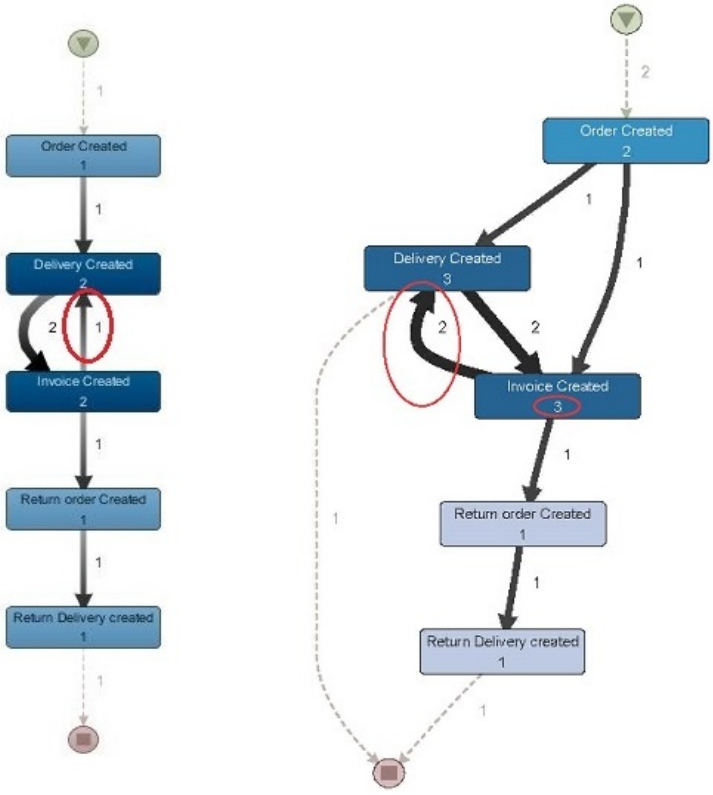

Fig. 6: Left: A causal graph of Sales Order S1; Right: A causal graph of the OTC example.

2.3 Artifact-Centric Approach

The data divergence and convergence problems discussed in the previous section show that the classical log conversion and mining approaches are unable to handle one-to-any and many-to-many relations between cases and their events, which are frequently observed in complex data models such as the ones employed by ERP systems. Such a complex data model contains several logically defined objects that are relevant for

the business process execution (e.g., the objects such as the sales orders, deliveries and invoices of the OTC example); each object has attributes and is related to other objects (i.e. has interactions with each other). During process execution, instances of these objects are created and related to other instances of other objects. Each of these objects (and each instance of an object) has a real-life interpretation. In order to deal with such complex processes, the *artifact-centric* approach has been proposed, which describes a process in terms of how (all of) its objects evolve through an execution, instead of a single monolithic process model [8, 9, 11].

An *artifact* is a conceptually relevant object (with a real-life interpretation) that observes a life-cycle of updates from instantiation to some goal state. We use the term *artifact type* to refer to the formal definition (i.e. the type) of an artifact and use the term *artifact instance* when we refer to a particular instance of an artifact type. For example, the notion of sales order documents can be considered as an business object, thus, an artifact. The formally defined artifact type of this artifact is *Sales Order* which contains an event type *Created*. The sales order S1 in the running example is an artifact instance that belongs to the artifact type *Sales Orders*.

An *artifact-centric model* encapsulates all the artifacts that are engaged in such a dynamic business process and visualizes the general life-cycle of each artifact. Actions of the process move an artifact instance from one state to another until some goal state is reached. Artifacts that are related to each other may influence each other, i.e., an action on one artifact instance may trigger/lead to an action of a related artifact instance. In other words, artifacts interact with each other.

Similar to finding an optimal notion of cases in the classical log conversion problem, finding a set of optimal artifacts from a given data source is difficult and depends on the goals of process analysis projects. Defining the scope of each artifact not only influences the life-cycle of artifacts but also the interactions between the artifacts. In addition, there is a trade-off between the number of artifacts and the amount of data per artifact, for example, in terms of the number of tables related to an artifact, which again affect the complexity of an artifact. This trade-off is depicted by Figure 7.

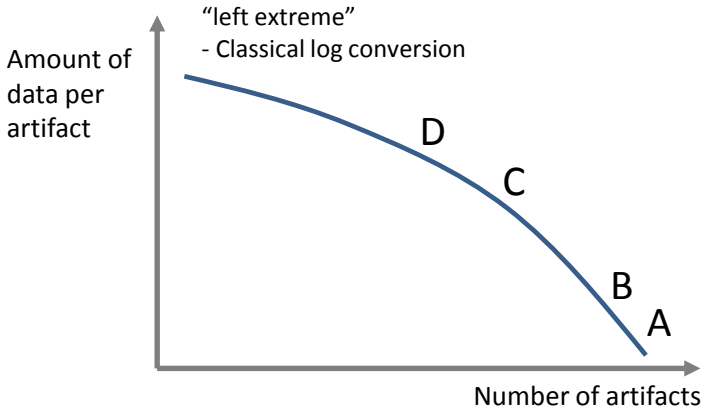

Fig. 7: The trade-off of defining artifacts

In this trade-off, the classical log extract represents the “left extreme” option which minimizes the number of artifact (to one) and maximizes the amount of data per artifact (to include all tables), thus, having only a single artifact containing all event data. The “right extreme” option minimizes the amount of data per artifact (resulting in simple artifacts) while maximizing the number of artifacts. Figures 29 and 6, obtained using classical conversion approach, already show examples of the “left extreme” option. Sections 2.3 and 2.3 discuss two examples of defining artifact types using the “right extreme” options A and B, and Sections 2.3 and 2.3 show two examples C and D in between. The artifact-centric models shown in this section are only to illustrate the different ways of constructing artifacts and its subsequent effect on the interactions between them, the exact meaning of the model is explained in Sections 4 and 5.

Example B - Tables as Artifact Types First we discuss a rather direct mapping from tables to artifacts: each table defines one artifact, each datetime column defines an activity (or step) in the artifact. Figure 8 shows an artifact-centric model of the OTC example consisting of three artifacts *SD*, *DD*, and *BD* (named after their originating tables and denoted by large grey rectangles), each of which consists of one event type *Created* (denoted by green rectangles within grey rectangles). Also, the self-loop on *Created* in artifact *SD* shows that the activity Created has been executed twice in an instance of that artifact. Finally, there is an interaction (denoted by arcs between green rectangles) from the creation of artifact *SD* to the creation of artifacts *DD* which leads to the creation of *BD*.

The model was obtained by mapping each table in Figure 1 to one artifact type and a datetime column to one event type. For example, table *SD* to artifact *SD*, and the datetime column *Date created* to activity *Created*.

One of the limitation of mapping one table to one artifact type and one column to one event type is that one table may hold data of conceptually different artifacts, e.g. sales orders and return orders are both stored in table SD. By considering these conceptually different artifacts as one artifacts, we lost the ability to distinguish the differences in their life-cycles and their interactions towards other artifacts. For example, we can not clearly see the sales orders S1 and S2 have interactions with delivery documents and return order documents whereas the return order document S3 stored in the same table only have relation to return delivery but no deliveries.

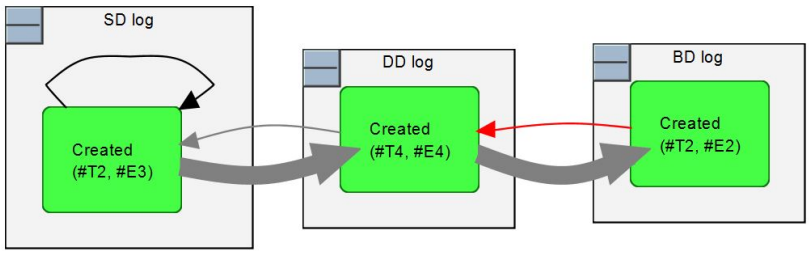

Fig. 8: Example B which maps each table as an artifact type.

Example A - Document Types as Artifacts A more fine-grained artifact-centric model of the OTC example is shown in Figure 3. This model was obtained by mapping a subset of a table to an artifact. For example, the table *SD* is mapped to two artifacts: the subset related to the document type "sales order" is mapped to the artifact *Sales Order*, whereas the other subset related to the document type "return order" constitutes the artifact *Return Order*.

Mapping subsets of a table to different artifacts, we are able to distinguish the difference between sales orders and return orders, and between deliveries and return deliveries. Moreover, and arguably more importantly, also the interactions between artifacts get refined. For example, the model shows that according to the current data set, the return deliveries have no relation with invoices whereas the deliveries do have invoices. In addition, considering the one-to-many relations as interactions, the artifact-centric model is able to show the true unusal flow (denoted by red arcs), i.e. the creation of an invoice happened before the creation of its related delivery, in comparison to the models shown in Figure 6 obtained using the classical log conversion.

Example C - Only One-to-One within Artifacts It is also possible to consider a set of tables to be related to an artifact. For example, one can consider the sales orders and their return orders and return deliveries as one artifact. Since there is only one-to-one relations between sales orders and return orders and return deliveries, the obtained lifecycle (process model) of this artifact do not have the data convergence and divergence issues.

Figure 9 shows the artifact-centric model consisting of the artifact *Sales Order* and the two artifacts *Delivery* and *Invoice*. Note that both relations within the artifacts as well as the interactions between the artifacts are simple to interpret.

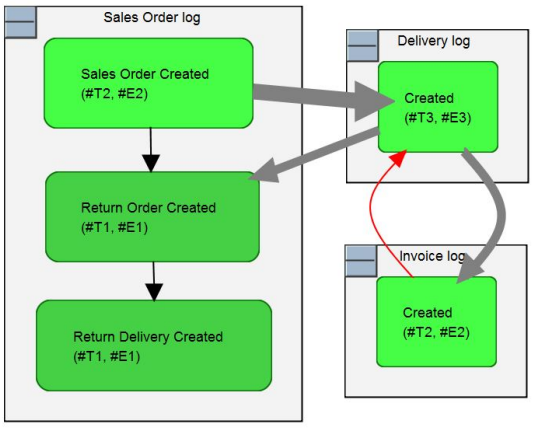

Fig. 9: Example C consists of artifacts within which only one-to-one references are allowed

Example D - One-to-Many within Artifacts Defining more complex artifacts and including non-one-to-one relation within artifacts is also an option (and is supported

by our approach). However, such artifacts increase the complexity of their life-cycles and the interactions between them, making the derived artifact-centric model more difficult to interpret. Figure 10 shows two artifacts: one is the *Sales Order* artifact which includes the sales orders, deliveries, return orders, and return deliveries; the other is the *Invoice* artifact. Since one sales order can be related to many deliveries, we already observed the data divergence within artifact *Sales Order*, i.e. the self-loop around the event type *Delivery Created*. It also increased the complexity of the interactions between *Sales Order* and *Invoice*. While the model clearly describes that *Sales Order Created* happened before the events *Created* of *Invoice* and the events *Created* of *Invoice* before the *Return Order Created*, the specific inter-leavings between the events *Delivery Created* and the events *Created* of *Invoice* are difficult to interpret, but relevant to a business user.

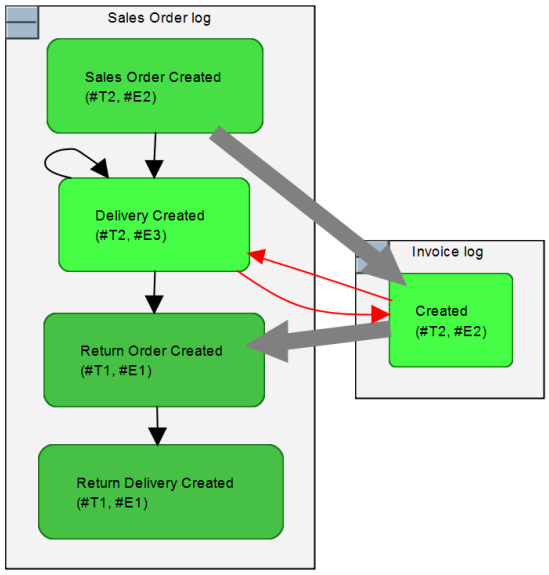

Fig. 10: Example D consists of artifacts within which one-to-many relations are allowed

To summarize, we have discussed various options to create artifact-centric models using the OTC example. In addition, the discussion shows that artifact-centric approaches are more general than and actually include the classical log conversion approaches (by mapping all data to one artifact). While artifact-centric approaches provide a more dynamic way to analyze a data source with complex data structures, discovering artifacts and interactions between them is crucial for conducting analysis.

3 Related Work

We discuss existing work along the main problems addresses in this paper: (1) discovering conceptual entities and their relations from a relational data structure, (2) extracting event logs from relational data structures, (3) discovering models or specifications of

a single entity/process from an event log, and (4) discovering/analyzing relations and interactions between multiple objects and processes.

Entity discovery. The relational schema used in a database may differ significantly from the conceptual entities which it represents, mostly to improve system performance. Various existing works solve different steps along the way. After discovering the actual relational schema from the data source [12–14], an (extended) ER model can be retrieved that turns foreign keys between tables into proper relations between entities [15–17]. The artifact discovery problem faced in this paper (Sect. 4) differs from this problem as one artifact type may comprise *multiple entities* as long as they are considered to be following a joint life-cycle, that is, multiple entities may grouped into the same artifact type, such that *convergence* and *divergence* (see Sect. 2.2) do not arise. This problem has been partly addressed in [18] through schema *schema summarization* techniques [19] though convergence and divergence may still arise.

It is also possible to discover entities and artifact types from a *raw event stream* (instead of a relational structure); the prerequisite is that each event carries enough attributes and identifiers. The approach in [20] first reconstructs a simple relational schema from all events and their attributes, two related entities can be grouped into the same artifact if one entity is always created before the other (according to the event stream); this extraction dismisses interactions between different artifacts which is crucial to our approach (step 2.1 in Fig. 4. This work presents a first complete solution to discovering entities, artifacts, and their interactions from relational data in Sect. 4 (steps 1.1 and 1.2 in Figure 4), and Sect. 5.1 (step 2.1).

Log Extraction. Existing work on extracting event logs from relational data sources (step 1.3 in Figure 4) mainly focus on identifying a monolithic process definition and extracting one event log where each trace describes the (isolated) execution of one process instance. Manually approaches to extracting data from relational databases of SAP systems particularly failed to separate events related to various processes; analyzing what was part of the process was hard and time consuming [21, 22]. In the generic log extraction approach of [5], the user defines a mapping from from tables and columns to log concepts such as traces, events, and attributes (assuming the existence of a single case identifier to which all events can be related); various works exist to improve finding optimal case identifiers and relations between the identifiers and events [6, 7, 23]. If the event data is structured along multiple case identifiers as in ERP systems, all these approaches suffers from data convergence and divergence (Sect. 2.2). In this work, we identify multiple artifact types (each having their own case identifiers) and separate events into artifact types such that convergence and divergence do not arise; having identified proper case identifiers and related events, we then reuse the approach of [5] to extract an event log for each artifact type. No existing work extracts attributes that describe the interaction between different artifact instances; we present a first solution in Sect. 5 (step 2.2 of Fig. 4).

Model discovery. Much research has been conducted on the problem of discovering a (single) process model from other information artifacts. *Process mining* [1] takes as input an event log where each trace describes the execution of one process instance. An event in the log is a *high-level* event corresponding to a complex user action or system action, potentially involving dozens or thousands of method calls, service invocations,

and data updates. The log describes behavior that *actually* happened allowing to discover unusual and exceptional flows not intended by the original process design. Some well known process discovery techniques are Alpha algorithm [24], (Flexible) Heuristic miner [25], Genetic process mining [26], ILP mining [27], Fuzzy mining [28], and Inductive Mining [29] [30]. De Weerdt et al. [31] compared various discovery algorithms using real-life event logs. Existing discovery techniques mainly focus on a single process discovery and assume the model operates in an isolated environment. We will reuse existing process discovery techniques when discovering artifact life-cycle models (step 1.4) and artifact interactions (step 2.3 of Fig. 4).

One can also use low-level event logs where one event corresponds to an atomic operation (method invocation, data read/write, message exchange). Low-level event logs are usually considered when discovering models and specifications of particular software artifacts (the object-oriented source code of a module, the GUI, etc.). Various techniques are available to discover formal behavioral specifications such as automata [32, 33], scenario-based specifications [34], or object-usage models [35] from low-level event logs; see [36, 37] for overviews. Like artifacts, object-usage models describe how an object is being used in a context. These techniques rely on the assumption of sequential execution (on a single machine) and strict patterns (following code execution), while our problem features a high degree of concurrency and user-driven behavior. Concurrent use and user influence is considered in [38] being essentially a variant of process mining discussed above.

Other works use event data generated by users in the application interface to discover models of how a user operates an application. These events can be used to analyze styles of process modeling [39] or problem solving strategies in program development environments [40]; these works cannot analyze events beyond the user interface which is the scope of this paper. In [41] it is shown how to generate application interface test models by generating user interface on a web interface; this work synthesises the user behavior whereas we analyze actual user behavior.

Interactions and deviations. The notion of *artifacts* [8, 9] where a (complex) process emerges from the interplay of multiple related objects has proven to be a useful conceptual lens to describe behavioral data of ERP systems. The feasibility of the artifact idea in process mining was demonstrated in [42, 43] by checking the conformance of a given artifact-centric model to event data with multiple case identifiers. In [18, 44], the XTract approach was introduced which allows for fully automatic discovery of an artifact-centric model (multiple artifacts and their life-cycles) from a given relational data source. It is also possible to discover artifact-centric process models from event streams where events contain enough attributes to discover entities and relations [20]; this work also shows how to produce life-cycle models in GSM notation [11], a declarative language for describing artifact-centric processes. Both approaches are limited to identifying individual artifacts, extracting logs, and discovering life-cycles, but cannot identify interactions between artifacts and may suffer from convergence and divergence. In this paper, we extend this approach to avoid the problems and also discover interactions between artifacts.

With respect to the second problem of discovering interactions between artifacts, much less literature has been found. Petermann et al. [45] proposed to represent rela-

tional data as graphs in which nodes are objects or instances and edges are relations, which is comparable to (2.1) in Figure 4. However, the scope of their approach are only limited to instances and direct relations between objects, while neglecting the dynamic life-cycles of instances and the interrelations between them. Conforti et al. [46] proposed another way to address data divergence and convergence by contextualizing one-to-many relations as subprocesses of a BPMN model instead of interactions between artifacts; this approach unable to handle many-to-many relations as encountered in this paper.

Also object-usage models and scenario-based specifications have been used to study object interactions. In [47] it is shown how to discover from source code how an (object-oriented) object is being used in a caller context; such models can also be discovered from low-level execution traces [35]. Also scenario-based specifications discovered from low-level event logs [34] describe interactions between multiple objects. However, all these works either focus on a single object or do not distinguish multiple instances of several interacting objects in many-to-many relations, i.e., two orders being processed in three deliveries, which is a crucial property of our problem. Using event logs from two different versions of an object, it is possible to reveal detect changes in object usage [48]. In this paper, we want to detect deviations of usage of a single version of an object to identify outlier behavior.

To summarize, our approach addresses a more general problem than all preceding approaches: (1) discover multiple artifacts (comprising multiple entities) that are in many-to-many relations to each other such that data divergence and convergence do not arise, and (2) discover interactions between artifacts and identify outliers in these interactions. Sections 4 and 5 address the first and second problem, respectively, and explain our approach more in detail. The methodology of using our approach to conduct artifact-centric process mining analyses is discussed in Section 6.

4 Artifact-Centric Log Extraction and Life-cycle Discovery

The first step in our approach is to identify artifact types from a given relational data source. The artifact types typically describe high-level objects with a real-life interpretation. However, the relational schema used in the data source may differ significantly from the conceptual model it represents, usually due to performance optimizations. We first discuss this problem and then our approach to overcome it.

4.1 Relational Schemas vs. High-Level Models

One can describe the difference between conceptual high-level models and relational schemata in terms of four basic operations. (1) *Horizontal partitioning* specializes a general entity (or artifact) into multiple different tables depending on their kind. For example, “Documents” are distinguished into “Sales Documents” and “Delivery Documents” with different tables, see Fig. 1. (2) *Vertical partitioning* distributes properties of one entity into multiple different tables. For example, the “Changes” to a “Delivery Document” are not stored in the “Delivery Documents” table, but in a separate “Document Changes” table. (3) *Horizontal Anti-Partitioning* generalizes data from multiple

entities into one table. For example, changes of different document types are all stored in the same "Document Changes" table rather than in separate tables. (4) *Vertical Anti-Partitioning* aggregates attributes of multiple entities into the same table. For example, "Sales Documents" aggregates attributes for "Sales Order" and "Return Order" (even though "Reference id" is only required by "Return Order"). The examples also show that one table may be the result of multiple such operations.

Artifact identification has to undo these operations. The problem is similar to recovering a classical entity-relationship (ER) model from a relational data source; see Sect. 3. The artifact discovery problem solved here differs from this problem as one artifact type may comprise *multiple entities* as long as they are considered to be following a joint life-cycle, see for example Sect. 2.3 combining entities "sales orders", "return orders", and "return deliveries" into one artifact. The XTract approach [18] uses schema summarization techniques [19] to cluster tables in the data source based on their "informational distance"; This approach can undo some cases of horizontal partitioning and some cases of vertical partitioning by grouping multiple related tables into the same cluster.

In the following, we present a more general, semi-automatic approach for artifact identification. We want to identify artifact types from a relational data source and then extract an event log describing the artifact's life-cycle. Therefore, each artifact type shall comprise all attributes, including time stamps, related to a particular high-level business object.

Due to vertical partitioning, an artifact's attributes may be distributed over many tables. We undo vertical partitioning by grouping all tables related to an artifact in order to collect all its attributes. This first step, described in Sect. 4.3, yields an *artifact schema* that potentially contains information of multiple different artifacts that were all stored in the same tables due to horizontal anti-partitioning. The artifacts in one schema are all of a similar form. However, because of vertical anti-partitioning, there may be tables containing information of artifacts of very dissimilar form, such as table "Changes" in Fig. 1. To overcome this side effect of vertical anti-partitioning, the same table may be part of different artifact schemas; this refinement of artifact schemas may require user interaction.

Next, we refine an artifact schema into individual artifact types by letting the user specify a discriminating predicate for each artifact thus undoing horizontal anti-partitioning. This step also reverts vertical anti-partitioning by selecting from the artifact schema only those attributes that are relevant for an artifact type as shown in Sect. 4.4. Each resulting artifact type allows to extract an event log describing the life-cycle of this artifact; this step is discussed in Sect. 4.5. Reversing horizontal partitioning (i.e., dealing with specialization) is discussed in Sect. 4.6. Finally, we discuss how existing process discovery techniques can be used to discover a suitable life-cycle model for each artifact.

4.2 Preliminaries - Relational Data

Before going into details, we briefly recall some standard relational concepts [49].

Definition 1 (Tables, Columns). $\mathbb{T} = \{T_1, \cdots, T_n\}$ *is a set of tables of a data source, where each* table $T_i = \langle \mathbb{C}, \mathbb{C}_p \rangle$ *is a tuple of its* columns $\mathbb{C}$ *and its* primary keys $\mathbb{C}_p$.

In our OTC example, we have four tables, each of which has one column as primary key, i.e. $\mathbb{T} = \{$SD, DD , BD, Changes$\}$ and e.g. table $SD = \langle \{$SD id, Date Created, Reference id, Document Type, Value, Last change $\}, \{$SD id$\}\rangle$.

Definition 2 (References). *$F = \langle T_p, \mathbb{C}_p, T_c, \mathbb{C}_c, F_{condition}\rangle$ is a reference if and only if*

- *T_p is the* parent table *,*
- *$\mathbb{C}_p$ is an ordered subset of columns denoting the primary key of the parent table,*
- *T_c is the* child table*,*
- *$\mathbb{C}_c$ is an ordered subset of columns denoting the foreign key, and*
- *$F_{condition}$ is the extra condition for the reference (which can be appended in the FROM part or the WHERE part of an SQL query).*

The condition $F_{condition}$ reflects the as-is situation in various ERP systems such as SAP where $\mathbb{C}_c$ only is a proper reference to an entry in T_p if that entry has a particular value in particular column of T_p. For example, the foreign key F_4 can be defined by three references, and one of these references is $\langle$ [SD], $\{$SD id$\}$, [Changes], $\{$Reference Id$\}$, " [Changes].[Table name] = "SD " "$\rangle$. The condition $F_{condition}$ could be empty indicating $F_{condition}$ is true.

Definition 3 (Data schemas). *$\mathfrak{S} = \langle \mathbb{T}, \mathbb{F}, \mathbb{D}, column\_domain\rangle$ is a data schema with:*

- *$\mathbb{T}$ is a set of the tables with the primary keys of each table filled in;*
- *$\mathbb{F}$ is a set of references between the tables;*
- *$\mathbb{D}$ is a set of domains; and*
- *column_domain that assigns each column a domain.*

The data schema of a relational data source describes the relational structure of the data source. Since our approach requires a data schema as input, the data schema can be either discovered using the original XTract approach or imported.

4.3 Artifact Schema Identification

Our first step is to identify artifact schemas, where one artifact schema contains all attributes related to all artifacts of a similar form. Formally, an artifact schema is a collection of related tables; a distinguished main table holds the identifier.

Definition 4 (Artifact-schemas). *$\mathfrak{S}_A = \langle \mathbb{T}_A, \mathbb{F}_A, \mathbb{D}_A, column\_domain, T_m\rangle$ is an artifact-schema if and only if $\mathfrak{S}_A$ a subset of the schema $\mathfrak{S} = \langle \mathbb{T}, \mathbb{F}, \mathbb{D}, column\_domain\rangle$, i.e.,*

- *$\mathbb{T}_A \subseteq \mathbb{T}$ is a subset of tables;*
- *$\mathbb{F}_A \subseteq \mathbb{F}$ is a subset of reference;*
- *$\mathbb{D}_A \subseteq \mathbb{D}$ is a subset of domains;*
- *column_domain is the assignment function of the schema; and*
- *$T_m \in \mathbb{T}_A$ is the main table in which the trace identifiers can be found.*

While the existence of a unique main table T_m cannot be formally guaranteed for all relational schemas, previous studies and our own results suggest that such a table can always be found in practice [4, 6, 18, 22, 50].

The starting point for finding artifact schemas in the relational data source is its schema $\mathfrak{S}$. We can assume that this schema to be known either from existing documentation or through schema summarization techniques used by [18].

From this graph, we can remove the references which are not one-to-one, thus resulting in a graph only connected by one-to-one references. Each of the resulting connected sub-graphs can be considered as valid artifact schemas as it only contains tables which are linked by one-to-one references. The main table T_m can be selected as a table which has no parent in the set $\mathbb{T}_A$ of the selected tables. The set $\mathbb{D}_A$ of domains is the union set of all domains of columns of the tables in $\mathbb{T}_A$. We can obtain an artifact schema $\mathfrak{S}_A$ and add it to the set $\mathbb{S}$ to be returned.

Algorithm *ComputeArtifactSchemas*($\mathfrak{S}$)
1. Let a graph $G = (\mathbb{T}_G, \mathbb{F}_G) \leftarrow (\mathfrak{S}.\mathbb{T}, \mathfrak{S}.\mathbb{F})$
2. **for** $F \in \mathbb{F}_G$
3. **do if** (F is not one-to-one)
4. **then** remove F from $\mathbb{F}_G$
5. **for** each connected sub graph $g = (\mathbb{T}_g, \mathbb{F}_g) \subseteq G$
6. **do** $\mathbb{T}_A \leftarrow \mathbb{T}_g$, $\mathbb{F}_A \leftarrow \mathbb{F}_g$,
7. Select a table $T_m \in \mathbb{T}_A$ which has no parent table in $\mathbb{T}_A$
8. $\mathbb{D}_A \leftarrow$ the union set of domains of columns of the tables in $\mathbb{T}_A$
9. $\mathfrak{S}_A \leftarrow \langle \mathbb{T}_A, \mathbb{F}_A, \mathbb{D}_A, \mathfrak{S}.\mathit{column\_domain}, T_m \rangle$ ($*$ create a new artifact schema $*$)
10. Add artifact schema $\mathfrak{S}_A$ to $\mathbb{S}$
11. **return** $\mathbb{S}$

The algorithm *ComputeArtifactSchemas* presented is a simple brute-force way of partitioning the tables into artifact schemas containing only one-to-one relations. This is to prevent a potential de-normalization during log extraction which could result in duplication of records and extracted events (see Sect. 2.2). Note that the one-to-many relations are not dropped or removed. Rather, they describe relations *between* different artifact schemas and will be used in Sect. 5 when discovering interactions between artifacts; see also the overview of our approach in Figure 4.

The initial partitioning returned by *ComputeArtifactSchemas* is a "safe" partitioning that prevents data divergence and convergence problems occurring within artifacts, as discussed in Section 2.2. However, these safe artifact schemas might not yet match the intended conceptual schemas: one might obtain trivial artifact schemas containing only one table, or incomplete artifact schemas missing information contained in a table related to another artifact. We have shown in Sect. 2.3 that different artifacts can be conceptualized from the same relational data source depending on how tables are grouped. Thus, as a second step, we allow users to add or remove tables from a schema

in order to obtain the intended artifacts. This way, also one-to-many relations may be included in an artifact schema at the potential cost of data convergence and divergence. Moreover, as one table may contain information of artifacts stored in different schemas (vertical anti-partitioning), we explicitly allow artifact schemas to overlap in tables.

The manual refinement of artifact schemas requires domain knowledge, which is typically available for standard ERP systems by Oracle or SAP. In case no domain knowledge is available, earlier works [18, 44] could be used to automatically identify artifact schemas based on their informational contents. However, the resulting artifact schemas may include one-to-many relations and thus induce data convergence and divergence. Again, a subsequent manual refinement is required to obtain the desired artifact schemas. We illustrate the difference between the original XTract approach [44] and our artifact schema identification using the OTC example. The XTract approach returns the three artifact schemas shown in the left table of Figure 11 when we set the number of artifacts to be 3. Our approach first return three artifact schemas SD, BD, and DD as shown by the black tables in the right table of Figure 11. Since only one invoice has a change, the document changes table is assigned to the BD artifact schema. The SD artifact schema returned only contains the SD table, similar for the DD artifact schema. Now if users desire to include changes for the SD artifact, they can add the changes table to the SD artifact schema.

XTract: Artifact Schemas (k = 3)		
Name	**Main table**	**Tables**
BD	BD	BD
Changes	Changes	Changes
Changes	?	SD, DD

Our Approach: Artifact schemas		
Name	**Main table**	**Tables**
BD	BD	BD, Changes
SD	SD	SD (, Changes)
DD	DD	DD

Fig. 11: Comparing the artifact schemas obtained using the XTract approach and our approach with respect to tables $\mathbb{T}$ and the main table T_m

4.4 Artifact Identification

The tables of an artifact schema $\mathfrak{S}_A$ may contain information about multiple similar artifact types, due to horizontal anti-partitioning. Next, we refine an artifact schema into its artifact types by specifying discriminating predicates. Also, due to vertical anti-partitioning, the artifact schema may contain attributes that are not relevant for each of its artifact type. Thus, we project the artifact schema onto only those attributes (identifiers, time stamps, etc.) that belong the artifact type.

Definitions Formally, we center the definition of an artifact type around the events describing its life-cycle. Intuitively, each time-stamped value in the data source describes an event, the attribute (or column) containing that value is classified as an *event type*. That is, an artifact type is a collection of columns containing time-stamp values, and an identifier. All other columns in the tables of an artifact are considered to be attributes of the various event types where they are accessible for subsequent process mining analysis. The formal definitions read as follows.

Definition 5 (Event types). $E_i = \langle E_{name}, \mathbb{C}_{Eid}, C_{time}, \mathbb{C}_{Eattrs}, E_{condition} \rangle \in \mathbb{E}$ *is an event type if and only if:*

- E_{name} *is the name of the event type;*
- $\mathbb{C}_{Eid}$ *is a set of columns defining the event identifier;*
- C_{time} *is the column indicating the ordering (or the timestamps) of events of this event type;*
- $\mathbb{C}_{Eattrs}$ *is a set of columns denoting the attributes of the event type; and*
- $E_{condition}$ *is a condition (which can be appended in the FROM part or the WHERE part of an SQL query) to distinguish various event types stored in the same column* C_{time} *of the data source.*

Definition 6 (Artifact types). $A = \langle A_{name}, \mathbb{C}_{Aid}, \mathbb{E}, \mathbb{C}_{attrs}, \mathbb{I}, \mathfrak{S}_A, A_{condition} \rangle$ *is an artifact if and only if:*

- A_{name} *is and artifact name;*
- $\mathbb{C}_{Aid}$ *is a set of columns denoting the case identifier of the artifact;*
- $\mathbb{E}$ *is a set of event types;*
- $\mathbb{C}_{attrs}$ *is a set of columns denoting the case attributes,*
- $\mathbb{I}$ *is a set of interactions between this artifact A and other artifacts (which remains as an empty set in this section);*
- $\mathfrak{S}_A$ *is the corresponding artifact schema; and*
- $A_{condition}$ *is the an artifact condition which is an extra condition (which can be appended in the FROM part or the WHERE part of an SQL query) that is used to distinguish various artifacts having the same main table* T_m *(or having the same artifact schema).*

We show a concrete example of the artifact definition. Consider the artifact schema $\mathfrak{S}_A = \langle \mathbb{T}_A, \mathbb{F}_A, \mathbb{D}_A, column\_domain, T_m \rangle$, where the tables $\mathbb{T}_A = \{SD\}$ and the references $\mathbb{F}_A = \{\langle SD, \{\text{SD id}\}, SD, \{\text{Reference id}\}\rangle\}$, we would like to identify two artifacts, *Sales Order* and *Return Order*. Both artifacts may have the same artifact schema, but they could have different events, attributes, and interactions. For example, the artifact Sales Order $A_{SalesOrder} = \langle A_{name}, \mathbb{C}_{Aid}, \mathbb{E}, \mathbb{C}_{attrs}, \mathbb{I}, \mathfrak{S}_A, A_{condition} \rangle$ could have the structure shown in Table 1, in which each component of the artifact is given a value.

Discovery Algorithms To be able to semi-automatically discover artifacts from an artifact schema and a column (or columns) indicating the artifact, we define two functions. The first function constructs a single artifact, whereas the second function constructs multiple artifacts by calling the first function multiple times.

The first *createArtifact*($\mathfrak{S}_A$, A_{name}, $A_{condition}$) function takes an artifact schema $\mathfrak{S}_A$, an artifact name A_{name}, and an artifact condition $A_{condition}$ as input and returns one artifact. For this function, we assume that condition $A_{condition}$ is given or provided by a user with insights into the data model of the system. The case identifier $\mathbb{C}_{Aid}$ of the artifact is defined by the primary key of the main table of the inpu artifact schema $\mathfrak{S}_A$, each time-stamped column C_{time} in a table $T \in \mathbb{T}_A$ of the artifact schema defines an event type $E \in \mathbb{E}$, every other non-time stamped column in T defines an attribute of event type E. Every non-time stamped column that cannot be related to one specific event

Table 1: An example of the Sales Order artifact

Artifact's component	Value
A_{name}	Sales Order
$\mathbb{C}_{Aid}$	{ SD id }
$E_1 \in \mathbb{E}$	$\langle E_{name} =$ date created, $\mathbb{C}_{Eid} = \{$SD id$\}$, $C_{time} =$ [date created], $\mathbb{C}_{Eattrs} = \{\}$, $E_{condition} = \varnothing\rangle$
$E_2 \in \mathbb{E}$	$\langle E_{name} =$ last change, $\mathbb{C}_{Eid} = \{$SD id$\}$, $C_{time} =$ [latest change], $\mathbb{C}_{Eattrs} = \{\}$, $E_{condition} = \varnothing\rangle$
$\mathbb{C}_{attrs}$	{ [Document type], [Value]}
$\mathbb{I}$	$\varnothing$
$A_{condition}$	T_m.[Document type] = 'Sales Order'

type defines a case attribute. For instance, given (1) the aforementioned artifact schema $\mathfrak{S}_A$ whose $\mathbb{T}_A = \{SD\}$ and $T_m = SD$, (2) artifact name A_{name} as *Sales Order* and (3) artifact condition $A_{condition}$ as $[$*Document type*$]$ = "*Sales Order*", the primary key *SD id* of table SD is then set as the case identifier of artifact *Sales Order*. The two time-stamped columns *Date created* and *Last change* are considered as the C_{time} of two event types with names as *date created* and *last change*, respectively. The event type identifiers of these two event types are the same, the primary key *SD id* of the table SD because both time-stamped columns are in the same table SD. The three columns left, *Reference id*, *Document type* and *Value*, can not be assigned to a specific event type and thus define three case attributes. The discovered artifact is shown in Table 1; see [18] [44] for details.

Besides letting the user specify condition $A_{condition}$ manually, there is also a generic condition that allows to separate artifact types stored in the same main table. In this case, the main table T_m typically has a particular column C_{type} where the value of C_{type} indicates the artifact type to which an entry of T_m belongs. Let $v_1, ..., v_n$ be values found in C_{type}. Then calling *createArtifact*() with condition $C_{type} = v_i$ for each $i = [1, n]$, allows to extract all artifact types defined by C_{type}. This can be generalized to multiple columns and automated in our second function *createArtifactsByColumnValues*($\mathfrak{S}_A$, $\mathbb{C}_e$) having as arguments the artifact schema and a set of columns C_e that distinguish the different artifact types. For example, given (1) the aforementioned artifact schema $\mathfrak{S}_A$ whose $\mathbb{T}_A = \{SD\}$ and $T_m = SD$ and (2) the column *Document type*, we find two distinct values in *Document type*, "Sales Order" and "Return Order", which lead to the automatic discovery of two artifacts, $A_{SalesOrder}$ (shown in Table 1) and $A_{ReturnOrder}$ (shown in Figure 12 on the right-hand side).

It is also possible to identify multiple event types from the same timestamped column by using a condition similar to $A_{condition}$ used in *createArtifact*($\mathfrak{S}_A, A_{name}, A_{condition}$). Assume an artifact schema $\mathfrak{S}_A$ whose $\mathbb{T}_A = \{$SD, Changes$\}$, besides the two event types *Date created* and *Last change* we identified earlier, there is a third time-stamped

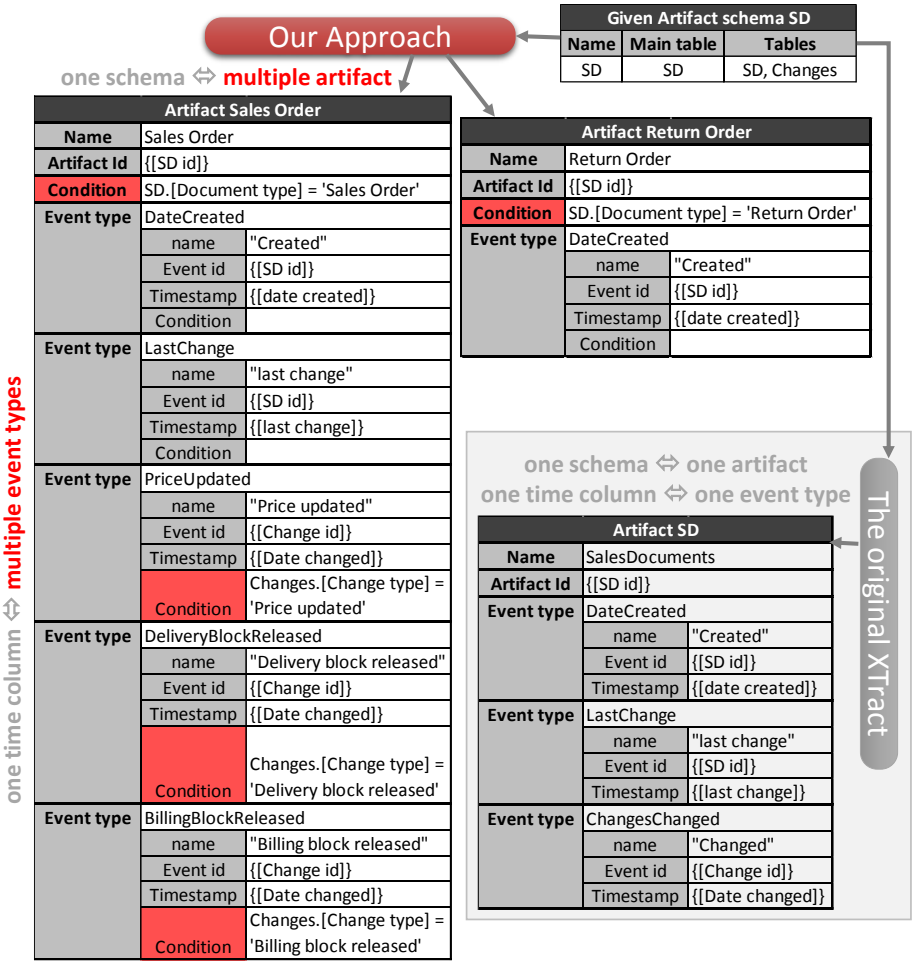

Fig. 12: Comparing the artifacts obtained using the XTract approach and our approach

column *Date changed*, which can either be considered as one event type, or we can use the column *Change type* to indicate different event type conditions resulting in three different event types (similar to $A_{condition}$). An example of the artifact *Sales Order* we then discovered is shown in the middle of Figure 12; see [51] for the technical details. Furthermore, the approach presented allows users to add, delete and modify each event type, event type attributes and case attributes.

Figure 12 demonstrates the difference in artifacts returned by the XTract approach and our approach. Given the artifact schema SD containing the table SD as the main table and table Changes, the XTract approach returns one artifact SD shown on the left hand side in Figure 12. In contrast, our approach allows the user to indicate the column *document type* as a condition column constituting $\mathbb{C}_e$ in the function *createArtifactsBy-*

ColumnValues($\mathfrak{S}_A$, $\mathbb{C}_e$). Two artifacts *Sales Order* and *Return Order* are then identified, as shown on the right hand side in Figure 12.

4.5 Artifact Extraction

To extract an event log for an artifact, the identified artifact is used to create a log mapping which maps the components of an artifact type to the components of a log. For example, the artifact identifier $\mathbb{C}_{Aid}$ is mapped to the trace identifier attribute; the event type identifier $\mathbb{C}_{Eid}$ is mapped to the event identifier attribute, each timestamp column C_{time} is mapped to the timestamp attribute. Note that the set of the interactions of each artifact type is still empty and no information about interactions is mapped nor extracted for now.

Next, the log mapping is used to create SQL queries which select the instances according to the log mapping and join the events and attributes to the instances. The result of the queries is stored in a cache database, which is then used to write event log files in XES format by calling the functions of the OpenXES library[3].

Figure 13 shows an example of an event log extracted for the artifact *Sales Order* in Figure 12. Only two entries in table SD satisfy the artifact condition *SD.[Document type] = "Sales Order"*, S1 and S2, which respectively result in two traces with S1 and S2 as trace identifiers. According to the event type definitions, we are able to extract five events for S1 and two events for S2. The corresponding values for the *ID*, *name*, *timestamp* and *attributes* of an event are also extracted. For example, event e2 of case S1 is extracted according to event type *Price updated* and thus has *Price updated* as name, the value 1 (which is the value of its primary key of column *Change id* in table *Documents Changes*) as event ID, 17-5-2020 as timestamp (which is the value of column *Date changed*), and some event attributes extracted from table *Documents Changes* (as example). Other events are extracted using the same method, see Figure 13.

Our approach basically reused the orginal XTract approach [44] [18] and only extended it by appending the conditions in the WHERE-part of the queries. For technical details, we refer to [51] [44].

Log	**Name**	**Sales Order**		
Trace	**ID = S1, Document type = "Sales Order", value = 100**			
	ID	name	timestamp	event attrs
Event e1	S1	Date created	16-5-2020	-
Event e2	1	Price updated	17-5-2020	Old value = "100", New value = "80"
Event e3	2	Delivery block released	19-5-2020	Old value = "x", New value = "-"
Event e4	3	Billing block released	19-5-2020	Old value = "x", New value = "-"
Event e5	S1	Last change	10-6-2020	-
Trace	**ID = S2, Document type = "Sales Order", value = 200**			
	ID	name	timestamp	event attrs
Event e1	S1	Date created	17-5-2020	-
Event e2	S1	Last change	31-5-2020	-

Fig. 13: An example of event log extracted for artifact *Sales Order*

[3] http://www.xes-standard.org/openxes/start

4.6 Handling Generalization

The previous sections described how to identify and extract artifact types and their life-cycle information from a relational data source. The presented steps allowed to revert vertical partitioning, and horizontal and vertical anti-partitioning of the given data. Here, we discuss how to handle horizontal partitioning in the data source, that is, when information about a conceptual general artifact is not stored as such, but has been distributed over many different tables. For example in Fig. 1, one could be interested in extracting a general "Documents" artifact rather than one artifact for different document types.

Generalizing different artifact types into one general artifact is similar to generalizing entities and highly depends on the given relational schema [52].

1. The specialization is materialized in the relational schema by a discriminating attribute. In this case all artifact types are found in the same tables, and hence will be contained the same artifact schema. When defining the artifact type, one simply specifies are more general discriminating condition $A_{condition}$ in Def. 6. The resulting general artifact type then contains more or even all event types and attributes in the artifact schema.
2. The specialization is materialized as an "IS-A" relationship with a "general table" and foreign keys from its specializations. In this case, the general table and all specializations of interest become part of the artifact schema, and the general table is chosen as main table. Artifact type definition proceeds as described above.
3. The specialization is materialized as separate tables without an "IS-A" relationship. In this case no generalizing main table for the different specializations can be defined. Two solutions are possible. (1) One can first extract the life-cycle event log for each specialized artifact, and then merge the resulting event logs into one generalized event log. Prefixing values of identifier attributes prevents collisions of different specializations. (2) For the purpose of the analysis, one could transform a copy of the original relational source, for example by introducing an "IS-A" relationship with appropriate foreign keys.

4.7 Artifact Life-cycle Discovery

For each artifact A_i which we identified on the database level, we have shown how to extract an event log L_i. To discover the life cycle M_i of artifact A_i from the corresponding log L_i, we can reuse existing process discovery algorithms. For discovering a life-cycle model from a log L_i, generally the same considerations apply as for discovering a process model from L_i. There are various discovery algorithms available and the user has to pick one that satisfies her desired criteria.

For Life-cycle discovery, we provide no new technique but re-use existing process discovery techniques that create from an event log of a process, a process model. The advantages and disadvantages of these techniques have been discussed extensively on a conceptual and on empirical level [31]. A user can choose a suitable algorithm based on these and the desired characteristics and quality criteria with respect to the target model (fitness, precision, simplicity, generalization). Often, different mining algorithms can

be used depending on the purpose, e.g., ILP miner for optimizing fitness and precision, ETM to balance quality criteria, heuristics miner to show a simple model without complex routing logic (though without operational semantics), etc. One characteristic that is specific to artifacts is that, unlike classical workflow processes, concurrency in the discovered may be of secondary concern (i.e., a business object may never be accessed concurrently by two users/processes at the same time, thus a transition system model [TS miner] could provide a the right representational bias that does not introduce artificial concurrency). The subsequent interaction discovery requires that each event of an artifact is translated into (exactly one) action of the life-cycle model as otherwise interactions cannot be discovered properly. This assumption excludes algorithms that may discard certain events during discovery or that may duplicate tasks.

For the remainder, we assume that $Miner(L)$ denotes some life-cycle miner that returns a process model M of the life-cycle of A_i. For the artifact type *Sales Order* we shown in Figure 12 and the event log that we extracted for this artifact shown in Figure 13, we discovered a life-cycle of this artifact shown in Figure 14 by applying the flexible heuristic miner [25].

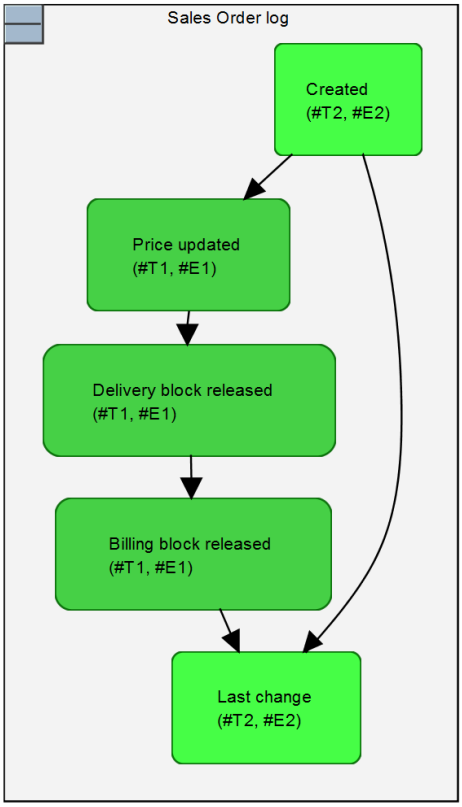

Fig. 14: The life-cycle discovered for the artifact *Sales Order*

5 Interaction Discovery

Having identified artifact life-cycles, we can now approach the problem of discovering interactions between artifacts. In Section 5.1, we begin with an analysis of the possible interactions between artifacts. In Section 5.2, we illustrate how to identify interactions between artifact types and between instances. Enriching the logs extracted for artifacts with these interactions is discussed in Section 5.3. These enriched logs can then be used to identify interactions between event which is discussed in Section 5.4. Finally, Section 5.5 presents the computation of artifact-centric models.

5.1 Interaction Types and Definitions

Interactions between artifacts can be studied from different levels of abstractions. Figure 15 illustrates four levels of interactions: interactions (a) between artifact instances, (b) between artifact types, (c) between events of artifacts, or (d) between event types of artifacts.

The (a) *artifact instance level interactions* denotes interactions between two artifact instances such as the sales order S1 and the return order S3. The (b) *artifact type level interactions* refer to interactions between two artifact types such as the *Sales Order* and the *Return Order*. The existence of interactions between two instances indicates, to some extent, the existence of an interaction between the two artifact types. The (c) *event level interactions* are interactions between two events such as the *Latest Change* event with id S1 and timestamp '10-6-2020' of the sales order S1 and the *Date Created* event with id S3 and timestamp '10-6-2020' of the return order S3 that is related to sales order S1. The (d) *event type level interactions* denote interactions between two event types such as the *Latest Change* event type of the sales order artifact and the *Date Created* event type of the return order artifact. For the sake of brevity, we also refer to artifact instance level interactions as instance level interactions and refer to artifact type level interactions as type level interactions.

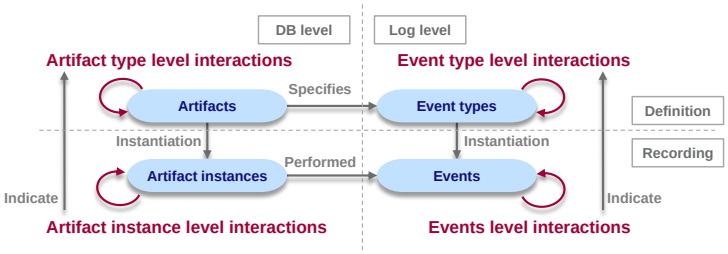

Fig. 15: The four levels of interactions

Direct Type- and Instance Level Interaction Before formally defining interactions, we explain briefly the reasoning behind the existence of interactions. Starting from two given artifacts, each of which has a main table containing its instances. If there exists a reference between the two main tables and there is an instance in one of the two tables

referring to an instance in the other table, then there may be an interaction between the two artifacts. This idea leads to our definition of the direct interactions.

Definition 7 (Direct artifact type level interactions, Parent artifacts, Child artifacts). *Let $\mathbb{A}$ be a set of artifacts and $\mathbb{F}$ a set of references. $(A_S, F, A_T) \in \mathbb{A} \times \mathbb{F} \times \mathbb{A}$ is a direct, artifact type level interaction between the two artifacts A_S and A_T, if and only if, (1) $F = \langle T^S_m, \mathbb{C}^S_{Aid}, T^T_m, \mathbb{C}^T_c, S_{condition} \rangle$ is a reference from some attributes $\mathbb{C}^T_c$ of the main table of A_T to the artifact identifier $\mathbb{C}^S_{Aid}$ of A_S, and (2) there has to be an entry s in T^S_m referring to an entry t in T^T_m that s satisfies the condition $A_S.A_{condition}$, and t satisfy the condition $A_T.A_{condition}$.*

We denote the artifact A_S as the parent artifact *and the artifact A_T as the* child artifact.

For example, $(A_{SalesOrder}, F_2, A_{ReturnOrder})$ is a direct, artifact type level interaction. The *direct instance level interactions* are instantiated from the interactions on the type level by joining the main tables based on direct type level interactions. Below, we show an SQL query that select the direct instance level interactions of the direct type level interaction $(A_{SalesOrder}, F_2, A_{ReturnOrder})$. The resulting instance level interaction is $(S1, S3)$. Note that the query can be generated automatically using the artifacts and references. For technical details, we refer to [51].

```
SELECT DISTINCT
 SalesOrder.[SD ID] AS [SO id],
 ReturnOrder.[SD ID] AS [RO id]
FROM  SD AS SalesOrder
INNER JOIN  SD AS ReturnOrder
ON SalesOrder.[SD ID] = ReturnOrder.[REFERENCE ID]
 AND SalesOrder.[DocumentType] = 'Sales Order'
 AND ReturnOrder.[DocumentType] = 'Return Order'
```

Lemma 1 (Reference property). *Given a direct, type level interaction (A_S, F, A_T), an instance in the parent artifact A_S can be linked to zero or multiple instances of the child artifact A_T, whereas an instance in the child artifact A_T can only be linked to zero or one instance of the parent artifact A_S.*

Proof. According to the definition of a direct type level interaction, the reference F is a direct foreign key between the two main tables. Therefore, the property of a foreign key (or the structure of a relational database) indicates that an entry in the parent table (in this case the main table $A_S.T_m$ of the parent artifact) is linked to zero or multiple entries in the child table (in this case the main table $A_T.T_m$ of child artifact), whereas an entry in the child table is only linked to zero or one entry in the parent table, which automatically implies the Reference property since each entry in the main table refers to an artifact instance. □

To represent a set of artifacts and the direct type level interactions between them, we use a graph of which the vertices are the artifacts and the edges are the interactions between the artifacts. We called the graph an *interaction graph*.

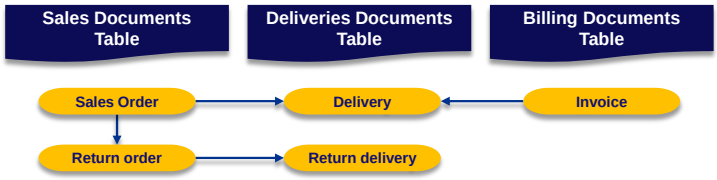

Fig. 16: The artifact type level interactions of the running example.

Definition 8 (Interaction graphs). *Given a set $\mathbb{A}$ of artifacts and a set $\mathbb{F}$ of references, $G = (\mathbb{A}, D)$ with $D \subseteq \mathbb{A} \times \mathbb{F} \times \mathbb{A}$ is an* interaction graph*, if and only if, for each $d = (A_s, F, A_t) \in D$, d is a direct, artifact type level interaction between artifacts A_s and A_t.*

The set of outgoing edges of artifact $A_S \in \mathbb{A}$ is denoted by $outEdges(A_S) = \{(A_S, F_i, x) \in D \mid x \in \mathbb{A}\}$. Similarly, the incoming edges of artifact A_S is defined as $inEdges(A_S) = \{(x, F_i, A_S) \in D \mid x \in \mathbb{A}\}$.

For example, Figure 16 shows the interaction graph of the OTC example including the direct interactions (denoted by the arcs) between the artifacts (denoted by the elliptic nodes). Note that there could be multiple edges (thus multiple direct type level interactions) between two artifacts because there could be different references linking the artifacts.

Indirect Type- and Instance Level Interaction Intuitively, the direct interactions between artifacts described above are sufficient to study how artifacts interact with each other. However, in practice two artifacts *A* and *B* of interest may not interact directly, but only indirectly via another artifact *C*. If artifact *C* is not of interest for the analysis, for instance because it is at a lower level of detail, we would like to omit *C* from the artifact extraction yet still be able to analyze the interaction between *A* and *B*. For instance, if the return orders shown in Figure 16 can be ignored, we would still like to be able to identify the interactions between the *Sales Order* and the *Return Delivery*. More importantly, this allows us to consider a many-to-many relation, which normally is composed by a one-to-many and a many-to-one relation, as a single interaction by omitting the intermediate artifact (or table). A real-life example encountered is that ERP systems in general have a document header and document lines structure where documents are connected through document lines instead of directly connected through the headers. To allow a user to study interactions between the artifacts of her choice, we define *artifact type level interactions*, direct or indirect, as follows.

Definition 9 (Artifact type level interactions). *Let $\mathbb{A}$ be a set of artifacts and $\mathbb{F}$ be a set of references. An* artifact type level interaction *$I_{S,T} = \langle d_1, \cdots, d_n \rangle \in (\mathbb{A} \times \mathbb{F} \times \mathbb{A})^*$ between artifacts A_S and A_T with $n \geq 1$ is a sequence of direct interactions $d_i = (A_{P_i}, F_i, A_{C_i}) \in \mathbb{A} \times \mathbb{F} \times \mathbb{A}$ and $1 \leq i \leq n$, which satisfies one of the following properties (also shown in Figure 17):*

(1) the parent artifact A_{P_1} of the first direct interaction d_1 is the artifact A_S, and the child artifact A_{P_n} of the last direct interaction d_n is the artifact A_T, and for $1 \leq i < n$, $A_{C_i} = A_{P_{i+1}}$; or

(2) *the child artifact A_{C_1} of first direct interaction d_1 is the artifact A_S, and the parent artifact A_{P_n} of the last direct interaction d_n is the artifact A_T, and for $1 \leq i < n$, $A_{P_i} = A_{C_{i+1}}$; or*

(3) *there is a number k and $1 \leq k \leq n$, for $1 \leq i < k$, $A_{C_i} = A_{P_{i+1}}$, and $A_{C_k} = A_{C_{k+1}}$, and for $k < i < n$, $A_{P_i} = A_{C_{i+1}}$, and the parent artifact A_{P_1} of first direct interaction d_1 is the artifact A_S, and the parent artifact A_{P_n} of the last direct interaction d_n is the artifact A_T.*

Additionally, in all cases the number of instance level interaction is greater than zero.

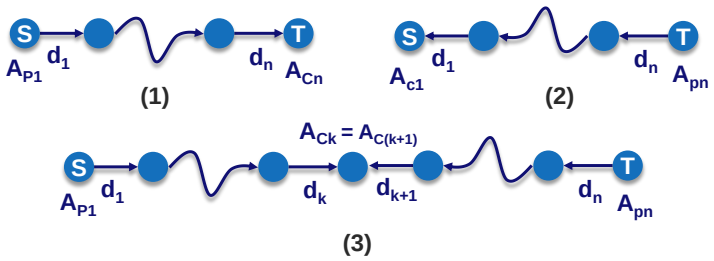

Fig. 17: Type level interactions

Definition 10 (Strongly connected interactions). *The artifact type level interactions $I_{S,T} = \langle d_1, \cdots, d_n \rangle$ between artifacts A_S and A_T that satisfy the first or the second properties in Definition 9, we called them the* strongly connected interactions. *The* length of strong joins *of this interaction is n.*

Figure 18(a) shows a *strongly connected interaction* consisting of two direct interactions $(A_{SalesOrder}, F_i, A_{ReturnOrder})$ and $(A_{ReturnOrder}, F_j, A_{ReturnDelivery})$ linked by $A_{ReturnOrder}$. Since both interactions share the return order artifact, it is possible to join the three main tables based on the two direct interactions. As the references are joined in the same direction, we have the guarantee that an instance of the artifact A_{T_i} is only linked to zero or one instance of the artifact A_{S_i}, based on the reference property. This property also indicates that an instance of the artifact A_{S_i} is linked to all related instances of the artifact A_{T_j}, and these instances of A_{T_j} are not linked to any other instances of the artifact A_{S_i}.

Definition 11 (Weakly connected interactions). *The artifact type level interactions $I_{S,T} = \langle d_1, \cdots, d_n \rangle$ between artifacts A_S and A_T that satisfy the third property in Definition 9, we called them the* weakly connected interactions. *The* length of strong joins *of this interaction is k, and the* length of weak joins *is $m = n - k$. When the length m of weak joins is greater than zero, the interaction instances could be an over-approximation. We define functions $lengthSJ : I \to \mathbb{N}$ and $lengthWJ : I \to \mathbb{N}$ to retrieve the length of strong joins (k) and the length of weak joins (m) of an interaction, respectively.*

Figure 18(b) shows a weakly connected interaction comprised two direct interactions $(A_{SalesOrder}, F_i, A_{Delivery})$ and $(A_{Invoice}, F_j, A_{Delivery})$ linked to the same child artifact

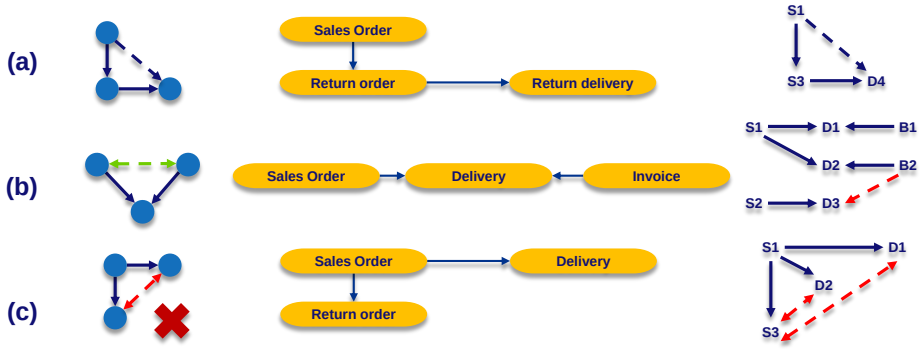

Fig. 18: The three possible cases of indirect interactions constituted of two direct interactions

$A_{Delivery}$. The instances of the artifact *Sales Order* are explicitly linked to the instances of the artifact *Invoice* via the instances of the artifact *Delivery*, as shown by the graph in Figure 18(b) on the right-hand side. For example, S1 linked to B1 via D1, S1 linked to B2 via D2, and S2 linked to B2 via D3. This explicitness is because an instance of the artifact *Delivery* can only be linked to one instance of *Sales Order* and one instance of *Invoice* (e.g. (S1, D1, B1)) based on the reference property. However, also based on the reference property, there is the possibility that an instance of the artifact *Invoice* is linked to multiple sales orders (e.g. B2). As a result, when the invoices B1 and B2 are selected as the indirect instance level interactions for the sales order S1, we have also included a part of B2 that is not related to S1 but related to S2. Therefore, there is an *over-approximation* when a weakly connected interaction is used to derive instance level interactions.

Invalid Interactions. Compared to weakly connected interactions, sequences that have two consecutive direct interactions (A_{S_i}, F_i, A_{T_i}) and (A_{S_i}, F_j, A_{T_j}) with the same parent artifact A_{S_i} and different target artifacts is invalid. Figure 18(c) shows an example: the interaction composed of $(A_{SalesOrder}, F_i, A_{Delivery})$ and $(A_{SalesOrder}, F_j, A_{ReturnOrder})$. Since an instance of the artifact *Sales Order* can be linked to multiple instances of the artifact *Delivery* and multiple instances of the *Return Order*, it is impossible to determine the exact relations between the instances only based on the references. For instance, D1 and D2 refer to S1, but also S3 refers to S1, but there is no explicit relation that indicates whether D1 is then related to S3 or D2 to S3. Therefore, this type of interaction is considered to be invalid.

Possibility of Identifying Event Level Interactions The definition and the examples demonstrate that type level interactions and instance level interactions between two artifacts can be identified on the database level. However, the database is not suitable for identifying event level interactions as we illustrated on our OTC example. The direct interaction between the *Sales Order* and the *Return Order* does not indicate whether the *Date Created* event type of *Return Order* is (causally) related to the *Date Created* event type of *Sales Order* or to the *Latest Change* event type of the *Sales Order*. Moreover, we aim at identifying a "direction" of interaction such as "event e_1 caused event e_2".

This information requires to compare the order of different events in time. For this purpose, the event log structure and process mining techniques are much more suitable than the tables of a relational database. Therefore, we propose to identify event level interactions *after* we extracted the artifact event logs (which will be discussed in Section 5.4). Though in order to allow finding event level interactions in event logs, we first have to enrich the event logs (extracted in Section 4) with information about artifact level interactions. The algorithms for enriching logs are presented in Section 5.3.

5.2 Artifact Type Level Interaction Discovery

To compute artifact type level interactions, our method consists of two parts. First, given a set of artifacts $\mathbb{A} = \{A_1, \cdots, A_n\}$, we compute an interaction graph $G = (\mathbb{A}, D)$. From the interaction graph G, we then compute a set of type level interactions $\mathbb{I}$, direct and indirect, for each artifact.

Interaction Graph Construction To solve the first part, we introduce the algorithm *ConstructInteractionGraph*$(\mathbb{A}, \mathbb{F})$. We first define an auxiliary function $\mathcal{A}rtifacts(T) = \{A \in \mathbb{A} \mid A.T_m = T\}$ to retrieve a set of artifacts that shares the same main table T. Then, using $\mathcal{A}rtifacts(T_p)$ and $\mathcal{A}rtifacts(T_c)$, for each reference $F \in \mathbb{F}$ between T_p and T_c, we retrieve all artifacts A_p that has the parent table T_p and all artifacts A_c of the child table T_c, respectively (see Line 3). For each combination of A_p and A_c with the reference F between them, we select the count using the *countSelect* query defined in Section 5.1 to verify whether it is a direct interaction (see Line 4). If the count is greater than 0, then an edge (A_p, F, A_c) is added to D as an edge in the interaction graph G.

Algorithm *ConstructInteractionGraph*$(\mathbb{A}, \mathbb{F})$
1. $D \leftarrow \varnothing$
2. **for** $F = \langle T_p, \mathbb{C}_p, T_c, \mathbb{C}_c, S_{condition} \rangle \in \mathbb{F}$
3. **do for** $A_p \in \mathcal{A}rtifacts(T_P), A_c \in \mathcal{A}rtifacts(T_c)$
4. **do if** $countSelect(A_p, F, A_c) \geq 0$
5. **then** $D \leftarrow D \cup (A_p, F, A_c)$
6. **return** $G = (\mathbb{A}, D)$

Computing Artifact Type Level Interactions To solve the second part, i.e. discovering a set of (direct and indirect) interactions for each artifact given, we present the algorithm *CalculatesInteractions*$(\mathbb{A}, G, r, k, m)$, with $r > 0$, $k \geq 1$ and $0 \leq m \leq k$, which returns the set $\mathbb{A}$ of artifacts, and for each artifact $A_S \in \mathbb{A}$ we fill in the set $\mathbb{I}_S$ of interactions with the following requirements. For each interaction $I_S \in \mathbb{I}_S$, the length of the strong joins is at most k, the length of weak joins is at most m and the number of distinct instance level of interactions is at least r.

The algorithm *CalculatesInteractions* runs the following. For each artifact $A_S \in \mathbb{A}$, it initializes the set $\mathbb{I}$ of interactions as empty set. Next, taking A_S as the start point, the

algorithm simply explores all paths in the interaction graph G with a depth first strategy by recursively calling the algorithm *calculateJoins*. The paths that satisfy the requirements w.r.t. k, m, and r are returned as type level interactions for A_S (see Appendix for details).

Algorithm *CalculatesInteractions*$(\mathbb{A}, G, r, k, m)$
1. **for** $A_S \in \mathbb{A}$
2. **do** $\mathbb{I} \leftarrow \varnothing$
3. **for** $(A_S, F, A_t) \in outEdges(A_S)$
4. **do** $I_{current} \leftarrow \langle (A_S, F, A_t) \rangle$
5. **if** $countSelect(I_{current}) \geq r$
6. **then** $\mathbb{I} \leftarrow \mathbb{I} \cup I_{current}$
7. calculateJoins(A_S, A_t, $I_{current}$, $\mathbb{I}$, r, $k-1$, m)
8. $A.\mathbb{I} \leftarrow \mathbb{I}$
9. **return** $\mathbb{A}$

After the algorithm *CalculatesInteractions*$(\mathbb{A}, G, r, k, m)$ terminates, we have, for each artifact $A \in \mathbb{A}$, the set $A.\mathbb{I}$ of interactions calculated and returned. Users can select a set of desired type level interactions from the set $A.\mathbb{I}$.

Single-Sided Discovery Note that we made a design decision to restrict the identification of symmetric interactions on the parent artifact by using the condition that the length of strong joins is at least the length of weak joins, in addition to that the number of strong joins greater than zero (i.e. $k \geq 1$). In other words, recalling the three situations of type level interaction shown in Figure 17, we only obtain interactions of the parent artifact A_S that satisfy (1), or (3) with $m \leq k$. The interactions that satisfy (2) or (3) with $m > k$ will only be extracted as the interactions of artifact A_T. Note that this decision will also effect the event logs we extract. An instance level interaction will only be extracted for the parent artifact.

There are several advantages to only identify interactions on the one side. First, limiting the interactions to the parent artifact can prevent duplicated attributes to be extracted to improve the performance of log extraction. Furthermore, this decision also limits the number of discovered interactions to decrease the number of manual selections that are needed by users. Moreover, we retain the parent and child structure and the reference property which can be reused during the process discovery. We have observed one disadvantage. For example, when a child artifact instance has an interaction with a parent artifact instance but this parent instance is not extracted in the event log, then we miss the information that the child instance has an interaction in the event log. Changing the proposed algorithm to extract interactions on both side is considered as future research.

5.3 Enriched Log Extraction

Using the method described in Section 5.2, we discovered for each artifact a set of type level interactions and their instance level interactions. These instance level interactions are used to enrich logs extracted for artifacts (which is discussed in Section 4.5) and thus retained for the mining phase. The method runs as follows.

Each artifact type level interaction is used to generate an SQL query that joins the main tables of artifacts and select interactions between the artifact instances. These instance level interactions are considered as (a list of) attributes of an artifact instance and extracted as the case attributes for the artifact instance. For example, Figure 19 shows the artifact type *Sales Order* filled with two type level interactions $I1$ and $I2$ (omitted the five event types in Figure 11 for the sake of brevity). Using similar automatically generated SQL queries as we discussed in Section 5.1, we obtain {(S1, S3)} and {(S1, D1), (S1, D2)} as instance level interactions of sales orders S1 with return orders and deliveries, respectively. For sales order S2, only interactions with deliveries are found, {(S2, D3)}; the instance level interactions between S2 and return orders is an empty set. The instance level interactions (stored in the cache database) can now be extracted to enrich the event log of artifact *Sales Order*. Thus, the log shown in Figure 13 is now enriched with the three case attributes as Figure 20 illustrated with the highlighted parts (we omitted the events for the sake of brevity). For instance, trace S1 has now two case attributes additionally: one indicates its interacting traces in the artifact *Delivery* which are D1 and D2; another indicates its interacting trace in artifact *Return Order* which is S3. For technical detail, we refer to [51].

Artifact Sales Order	
Artifact name	Sales Order
Artifact Identifier	{[SD id]}
Condition	SD.[Document type] = 'Sales Order'
Interaction I1	<A_SalesOrder, F1, A_ReturnOrder>
Interaction I2	<A_SalesOrder, F2, A_Delivery>

Fig. 19: An example of artifact *Sales Order* with type level interactions

Log	Name	Sales Order		
Trace		**ID = S1, Document type = "Sales Order", value = 100**		
		Interaction_Delivery = {D1, D2}, Interaction_ReturnOrder = {S3}		
Events	ID	name	timestamp	event attrs
Trace		**ID = S2, Document type = "Sales Order", value = 200**		
		Interaction_Delivery = {D3}		
Events	ID	name	timestamp	event attrs

Fig. 20: An example of event log extracted for artifact *Sales Order* with instance level interactions as case attributes

5.4 Event Type Level Interaction Discovery

We have shown how to identify type level interactions and enrich the logs with trace attributes of which instance interacts with which other instances. From a log perspective,

a trace t_1 of artifact A_1 referring to a trace t_2 of artifact A_2 means that some of the events in t_1 interact with some of the events in t_2. In the following, we present techniques to identify these interactions between events and event types. First, we retrieve for any two artifacts A_1 and A_2 the pairs of interacting traces. Then, we provide two different kinds of techniques to identify patterns of possible event interactions between the interacting traces of A_1 and A_2. From these, we then derive event type level interactions. In particular we distinguish frequent interactions and infrequent interactions (outliers).

Preliminaries - Event logs We use the formal definition of event log in [1]. In general, an event log comprises a list of traces of which each *trace* contains all events that occurred in a case, i.e., an execution of the process.

Definition 12 (Event universe $\mathcal{E}$). *Let $\mathcal{E}$ be the* event universe*, i.e. the set of all possible event identifiers. Events may be characterized by various* attributes*. Let AN be a set of attribute names. For any event $e \in \mathcal{E}$ and name $n \in AN$: $\#_n(e)$ is the value of attribute n for event e. If event e does not have an attribute named n, then $\#_n(e) = \perp$ (null value).*

For example, $\#_{type}(e)$ is the event type associated to event *e* describing the activity that has been executed; $\#_{time}(e)$ is the timestamp of event *e* (also denoted by $\mathcal{T}(e)$).

Definition 13 (Case universe $\mathcal{L}$). *Let $\mathcal{L}$ be the* case universe*, i.e. the set of all possible case identifiers. A case also has attributes. For any case $c \in \mathcal{L}$ and name $n \in AN$: $\#_n(c)$ is the value of attribute n for case c. If case c does not have an attribute named n, then $\#_n(c) = \perp$.*

Definition 14 (Traces). *Each case has a special mandatory attribute* trace*: $\#_{trace}(c) = \sigma_c \in \mathcal{E}^*$, which is a finite sequence of events $\sigma \in \mathcal{E}^*$ such that each event appears only once. We assume traces in a log contain at least one event, i.e. $\#_{trace}(c) \neq \langle\rangle$.*

Definition 15 (Logs). *An* event log *is a set of cases $L \subseteq \mathcal{L}$ such that each event appears at most once in the entire log. We use $\mathcal{A}$ to denote the set of all event types appearing in log L, i.e. $\mathcal{A} = \{\#_{type}(e) \mid c \in L \wedge e \in \#_{trace}(c)\}$.*

We also define the following short notations for your convenience.

Definition 16 (Projection $\mathcal{F}$). *Given a log L and an event type $E \in \mathcal{A}$, $\mathcal{F}_E(L)$ filters L and retain, for each of its traces, the event of event type E only.*

For example, the trace sales order σ_{S1} consists of two events, i.e. $\langle(S1, \textit{created}, \text{16-5-2020}), (S1, \textit{latestchange}, \text{10-6-2020})\rangle$. If we apply the project function on the trace σ_{S1} with $E = \textit{created}$, we obtain $\mathcal{F}_{\textit{created}}(\sigma_{S1}) = \langle(S1, \textit{created}, \text{16-5-2020})\rangle$.

Definition 17 (Trace precedence $<_\mathcal{T}$). *Given two traces σ_s and σ_t, $\sigma_s <_\mathcal{T} \sigma_t$, if and only if, for each event $e_s \in \sigma_s$ and $e_t \in \sigma_t$, the timestamp $\#_{time}(e_s)$ of the event e_s is before the timestamps $\#_{time}(e_t)$.*

For example, $\mathcal{F}_E(\sigma_s) <_\mathcal{T} \mathcal{F}_E(\sigma_t)$ means that each event $e_s \in \sigma_s$ which has the event type *E* is executed earlier than each event $e_t \in \sigma_t$ which has the event type *E*. $\sigma_s =_\mathcal{T} \sigma_t$, $\sigma_s >_\mathcal{T} \sigma_t$, $\sigma_s \leq_\mathcal{T} \sigma_t$ and $\sigma_s \geq_\mathcal{T} \sigma_t$ are also similarly defined.

Interacting Traces We define a function $\#_{\mathcal{I}\_T}(c_s)$ which returns the set of the instance level interactions of case c_s with the cases in artifact T.

Definition 18 (Function $\#_{\mathcal{I}\_T}(c)$). *Given a log L, of which each case is enriched with instance level interactions, a case c, and artifact T, we use the notation $\#_{\mathcal{I}\_T}(c)$ to retrieve the instance level interactions between the case c and the set of cases of artifact T.*

If we use the log shown in Figure 20 as an example, $\#_{\mathcal{I}\_Delivery}(S1) = \{D1, D2\}$, $\#_{\mathcal{I}\_ReturnOrder}(S1) = \{S3\}$, and $\#_{\mathcal{I}\_Delivery}(S2) = \{D3\}$. If the case has no interaction with the given artifact, an empty set is returned, e.g. $\#_{\mathcal{I}\_ReturnOrder}(S2) = \varnothing$.

Definition 19 (Function $\mathcal{I}$). *Given two event logs L_S and L_T, we define $\mathcal{I}(L_S, L_T) = \{(c_s, c_t) \mid c_s \in L_S \wedge c_t \in L_T \wedge c_t \in \#_{\mathcal{I}\_T}(c_s)\} \subseteq L_S \times L_T$ as the* set of trace level interactions *between the logs L_S and L_T, where each $(c_s, c_t) \in \mathcal{I}(L_S, L_T)$ indicates that there is an instance level interaction between the two traces c_s and c_t seen from the* parent artifact A_S *to the* child artifact A_T.

We illustrate the function $\mathcal{I}$ with the OTC example shown in Figure 18(b). Assume we have extracted an event log of the artifact *Sales Order* with trace level interactions to the artifact *Invoice* and an event log of the artifact *Invoice* (i.e. the artifact *Sales Order* is the parent artifact). We use the trace id to represent the trace. Thus the set of trace level interactions $\mathcal{I}(L_{SalesOrder}, L_{Invoice}) = \{(S1, B1), (S1, B2), (S2, B2)\}$.

Definition 20 (Event type level interactions (ETLI)). *Given two artifact types A_S and A_T, and their set of event types $\mathcal{A}_S$ and $\mathcal{A}_T$, respectively, an event type level interaction (a_p, a_c) between A_S and A_T satisfies, at least, the requirement that $a_p \in \mathcal{A}_S$ and $a_c \in \mathcal{A}_T$, or vice versa.*

In the following, we present two methods to identify event type level interactions between the events of two interacting traces L_S and L_T. More specifically, we identify the relation $X \subseteq (\mathcal{A}_S \times \mathcal{A}_T) \cup (\mathcal{A}_T \times \mathcal{A}_S)$ where $\mathcal{A}_S$ and $\mathcal{A}_T$ denote the event types of L_S and L_T, respectively.

ETLI Discovery by Merging Logs The first idea for discovering event type level interactions is to compute a merged trace from any two interacting traces of artifacts A_1 and A_2 (by putting all their events in one trace). Doing this for all interacting traces gives a merged log. Applying a classical process discovery algorithm then allows to identify causal dependencies between activities from artifacts A_1 and A_2. These can be interpreted as event type level interactions between A_1 and A_2.

We present the *CalcETLInteractionsByMergingLogs* algorithm below. In general, for each $(\sigma_s, \sigma_t) \in \mathcal{I}(L_S, L_T)$, we merge the two traces to a new trace using the merge function $\mathcal{M}$ (see Line 5) that simply builds the union of the events of two traces σ_s and σ_t and orders all events by their timestamp (see [51] for a formal definition). All merged traces are added to the log L_{new} containing events originally from two artifacts. Now, a process discovery algorithm *Miner*() can be applied on the merged log L_{new} to discover dependencies between the two sets of event types, i.e. $\mathcal{A}_S$ and $\mathcal{A}_T$. (see Line 7).

Each direct succession (E_i, E_j) between the two event types E_i and E_j discovered where $E_i \in \mathcal{A}_S$ and $E_j \in \mathcal{A}_T$ (or vice versa) is considered an *event type level interaction* between artifacts A_S and A_T, and added to the result X (see Lines 8-9).

Algorithm *CalcETLInteractionsByMergingLogs*(L_S, L_T)
1. **if** $\mathcal{I}(L_S, L_T) = \varnothing$
2. **then return** $\varnothing$
3. **else** $L_{new} \leftarrow [], X \leftarrow \varnothing$
4. **for** $(\sigma_s, \sigma_t) \in \mathcal{I}(L_S, L_T)$
5. **do** $\sigma_{new} \leftarrow \mathcal{M}(\sigma_s, \sigma_t)$
6. if $\sigma_{new} \neq \langle\rangle$ then add σ_{new} to L_{new}
7. $M_{new} \leftarrow Miner(L_{new})$
8. **for** (E_i, E_j) where E_j is a direct successor of E_i in the model M_{new}, and $(E_i, E_j) \in \mathcal{A}_S \times \mathcal{A}_T$ or $(E_i, E_j) \in \mathcal{A}_T \times \mathcal{A}_S$
9. **do** add (E_i, E_j) to X.
10. **return** (L_{new}, X)

For example, if we use the heuristic miner as the $Miner(L_{new})$, a simple causality net $DG = (\mathcal{A}_{new}, D)$, in which $D \subseteq \mathcal{A}_{new} \times \mathcal{A}_{new}$ indicates the direct succession between event types, is obtained from the log L_{new}. Here, a dependency $(E_i, E_j) \in D$ with $E_i \in A_{L_S}$ and $E_j \in A_{L_T}$ (or vice versa) is returned as an event type level interaction. When using a miner that returns a Petri net, we first compute the direct succession between transitions by omitting the places and then identify event type level interactions.

An important remark is that different process discovery techniques return a different set of event type level interactions. The meaning of an event type level interactions identified also varies depending on the discovery miner chosen as $Miner(L)$. For example, the interactions (dependencies) between two event types identified by the alpha miner are absolute precedence, thus event type A always before event type B, and no B is found before A in the log. In contrast, the event type level interactions returned by the flexible heuristic miner have a different meaning, i.e. event type A is mainly before event type B, and B before A is much less frequent (or lower than the threshold) in the log.

ETLI Discovery by Using Defined Criteria Using an existing miner to identify interaction may not be feasible in all situations. In particular the miner might produce models of low quality on the given data, or the user might not be interested in all interactions, but only in a specific subset. For such situations, we introduce a different method for identifying event type level interactions based on a simple precedence of events and time intervals between directly succeeding events. This method allows to consider all possible interactions (including outliers) or filter interactions based on a well-defined criterion.

We define several criteria, and for each criterion, a suitable situation is given as an example to use this criterion. For all criteria, let L_S, L_T be two logs with event types $\mathcal{A}_S$

and $\mathcal{A}_T$ respectively, and $\mid \mathcal{I}(L_S, L_T) \mid > 0$. Further, assume a user selected subsets of event types with $\mathcal{A}_{subS} \subseteq \mathcal{A}_S, \mathcal{A}_{subT} \subseteq \mathcal{A}_T$. We would like to find a set X of event type level interactions between $\mathcal{A}_{subS}$ and $\mathcal{A}_{subT}$ such that one of the following criteria holds.

Definition 21 (Criterion *absolute precedence*). *For each* $(E_p, E_c) \in \mathcal{A}_{subS} \times \mathcal{A}_{subT} \cup \mathcal{A}_{subT} \times \mathcal{A}_{subS}$, $(E_p, E_c) \in X$ *is an ETLI according to* absolute precedence*, iff, for all* $(\sigma_s, \sigma_t) \in \mathcal{I}(L_S, L_T) : \mathcal{F}_{E_p}(\sigma_s) \leq_{\mathcal{T}} \mathcal{F}_{E_c}(\sigma_t)$

The criterion *absolute precedence* basically says that in every trace any event of type E_p happens before any event of type E_c, without any exceptions.

It is possible that there might be exceptions. For example, the event type *Created* of sales orders should always happened before the event type *Created* of the related deliveries, but the recorded data may have been entered manually, and therefore, some deliveries are created before the creation of orders (and other events could happened between). To be able to detect both the "majority flow" and the "minority flow", we define the *existing precedence* criterion which says that there exists a trace where E_p is preceded by E_c.

Definition 22 (Criterion *existing precedence*). *For each* $(E_p, E_c) \in \mathcal{A}_{subS} \times \mathcal{A}_{subT} \cup \mathcal{A}_{subT} \times \mathcal{A}_{subS}$, $(E_p, E_c) \in X$ *is an ETLI according to* existing precedence*, iff, there is* $(\sigma_s, \sigma_t) \in \mathcal{I}(L_S, L_T) : \mathcal{F}_{E_p}(\sigma_s) \leq_{\mathcal{T}} \mathcal{F}_{E_c}(\sigma_t)$

From a time perspective, we also consider event types that always happen shortly after each other, i.e. the average time duration between the events of the two event types are shortest. We call this the *shortest time* criterion.

Definition 23 (Criterion *shortest time*). $(E_p, E_c) \in X$ *is an ETLI according to* shortest time*, iff, (1)*

$$min_{(E_p,E_c)\in\mathcal{A}_{subS}\times\mathcal{A}_{subT}}\left(\frac{\sum_{(\sigma_s,\sigma_t)\in\mathcal{I}(L_S,L_T)} AvgTimeDur(\mathcal{F}_{E_p}(\sigma_s),\mathcal{F}_{E_c}(\sigma_t))}{|\mathcal{I}(L_S,L_T)|}\right)$$

or (2)

$$min_{(E_p,E_c)\in\mathcal{A}_{subT}\times\mathcal{A}_{subS}}\left(\frac{\sum_{(\sigma_s,\sigma_t)\in\mathcal{I}(L_S,L_T)} AvgTimeDur(\mathcal{F}_{E_p}(\sigma_s),\mathcal{F}_{E_c}(\sigma_t))}{|\mathcal{I}(L_S,L_T)|}\right)$$

Note that in each pair of interacting logs, there is only one pair of events with a shortest time interval. Despite this limitation, we found this criterion in our case studies (see Section 7) to be very effective in identifying the main event type level interactions and hiding the complexities of interleaving relations between artifacts.

In addition, we define *event level interactions* using merged traces, based on which we then introduce the criterion *existence of an event level interaction*, which to some extent indicates the set of all possible *event type level interactions*.

Definition 24 (Event level interactions). *Let* $\sigma = \langle e_1, e_2, \cdots, e_n \rangle$ *be a trace. For all* $i = [1, ..., n-1]$*, we call event* e_{i+1} *the direct successor of* e_i*; we write* $(e_i, e_{i+1}) \in$

$succ(\sigma)$. *Let* $succ(L)$ *denote the direct successors of all traces in L. Now, if L is merged from two logs* L_S *and* L_T *(using the function described in Section 5.4), an event* $e_i \in L$ *indicates* $e_i \in \mathcal{E}_S \cup \mathcal{E}_T$ *which is the union set of events of* L_S *and* L_T*. We define an* event level interaction *as follows. A succession* $(e_i, e_{i+1}) \in succ(L)$ *with L merged from* L_S *and* L_T *is an* event level interaction *between the two events if and only if* $e_i \in \mathcal{E}_S$ *and* $e_j \in \mathcal{E}_T$ *or vice versa.*

The criterion *existence of an event level interactions* states if there exists an event level interaction (e_i, e_{i+1}) between two event types E_c and E_p, we consider (E_p, E_c) as an event type level interaction.

Definition 25 (Criterion *existence of an event level interaction*). $(E_p, E_c) \in X$ *is an ETLI according to* existence of an event level interaction*, iff, there exists* $(e_i, e_{i+1}) \in succ(L)$ *such that (1)* $e_i \in \mathcal{E}_S \wedge e_{i+1} \in \mathcal{E}_T$ *(or vice versa) and (2)* $\#_{type}(e_i) = E_p \wedge \#_{type}(e_{i+1}) = E_c$

The criterion *max number of event level interactions* explicitly count how often an event level interaction occurs in the merged log and treat the maximal interactions.

Definition 26 (Criterion *max number of event level interactions*). $(E_p, E_c) \in X$ *is an ETLI according to* max number of event level interactions*, iff, , iff,*

$$
\begin{aligned}
max_{\,(E_p,E_c)\in\mathcal{A}_{subS}\times\mathcal{A}_{subT}} \bigcup_{(\sigma_s,\sigma_t)\in\mathcal{I}(L_S,L_T)} (& \\
\{(e_i,e_{i+1}) \mid (e_i,e_{i+1}) \in \sigma \wedge \sigma \in \mathcal{M}(\sigma_s,\sigma_t) & \\
\wedge \#_{type}(e_i) = E_p \wedge \#_{type}(e_{i+1}) = E_c\}) & \\
\vee & \\
max_{\,(E_p,E_c)\in\mathcal{A}_{subT}\times\mathcal{A}_{subS}} \bigcup_{(\sigma_s,\sigma_t)\in\mathcal{I}(L_S,L_T)} (& \\
\{(e_i,e_{i+1}) \mid (e_i,e_{i+1}) \in \sigma \wedge \sigma \in \mathcal{M}(\sigma_s,\sigma_t) & \\
\wedge \#_{type}(e_i) = E_p \wedge \#_{type}(e_{i+1}) = E_c\}) &
\end{aligned}
$$

Identifying Unusal Interactions The two techniques introduced in Sections 5.4 and 5.4 are designed to identify the artifact interactions. However, a user might also be interested in particularly finding outliers, i.e., deviations from the main flow. Next, we present a corresponding technique based on the merged event logs of Section 5.4.

To identify outliers in the interaction, we first consider all event level interactions (see Definition 24), thus the directly succeeding events (e_i, e_{i+1}) in a merged log. Using these *event level interactions*, we can identify outliers as follows. On the merged log L, we first consider all pairs of event types (of $\mathcal{A}_S$ and $\mathcal{A}_T$) that directly follow each other in some trace of L. Then we identify the main interactions between $\mathcal{A}_S$ and $\mathcal{A}_T$ by applying a suitable discovery algorithm on the merged log (the technique of Section 5.4). Removing these main interactions from the set of all found direct successors yields the infrequent interactions. The formal definitions are as follows.

Let the set X_{all} denote the event type level interactions identified based on the *the existence of an event level interactions* criterion (which have included, to some extent, all possible type level interactions), and let X_{main} denote the event type level interactions that we identified by applying a miner on the merged log. Since we assume the miners identify the main flows, we consider $X_{all} \backslash X_{main}$ as the set of *unusual event type level interactions* (i.e. outliers).

5.5 Artifact-centric Model Discovery

We have shown two different methods to compute a set $X \subseteq \mathcal{A}_S \times \mathcal{A}_T \cup \mathcal{A}_T \times \mathcal{A}_S$ of event type level interactions from two given event logs L_S and L_T. In this section, we discuss how to discover an artifact-centric model.

Proclet System Discovery Formally, given a set of event logs $\{L_1, \cdots, L_n\}$, let a set $\mathbb{M}$ be the set of life-cycles discovered, i.e. each $M_i \in \mathbb{M}$ is a (Petri net) model describes the life-cycle of event log L_i extracted for artifact A_i. Given two life-cycles $M_S, M_T \in \mathbb{M}$, we can use a mapping function $\mathcal{X} : (\mathbb{M} \times \mathbb{M}) \rightarrow \mathcal{A} \times \mathcal{A}$ to return a set X of event type level interactions between the two life-cycles (using our techniques in Sections 5.4 and 5.4), i.e. $X \subseteq \mathcal{A}_S \times \mathcal{A}_T \cup \mathcal{A}_T \times \mathcal{A}_S$. In addition, for each model M_i, we have a function $AT_i : \mathcal{A}_i \rightarrow T_{iv}$ that maps an event type to the corresponding labeled transition in the model. To express the identified interactions between these life-cycle models, we turn each model into a proclet and add ports and channels between transitions that describe interacting event types.

We define the constraint that each port is connect to one transition, and each transition is only connect to one input port and one output port. This constraint is due to the fact that when a transition is connected to two output ports, there is no explicit expressiveness (identifiable) to distinguish whether a message is sent via both output ports (AND-splits), or it is only sent via one output port depending on a condition.

First, we create ports for each model $M_S \in \mathbb{M}$ as follows. Given a model M_S representing the life cycle of artifact A_S, let $X_S = \bigcup_{M_i \in \mathbb{M}} \mathcal{X}(M_S, M_i)$ denote all interactions that A_S has with all other artifacts. For each event type E_{out} of the artifact A_S if there is an event type level interaction $(E_{out}, E_i) \in X_S$ and $(E_{out}, E_i) \in \mathcal{A}_S \times \mathcal{A}_i$, we create an output port for the labeled transition $AT_s(E_{out})$ in model M_s. For each event types E_{in} of M_s, if there is an event type level interaction $(E_i, E_{in}) \in X_s$ and $(E_i, E_{in}) \in \mathcal{A}_i \times \mathcal{A}_s$, we create an input port for the transition $AT_s(E_{in})$ in model net M_s.

Finally, we connect the output port of transition $AT(E_i)$ to the input port of transition $AT(E_j)$ for each distinct type level interaction $(E_i, E_j) \in \mathbb{X} = \bigcup_{M_i \in \mathbb{M}, M_j \in \mathbb{M}} \mathcal{X}(M_i, M_j)$.

Simple Representation To be able to support business users by using our approach, interviews have be conducted within KPMG to investigate the requirements of clients for a suitable visual representation. The result of the investigation indicates that the proclet notation might be too formal to be communicated with a non-technical process stakeholder. One of the customers involved in the case study also indicated that he is less interested in the notions of places, tokens, silent transitions, AND-splits and OR-splits and finds the sequential relations containing adequate information. Therefore, we

developed a simpler notation to visualize the life-cycles and their interactions, which is similar to a simple dependency graph. We obtained this simpler representation by omitting the places of the Petri net and the ports of the system and directly connecting the transitions with their successors. Examples are shown in Figures 25 and 3.

Complexity Analysis In this section, we provide a complexity analysis of our approach. The running time of algorithms are summarized in Table 2. We use the same number to refer to the same step in the overview shown in Figure 4.

In the following analysis, we use $\mid \mathbb{T} \mid$ to denote the number of tables, $\mid \mathbb{C} \mid$ to denote the number of columns, $\mid \mathbb{F} \mid$ the number of references, $\mid \mathbb{A} \mid$ the number of artifact types, $\mid \mathbb{E} \mid$ the number of event types, and $\mid L \mid$ to denote the size of log in term of number of events.

The database schema identification (1.0), including primary key and foreign key extraction, is an NP-hard problem. Our approach and tools support both importing the existing data schemas, which takes $O(\mid \mathbb{T} \mid + \mid \mathbb{F} \mid)$, as well as using the original XTract approach to discover data schemas, which is exponential in number of columns.

The artifact schema identification (1.1) runs in linear with respect to the number of tables or the number of references. The artifact identification (1.2), which follows, runs in worst case $O(\mid \mathbb{A} \mid \times \mid \mathbb{C} \mid)$ because for each artifact type the algorithm has to include all columns (as identifiers, event types, or attributes of the artifact type). The extraction of a log of a defined artifact type (1.3) currently takes quadratic in terms of the number of entries in the data set, in worst case, since each entry in the main table is joined with all other entries to obtain its events and attributes. In theory, this running time can be improved to be linear in terms of the number of entries. Finally, the complexity of the discovery of a life-cycle of an artifact (1.4) depends on the discovery algorithm selected, which might be linear or exponential in terms of the number of events of the log for the artifact.

For discovering interactions between artifact types (2.1), the algorithm basically follows a depth-first-search to select all paths composed of references of which the number of strong joins is at most k and the number of weak joins is at most m, and therefore, grows exponentially in $(k + m)$. The interactions between artifact types are then used to extract interactions between artifact instances for logs (2.2), which takes quadratic in terms of the number of entries in the data set and is executed during step 1.3. Discovering interactions between event types (2.3) of two interacting artifacts requires to merge their logs, which takes $O(\mid L \mid^2)$, to run a discovery algorithm. To discovery an artifact-centric model, step 2.3 is re-run for every two interacting artifacts, and between each two event types, thus $O(\mid \mathbb{E} \mid^2)$, if they interact, we add an event type level interaction.

During the case studies, which is discussed in Section 7, steps (1.3) combined with (2.2) are the most time-consuming part; to extract about 30000 traces, in total circa 30000 events, it takes almost a hour. Other steps executed during the two case studies take less than ten minutes[4].

[4] We import the data schema, and for step (2.1) we use $k = 2$ and $m = 1$

Table 2: Running time analysis

Step	Running time
1.0	NP-hard or $O(\mid \mathbb{T} \mid + \mid \mathbb{F} \mid)$
1.1	$O(\mid \mathbb{T} \mid + \mid \mathbb{F} \mid)$
1.2	$O(\mid \mathbb{A} \mid \times \mid \mathbb{C} \mid)$
1.3	$O(\mid \textit{Entries} \mid^2)$
1.4	running time on the discovery algorithm selected
2.1	$O(\mid \mathbb{A} \mid \times \mid \mathbb{F} \mid^{k+m})$
2.2	$O(\mid \textit{Entries} \mid^2)$
2.3	$O(\mid L \mid^2 +$ running time of discovery algorithm)
2.4	$O(\mid \mathbb{E} \mid^2)$

6 Artifact-Centric Process Mining Methodology

In this section, we explain our methodology for conducting an artifact-centric process analysis project, which we also employed during our case studies. Figure 21 shows the methodology and indicates which parts of the methodology is supported by our approach and implementation.

An artifact-centric process analysis project starts with selecting a data source to be analyzed, and users can import the data source and data schema using our approach; XTract [18] can be used to automatically discover an unknown data schema. After importing the data source, users can discover, create and modify artifact schemas, their artifacts and interactions between them based on the methods described in Sections 4.3, 4.4 and 5.2. Once the artifacts and the interactions between them are specified, our techniques can automatically extract an event log for each artifact, as shown in Sections 4.5 and 5.3. The logs are used to discover an artifact-centric process model that shows the life-cycle of each artifact and the interactions between the artifacts during their life-cycle, which are discussed in Sections 4.7, 5.4 and 5.5. Domain experts can use this model to analyze and evaluate the business processes in its context and refine the artifacts, their event types and their interactions if desired. It is common practice in process analysis to refine the analysis and data extraction in several iterations.

During the case study, we obtained some best practices with respect to constructing and evaluating an artifact-centric model. To start with, it is much easier to **first create an overview of simple artifacts, each of which has only one or two event types (e.g. only *Created*), and with no more than two interactions between any two artifacts**, to help both analysts and clients to start the analysis and to understand the main flow of the process. Moreover, such a simple overview can be used to communicate between stakeholders, to further elaborate on requirements and questions, and to find more fine-grained or more complex artifacts. For example, in Section 2.3, the example of the artifact-centric model shown in Figure 3 is much easier to understand than the other three. This observation is also verified later during the case study for the Oracle Project Administration process when we tried to communicate with the clients for the first time.

Generally, with respect to evaluating artifacts, we suggest to **have simple artifacts and to include only one-to-one relations within artifacts**, with one exception: if the entries in the table (to which the artifact's main table is related via a one-to-many relation) can not be considered as instances of any other artifacts but purely denoting events.

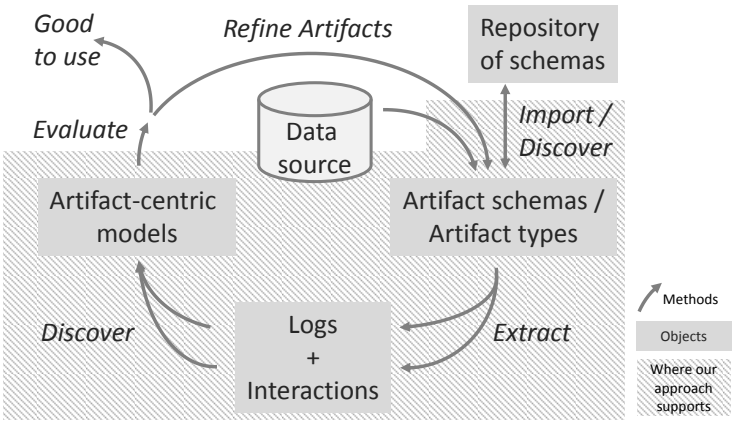

Fig. 21: The methodology used for conducting artifact-centric process analysis projects

For example, the change tables in SAP (CDHDR and CDPOS) only contain events that describe the changes of artifact instances, then it is harmless to include them as events of the artifact. Otherwise, it is better to consider the table as a separate artifact.

With respect to discovering and evaluating interactions between artifacts, we discuss the following cases which we learned during our Oracle case study (discussed in Section 7.4). When the artifacts in a model are clear and simple (e.g. less than two events), one can use the *merged traces* method to use discovery algorithm to discover the main event type level interactions. However, when the complexity of artifacts increases, the interactions discovered using this method are too complex to start with business users. In this situation, it is better to start with one of the criteria that only result in at most two interactions between artifacts (e.g. Definition 23 criterion *shortest time*). After the main flow of the model is understood, analysts can use *discovering all interactions and outliers* to inspect infrequent or abnormal flows individually.

To refine or evaluate artifacts and their interactions, **if an artifact appears to be interesting, one can start to unfold / extend the artifact by adding more tables and include more event types**. These event types are then also taken into account when discovering interactions. An example of extending simple artifacts is discussed using the SAP case study in Section 7.3. As a general rule, we found that **two event types should only be included in the same artifact type if one wants to compare the order of their occurrence on the same time-line**, for example, if we want to known when deliveries and return orders happened during the life-cycle of a sale-order. Generally, the **definition and refinement of artifact types should orient along the intended conceptual business objects**; a domain expert should validate the scope of the extracted business objects, a data expert can help in refining the artifact definition in order to retrieve the intended schema from the relational data source as explained in Section 4. To summarize, we suggest to start analyses with simple, clear artifacts and interactions and then gradually extend the artifacts and include more interactions while focus on the interesting observations made using discovered artifact-centric models.

7 Empirical Evaluation

We implemented the techniques presented in Sections 4 and 5. In this section, we briefly describe our implementation and then describe two real-life case studies where this tool was used.

7.1 Prototype Implementation

Our implementation consists of two parts. A standalone tool which is an extension of XTract [18] and implements the techniques described in Sections 4 and 5 for discovering artifact types and interactions between the artifact types, and for extracting life-cycle logs that are enriched with information about interactions between artifact instances. The tool reads as input the contents of a relational database. Then the user can import or automatically discover artifact schemas; for each schema the user can specify various artifact types choosing which events and attributes to include. Similarly, all valid artifact interactions are computed and the user selects the interactions of interest. From this selection one event log per artifact type is extracted containing the artifact's life-cycle and its interactions to other extracted artifacts.

The second part of our implementation is a plugin to the Process Mining toolkit ProM[5]. The plugin takes as input a set of extracted life-cycle logs. Then, a life-cycle model is discovered for each artifact and the user can choose which method to use for discovering artifact interactions. The resulting life-cycle models with interactions are shown graphically to the user on a screen where she has various filtering options available, for instance to highlight outliers. See [51] for details.

7.2 Case Study Design

The goal of the case studies was to evaluate the feasibility and practicability of our approach in a real-life setting. More specifically, we wanted (1) to verify whether our approach can indeed discover an artifact-centric process model including interactions in a practical setting as a proof-of-concept and (2) avoid false positive flows that showed up in earlier approaches because of convergence and divergence (see Section 2.2). Furthermore, we wanted to evaluate whether (3) the discovered model and interactions can provide accurate data- and facts-based insights of the given data source helping a business analyst to analyze the process and (4) whether the discovered models can be used to communicate with and understood by process stake holders with ease.

The first case study was performed in the Order to Cash (OTC) process of SAP. In this case study, we emphasized on the execution of our approach such as decisions about the artifact selection and interaction selection since no customer was involved. The second case study was conducted in the Project Administration (PA) process of Oracle. Since real customers were involved, we emphasize the discussion of the result of analyses.

For each case study, we (1) introduce the process and the data source, (2) explain the execution of our approach including decisions made, and (3) discuss observations and

[5] `www.promtools.org`

results we have obtained. In addition, during the first case study, we compare our artifact discovery to the one used by the original XTract approach, and we show some statistics about the false-positive flows with respect to a classical log conversion approach.

7.3 Case I - SAP Order To Cash Process

The first case study was performed for the Order to Cash (OTC) process supported by SAP systems. The data has been provided by KPMG. The main goal of this case study was to evaluate the technical abilities of our technique. Thus, the authors with an affiliation to KPMG, who are the experts in ERP systems and data analytic and have rich experience of conducting advisory projects for clients, provided the requirements and evaluated the results based on their experiences in earlier projects.

SAP OTC process and data structure A default OTC process in SAP starts with creating a sales order. After the order is delivered, a delivery document is created. Then, an invoice document is created in the system, sent to customer and posted in the account receivable (table). After receiving the payment, this default OTC process ends. However, there are many complex variations of this process. For example, the orders could be linked to a contract document, or return orders could be placed and return deliveries are made. Credit memo requests might be received from customers when the goods are incomplete or damaged. Invoices might also be canceled.

The data structure used to support the OTC process allows flexibility to deal with the aforementioned variance, but it is also very complex. The relational data structure of SAP regarding the relevant tables of this case study is discussed in [51].

SAP OTC - Extraction and Discovery In this section, we emphasize the steps taken and decisions made to obtain an artifact centric model for the SAP - OTC process.

Table Name	Constraint and Time scope	Row Count used	Column Count
BKPF	where '2012-09-01' <= cpudt and cpudt < '2012-11-01' and awtyp = 'vbrk'	11358	32
BSID	where '2012-09-01' <= cpudt and cpudt < '2012-11-01'	4428	49
BSAD	where '2012-09-01' <= cpudt and cpudt < '2012-11-01'	911	49
CDHDR	where '2012-09-01' <= [UDATE] and [UDATE] < '2012-11-01' and (cdhdr.objectclas = 'VERKBELEG' or cdhdr.objectclas = 'faktBELEG')	13903	9
CDPOS	inner join cdhdr on cdhdr.changenr = cdpos.changenr where '2012-09-01' <= [UDATE] and [UDATE] < '2012-11-01' and (cdhdr.objectclas = 'VERKBELEG' or cdhdr.objectclas = 'faktBELEG')	56018	10
DDFTX	where tabname = 'vbak' or tabname = 'vbrk'	237	5
VBAK	where '2012-09-01' <= erdat and erdat < '2012-11-01'	3383	40
VBAP	where '2012-09-01' <= erdat and erdat < '2012-11-01'	4317	35
VBRK	where '2012-09-01' <= erdat and erdat < '2012-11-01'	5285	37
VBRP	where '2012-09-01' <= erdat and erdat < '2012-11-01'	10206	31
LIKP	where '2012-09-01' <= erdat and erdat < '2012-11-01'	11623	51
LIPS	where '2012-09-01' <= erdat and erdat < '2012-11-01'	13157	15

Fig. 22: SAP OTC process - tables and record counts

First, we imported 11 tables shown in Figure 22 in which the name of the tables, the constraint for selective import, the number of records used, and the number of columns are described. We considered only the documents created between '01-09-2012' and

'31-10-2012'. As some time columns in the OTC process data were not recorded, we only considered 'date-stamps'. Primary keys and foreign keys where imported based on domain knowledge [51].

Second, we observed in the data that documents (e.g. a sales order) and lines (e.g. line items in a sales order) are stored separately in different tables. Moreover, the line-level objects have their own life-cycles which could be independent of the related document objects. For example, a line item might be rejected but the sales order is still pursued. As the line objects have many-to-one relation with their related document objects (e.g. a sales order can have multiple sales lines), documents and lines were split into separated artifact schemas (automatically done by the algorithm in Section 4.3), leading to 8 artifact schemas in total. For the sales documents (i.e. table *VBAK*) and lines (i.e. table *VBAP*), delivery documents (i.e. table *LIKP*) and lines (i.e. table *LIPS*), and invoice documents (i.e. table *VBRK*) and lines (i.e. table *VBRP*), there is a column *vbtyp* in the document tables that indicate the type of documents. Therefore, we created the artifacts of documents and lines based on the values found in this column. In total, we identified 35 artifacts. Figure 23 shows some example of the artifact schemas and the artifacts we identified. The complete list can be found in [51].

Art.Schema	Maintable	Artifact	Artifact condition	Extracted
BKPF	BKPF	PostInAR	<maintable>.awtyp = 'VBRK'	Yes
BSAD	BSAD	Payment05or15	(<maintable>.bschl = '05' or <maintable>.bschl = '15')	Yes
		Payment01or11	(<maintable>.bschl = '01' or <maintable>.bschl = '11')	
VBAK	VBAK	Order H	<maintable>.vbtyp = 'C'	Yes
VBAP	VBAP	Order L	inner join tableVBAK s4 on <maintable>.vbeln = s4.vbeln and s4.vbtyp = 'C'	

Fig. 23: SAP OTC process - artifacts

To create a high-level overview of the OTC process, our business analysts suggest to only consider the document level artifacts for now and omit the line level artifacts. Therefore, we identify all possible type level interactions between artifacts (Figure 24 shows a screen shot of the interaction graph that is constructed) and selected the (direct and indirect) interactions between document level artifacts for the data extraction, as follows. The *predecessor relation* which indicates the transformation (or causality) between objects (indicated by the foreign key relation *vgbel* and *vgpos* columns) is very interesting. However, the *predecessor relation* is mainly between lines, whereas we would like to identify interactions between the documents. To obtained the predecessor relation between document artifacts, the indirect interactions that join a document artifact with its line artifact, and then join the line artifact with its predecessor line artifact (preceding lines), and finally with the (preceding) document artifact to which the preceding line artifact is related (i.e. the number of strong joins $k = 2$, the number of weak joins $m = 1$) are required. Whenever this key relation contained a record, we select the indirect interactions between documents based on this key relation. Furthermore, as

one of our goal was to identify outliers, we included any interaction between document artifacts with at least one record (i.e. the least number of instance interaction $r = 1$).

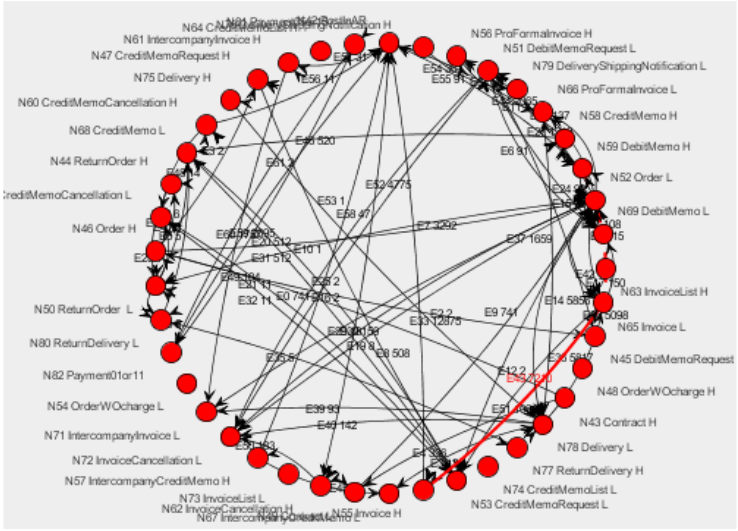

Fig. 24: SAP OTC process - interaction graph

For each artifact shown in Figure 25, we obtain an *event log* with trace level interaction. In total, 18 event logs are extracted and imported into ProM. Using the heuristic miner as the life-cycle miner and as the miner for discovering interactions on the merged logs, we obtain the artifact-centric model. Filtering the life-cycle logs to contain only *Creation* events prior to discovery allowed us to first obtain an overview on the process that is shown in Figure 25.

SAP OTC - Process Analyses and Discussion In this section, we discuss three observations: identifying unusual flows, drilling down from an overview to details by creating complex artifacts and the complexity of interactions.

Identifying Unusual Flows. As the high-level model of Figure 25 only contained the dominant artifact interactions, we set on to identify unusual flows using the *existence of an event level interactions* criterion (described in Section 5.4). The red arc in Figure 26 shows the *unusual event type level interactions* which were revealed by using this criterion. This unusual flow from *Payments Received* to *Invoice Created* indicates that there were payments received before the corresponding invoices were created in the account receivable.

We verified the existence of this unusual flow in the database (the SQL query used can be found in [51]) and found that for the cases that caused this flow, the database indeed contained a *Payment Received* date earlier than the *Posted In AR* date. This usual flow indicates that a manual change was made to the payment record, or a payment was made before the corresponding customer invoice was recorded in the system. The latter case can occur if payment was recorded through bank statement transfer, and

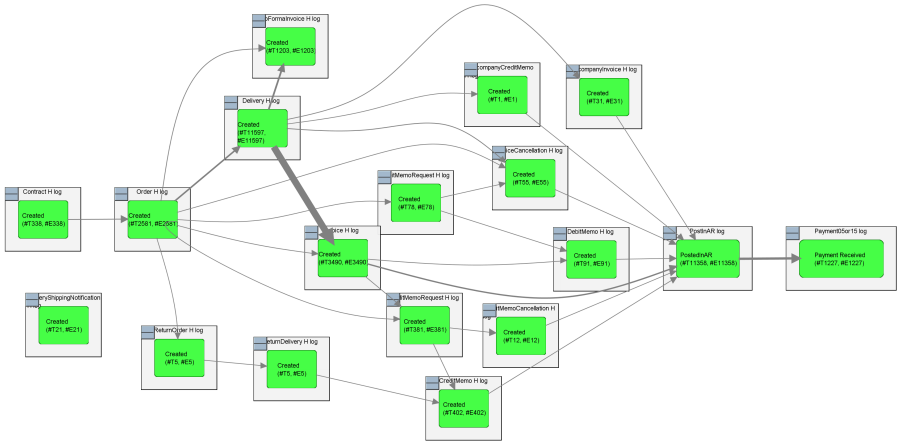

Fig. 25: SAP OTC process - artifact-centric model with simple representation

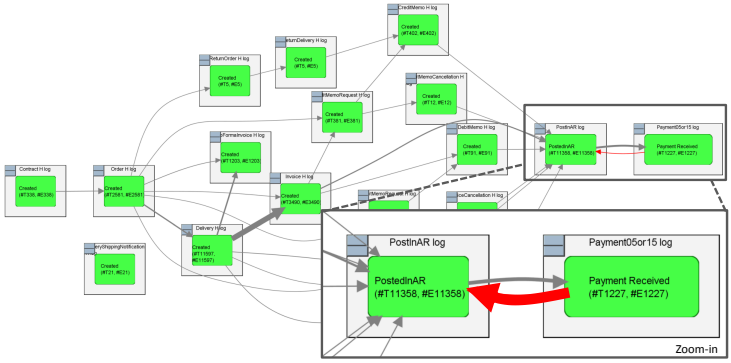

Fig. 26: SAP OTC process - artifact-centric model with outliers

the system was unable to apply the cash to an invoice. Later, the invoice is posted and manually matched to the already received payment. Both cases indicate a deviation from a reference sales process and/or untimely recording of customer liabilities. Further investigation revealed that the cases was (indeed) manually changed by someone using the transaction code FB05, which is used to apply for cash (automated or manually) incurring risks. In other words, we have successfully and exploratively discovered a true-positive unusual action.

Creating Complex artifacts. After this initial analysis on the basic relations of the creation of documents in the process, we conducted a second analysis to consider the life-cycle of the *Sales Order* artifact in more detail. For this, we returned to the data extraction and now included in the *Sales Order* artifact type all event types found in the change tables *CDHDR* and *CDPOS*; see [51] for details. Since the trace identifiers have not changed, and interactions have not changed, there is no need to re-extract other artifacts. Using the *max number of event level interactions* criterion, we obtain the artifact centric model shown in Figure 28.

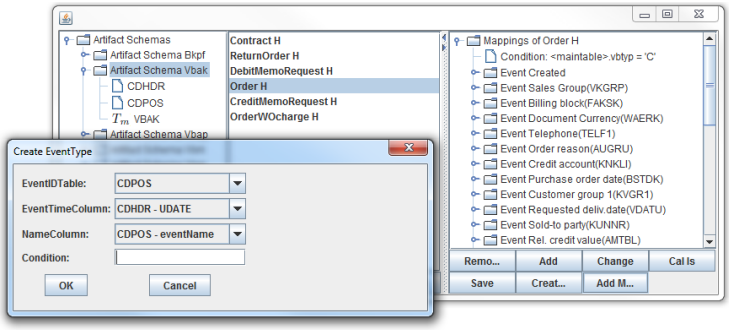

Fig. 27: SAP OTC process - the Orders artifact with change event types

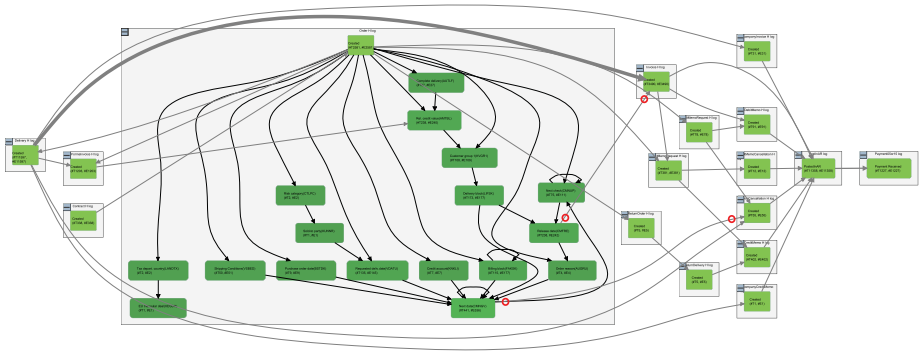

Fig. 28: SAP OTC process - the same artifact-centric model as Figure 25 except the life-cycle of artifact *Sales order* is extended

Complexity of Interactions. Note that we can now identify clear difference in event type level interactions between the sales order artifact and other artifacts. The red circles indicate an interesting difference. For example, most artifacts, e.g. *Delivery*, *Credit Memo Request*, *Debit Memo Request* are generally created directly after the creation of *Sales Order*, whereas the invoices (that are directly related to a sales order via *vgbel* and *vgpos*) are created after the *Release Date* change event type of this sales order. Similar for the invoice cancellation artifact, of which the creation generally takes place after the *Next Date* of the sales order changes.

At this point, while the results show that the first three goals (see Section 7.2) have been achieved, we also noted a limitation of our technique. Showing all interactions discovered on the merged log makes the artifact-centric model very complex and almost impossible to analyze, further simplification or filtering techniques have to be developed.

Comparing to Existing Approaches In this section, we compare our approach to a classical log conversion approach and the original XTract approach using the data set from the SAP case study.

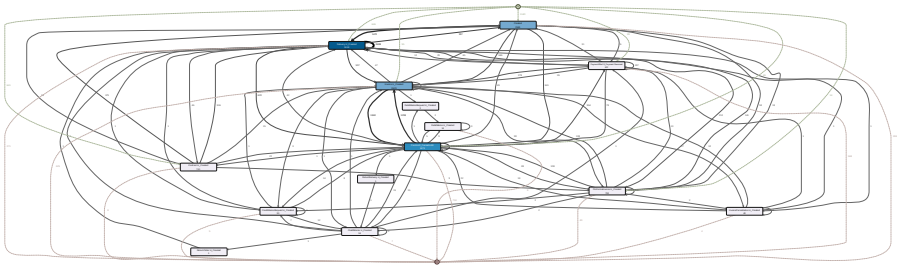

Fig. 29: SAP OTC process case study - a life-cycle of sales orders obtained using a classical log conversion approach.

Comparing To Classical Conversion Approach - False Positive Flows. To show the improvements of artifact-centric approach with respect to reducing the number of false positive "abnormal flows", we defined the artifact *Sales Order* that includes all the selected event types shown in Figure 26. In other words, we use the sales orders as our case notion and merged all events related to a sales order to the trace of this sales order. More specifically, *if an event e is indirectly related to a sales order via multiple other artifacts or events, the event e is only added once to the trace of this sales order.* The artifacts of which no event is related to any sales order is neglected. The events with the same time-stamps are sorted randomly.

The directly-follows-graph of the sales-order-oriented event log is shown in Figure 29. While created based on the same data source and with less events (because many events are not related to any sales orders), the life-cycle model shown is much more complex due to data divergence and convergence problems. We observed that 9 out of the 14 event types have a self-loop, e.g. *Delivery H_Created*, *Invoice H_Created*, *PostInAR_PostedInAR*, which violates the artifact-centric model obtained and the fact that no event in the original data set is directly related to another event of the same type.

We count the absolute number of directly follows relations between event types and summarize them in Figure 30. In total, we found that 6696 out of 13644 directly follows relations *directly or transitively* violate the flows shown by the artifact-centric model in Figure 25, which is about 50%. 36 out of the 79 number of arcs found in Figure 29, each of which indicates a causality dependency between the source and the target event types, violates the causality dependencies indicated by the artifact-centric model. Thus, almost 50% of the dependencies, between events and between event types, are false.

Besides causing these false positive "abnormal flows", the statistics related to the number of events are also polluted. A clear example that shows the data convergence problem is the 119% increase in the number of contracts, i.e. from 338 (shown by the original data set and the artifact-centric models) to 741 (shown by the directly-follows graph in Figure 29). The contract created events are duplicated because a contract can be related to multiple sales orders.

Comparing to Artifact Discovery of the original XTract Approach. The original XTract approach uses k-means clustering approach to automatically discover a set of artifact types if the number of artifact types is given. We import the keys from our

n: violating / n: non-violating / n: aligning		C	O	D	RO	RD	I	DR	DM	IC	CR	CM	PI	AR	P	Total
Contract H_Created	C	0	371	316	0	0	15	0	0	0	0	0	9	22	0	
(Order) Created	O	53	0	1472	0	0	346	0	0	0	0	0	59	341	6	
Delivery H_Created	D	46	907	2439	0	0	597	0	0	1	1	1	212	620	23	
ReturnOrder H_Created	RO	0	0	1	0	0	0	0	0	0	0	0	0	0	0	
ReturnDelivery H_Created	RD	0	0	0	0	0	0	0	0	0	0	1	0	0	0	
Invoice H_Created	I	0	65	47	0	0	142	1	0	17	3	1	90	1360	278	
DebitMemoRequest H_Created	DR	0	0	0	0	0	0	0	2	0	0	0	0	0	0	
DebitMemo H_Created	DM	0	0	0	0	0	0	0	3	0	0	0	0	5	0	
InvoiceCancellation H_Created	IC	0	0	1	0	0	10	0	0	2	0	0	0	22	1	
CreditMemoRequest H_Created	CR	0	0	1	0	0	1	0	0	0	6	12	0	8	0	
CreditMemo H_Created	CM	0	0	0	1	0	1	0	0	0	7	2	0	15	1	
ProFormaInvoice H_Created	PI	0	18	141	0	0	138	0	0	2	2	2	257	108	13	
PostInAR_PostedInAR	AR	1	49	42	0	1	1298	1	9	18	14	15	89	854	354	
Payment05or15_Payment Received	P	0	31	12	0	0	26	0	0	0	0	0	0	78	107	
Sum total number of edges		100	1441	4472	1	1	2574	2	14	40	33	34	716	3433	783	13644
Sum the number of "False" edge		100	1070	2683	1	1	1468	1	12	20	27	17	257	932	107	6696
Relative False Edges w.r.t. Total in %		100%	74%	60%	100%	100%	57%	50%	86%	50%	82%	50%	36%	27%	14%	49%

Fig. 30: Statistics w.r.t. directly-follows relations discovered in the sales-order-oriented log

repository because the XTract approach was unable to discover the primary keys and foreign keys of SAP tables.

The artifact schemas returned by the XTract approach are listed in Table 3 when given k as the number of artifact types. Each artifact schema is converted into an artifact. For instance, when k is 2, tables from *VBAP* to BSAD are returned as one artifact type (C2) with main table *VBRP*; when k is 5, tables *VBAK*, *VBAP*, *LIPS* and *LIKP* are returned as one artifact type (C5) with the main table *VBAP*.

As shown by Table 3, when the number k of artifacts is chosen well, e.g. between 7 and 9, the resulting artifact types are simple and reasonable, each of which only includes one or two one-to-many relations within an artifact, thus very similar to the ones returned by our approach. However, there are three limitations shown by the table. First, when the number k of artifacts is small (e.g. $2 \leq k \leq 6$), the k-means clustering algorithm neglects the divergence and convergence problem and places many tables in the same cluster (e.g. clusters C2 and C5). These clusters result in complex and difficult-to-understand life-cycles such as the one shown by Figure 29. Second, when the number k of artifacts is large (e.g. ≥ 10), empty clusters are returned (e.g. $C10$). Third, the tables of each of these clusters can only be completely mapped to an artifact; no subset of the tables can be considered as an artifact. This limitation obstructs us from identifying, for example, different document types as different artifact types.

In comparison to our artifact discovery, we identify artifact schemas automatically, no need to choose a k for the number of artifacts. The artifact schemas can also be automatically mapped to one artifact. However, manual inputs are needed to discover more fine-grained artifacts.

7.4 Case II - Oracle Project Administration Process

The second case study is performed for the project administration (PA) process supported by the Oracle information system of an educational organization. This case study is performed on request of an educational organization to who we refer as the client. For this case study, we first downloaded a dataset from the client's information system, then analyzed the data and provided the client with feedback about our findings.

Table 3: Artifact schemas obtained using the original XTract approach.

k	2	3	4	5	6	7	8	9	10
CDPOS	C1	C1	C1	C1	C1	C1	C1	C1	C1
CDHDR			C2	C2	C2	C2	C2	C2	C2
VBAK							C8	C8	C8
VBAP		C2		C5	C5	C5	C5	C5	C5
LIPS									
LIKP						C7	C7	C7	C7
VBRK					C6	C6	C6	C6	C6
VBRP	C2		C3	C3	C3	C3	C3	C3	C3
BKPF		C3	C4	C4	C4	C4	C4	C4	C4
BSAD								C9	C9
no table									C10

Oracle PA Process and Data source An educational organization has thousands of projects running, e.g. different research projects. The project administration process supported by Oracle starts with creating projects in the system. At the moment of creating a project in the system, it is usually definitive that the project will be executed. It is possible that the project has already started before it is added to Oracle PA. After the project is created in the system, one can specify relevant information of the project such as its starting date. During the execution of the project, tasks are created for the project to declare different expenditures related to a task, e.g. personnel, materials. For assessing financial risks, it is important that the ending date of expenditures is before the the complete date of tasks. Moreover, all tasks should be completed before the completion date of the project. When the project is completed, it means that the main activities, such as the research itself, are finished. When the administration work is completed, such as financial checks, the project is closed.

For this case study, 18 tables were downloaded, and 7 tables were used in the actual analysis as shown in Figure 31. For our analysis, we considered data recorded between 01-06-2012 and 31-12-2012. The number of the records of each table used in the process analysis is also shown in the fourth column of Figure 31. Documentation about the data schema (esp. primary keys and foreign keys) was available.

Oracle PA - Extraction and Discovery In the chosen data, the table PA_PROJECT_STATUSES did not contain any information relation to process steps and were omitted from the analysis. Each of the remaining six tables was considered an artifact schema and also mapped to one artifact type, respectively. The six artifacts are shown in Figure 32. Seven direct type level interactions are found between the artifacts which are shown in Figure 33.

For the log extraction step, we considered only the artifacts *Projects*, *Tasks* and *Expenditures* as these are the primary objects in the process. The three event logs were imported into ProM. We confirmed that the number of traces (or cases) of an event log matched the number of the row count of the corresponding main table. The projects

Table Name	Row Count downloaded	Used	Row Count used	Column Count
FND_USER	905		905	27
HR_ALL_ORGANIZATION_UNITS	1053		1053	43
MTL_SYSTEM_ITEMS_B	0			
OE_ORDER_HEADERS_ALL	0			
PA_COST_DISTRIBUTION_LINES_ALL	95943	x	5543	15
PA_EXPENDITURE_COMMENTS	55149		30511	7
PA_EXPENDITURE_ITEMS_ALL	96978	x	5620	32
PA_EXPENDITURE_TYPES	94	x	94	10
PA_EXPENDITURES_ALL	682590	x	3100	24
PA_PROJECT_CUSTOMERS_V	16238		16238	17
PA_PROJECT_STATUSES	80	x	80	23
PA_PROJECTS_ALL	5364	x	1132	29
PA_TASKS	2416	x	1236	31
PA_TRANSACTION_SOURCES	48		48	4
PAY_COST_ALLOCATION_KEYFLEX	1694		1694	11
PO_HEADERS_ALL	5186		5186	139
PO_LINE_LOCATIONS_ALL	7700		7700	148
PO_LINES_ALL	7700		7700	135

Fig. 31: Oracle PA process table record counts

log, the task log and the ExpAll log have each in total 5329, 4383, and 9300 events, respectively.

For the artifact-centric process discovery, the heuristic miner was used as the life cycle miner and as the miner for discovering interactions on the merged logs. We obtain the following proclet model in simple representation shown in Figure 34. In a second run, we used the *max number of event level interactions* criterion (defined in Section 5.4) to identify the event type level interactions, to obtain a simpler model shown in Figure 35 which can be communicated with business users with more ease.

Process Analyses Result and Discussion Since we aim to perform process analysis using the artifact centric approach, the results obtained were discussed and validated internally as well as with the client. We found a set of unusual flows both within artifacts as well as in interactions between artifacts using the discovered artifact-centric model(s). We were able to retrieve the cases for each unusual flow we found and verified them in the original database. An interview with the client was conducted during which the unusual flows were discussed. Since the identification of unusual flows within artifacts are already covered by classical process mining techniques, we here focus on the unusual interactions and discuss two findings in detail. For more findings and discussion, we refer to [51]. In the following, we use the number between the parenthesis to refer to the corresponding finding annotated with the same number shown in the figures.

We observed cases where a task had been started before the related project had been created as shown in the model in Figure 36. In the interview, the client asked for the time duration between these two events. The client indicated that when creating a project in the system, the specification of all tasks related to this project are known, thus the creation time of tasks should happen shortly after the creation of the project in the system. If not, it may indicate that double administrative work have been performed.

Artifacts	Maintable	Extracted
Project	PA_PROJECTS_ALL	x
ExpAll	PA_EXPENDITURES_ALL	x
Tasks	PA_TASKS	x
CostDistr	PA_COST_DISTRIBUTION_LINES_ALL	
ExpItem	PA_EXPENDITURE_ITEMS_ALL	
ExpTypes	PA_EXPENDITURE_TYPES	

Interaction From	To	via
Project	Tasks	
Tasks	ExpAll	ExpItem

Fig. 32: The artifacts created for Oracle PA process

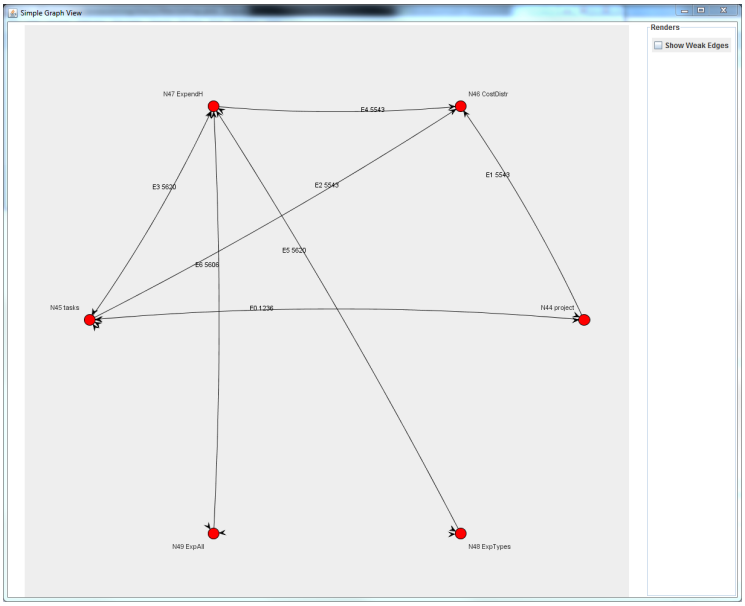

Fig. 33: Interaction graph of PA artifacts

Since the average time of different event type level interactions is calculated for the *shortest time* criterion defined in Section 5.4, we were able to retrieve this average time between the creation of project and tasks easily (see Figure 36 (5)), which is 1.088 day. As the average time might be an inaccurate indication, the client was asked to give a maximal threshold, which was less than or equal to two weeks. We had to verify this property manually in the database. Of the 1236 tasks, we found 1197 tasks that had been created less than a day after the project was created; 8 tasks that had been created between a day and 14 days after the project was created; and 31 tasks had been created later than 14 days after the project had started. We found no task had been created before the creation of its project (which show that our model has correctly shown the flows), see Figure 36 (5).

Another unexpected unusual observation that we made on the data was that there were many projects closed before the related tasks completed [51]. This ordering of events indicates the risk that expenditures can be booked on the tasks while the project is already closed. Therefore, the client asked us to further investigate whether there are expenditures created after the projects are closed or completed. For this analysis, we

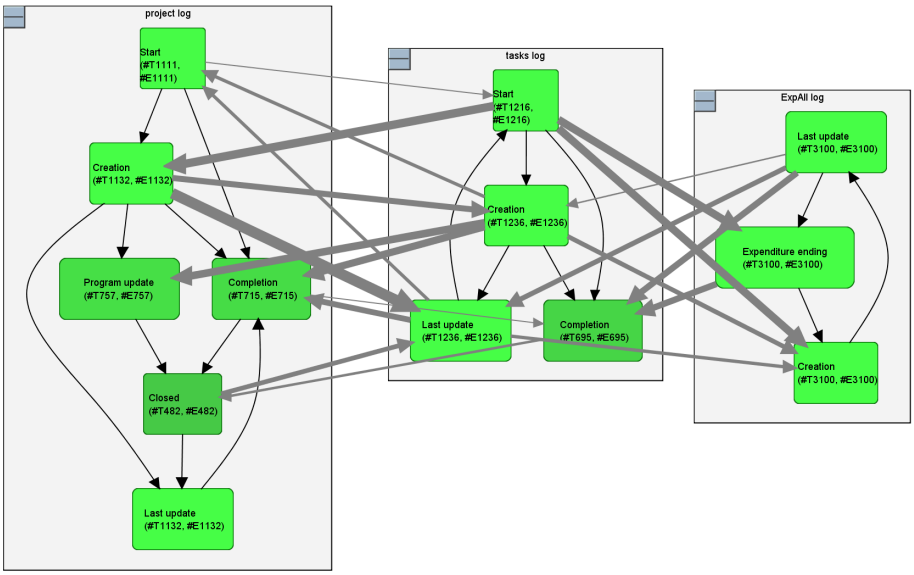

Fig. 34: A proclet system discovered using merging logs method

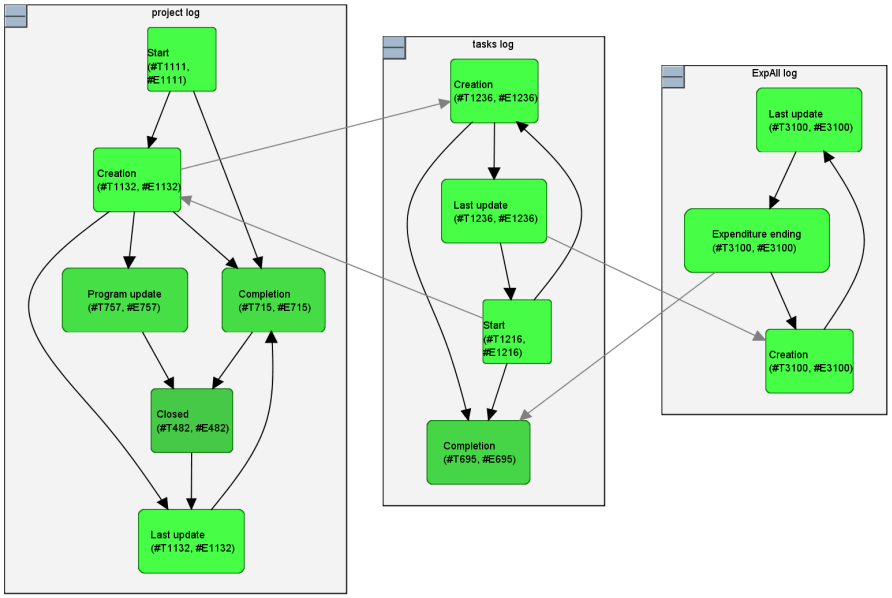

Fig. 35: A proclet system discovered by using the max number of event level interactions criterion

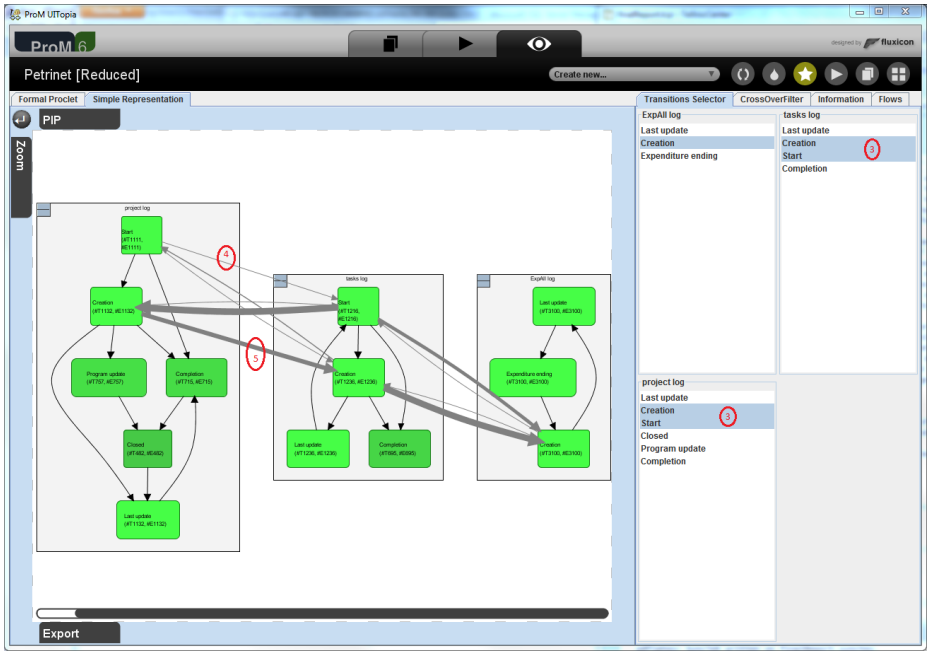

Fig. 36: A proclet system discovered by using the existing precedence criterion

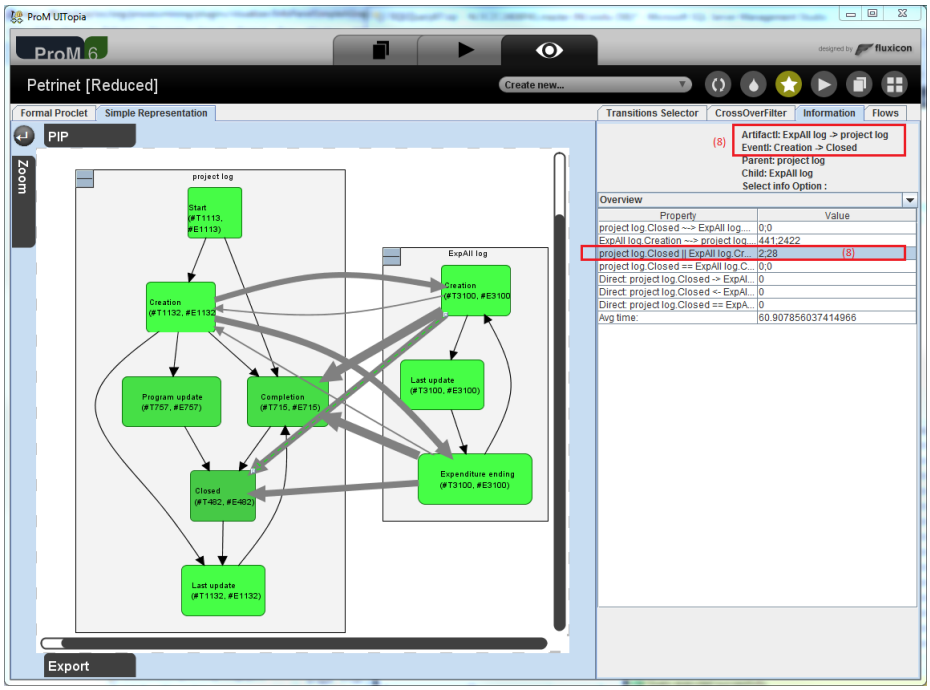

Fig. 37: Expenditures created after projects closed

extracted a new event log for the *Project* artifact enriched with the interaction to the *Expenditures* artifact. We now analyzed the *Project* artifact, the *Expenditures* artifact and their interactions; the resulting model is shown in Figure 37.

We found two projects had been created in parallel with the creation of the expenditures indicating that there are expenditures which are created after the two projects are closed (see Figure 37 (8)). Verifying this observation in the database, we retrieved that 5 out of the 28 expenditures related to these two projects were indeed created after the two projects were closed, shown in Figure 38. Two of the five expenditures are created longer than 24 hours after the closure of the projects; other three expenditures were created on the same day when the project are closed (which may due to the *CLOSED_DATE* events of the projects only have a 'date-stamp'). This result again shows that our approach is able to illustrate true positive unusual flows based on the recorded data.

PROJECT_ID	CREATION_DATE	CLOSED_DATE	COMPLETION_DATE	EXPENDITURE_ID	CREATION_DATE
6[redacted]9	2012-06-18 09:52:18.000	2012-08-30 00:00:00.000	2012-08-31 00:00:00.000	7[redacted]8	2012-08-31 18:11:56.000
6[redacted]9	2012-06-18 09:52:18.000	2012-08-30 00:00:00.000	2012-08-31 00:00:00.000	7[redacted]2	2012-08-31 18:11:58.000
6[redacted]5	2012-07-16 15:17:21.000	2012-09-19 00:00:00.000	2012-09-30 00:00:00.000	7[redacted]2	2012-09-19 18:15:24.000
6[redacted]5	2012-07-16 15:17:21.000	2012-09-19 00:00:00.000	2012-09-30 00:00:00.000	7[redacted]9	2012-09-19 18:15:16.000
6[redacted]5	2012-07-16 15:17:21.000	2012-09-19 00:00:00.000	2012-09-30 00:00:00.000	7[redacted]9	2012-09-19 18:15:24.000

Fig. 38: Expenditures created after projects closed in database

In addition, we found the following allowed unusual flows. We have found one task which is completed before the two related expenditures were ended. Consulting the Oracle website[6], we learned that the expenditures could end in the weekend of the same week that the task is completed, we verified in the original database that this was indeed the case; for detail see [51]. Furthermore, we found expenditures that had been created before the related task was created. To explain this observation, we drilled down in the data and split the Expenditure artifact schema into 8 different artifact types based on the expenditure category. The model shown in Figure 39 confirms the assumptions that each type of expenditure has different event type level interaction with the project life cycle. For example, the staff expenditures are rather created at the beginning of the project (see Figure 39 (9)), whereas the others expenditures are created after the project is definitive (in the system) or after the *Update Program* event type of the project (see Figure 39 (10)).

We also found some unusual flows within artifacts and discussed these with the client. For example, we found that a project is closed (i.e. financially complete) before the project is completed (i.e. research completed), which might indicate financial risks. In addition, we found a task was created after the task was completed.

Finally, the understandability of the model was discussed. The client indicated the process models shown in this paper were very hard to understand without further explanation. Especially, business users were used to static diagrams such as histograms, or pie charts. But after an interactive session of one hour showing and explaining the process models the client, who is an domain-expert on the analyzed process, could clearly

[6] `http://docs.oracle.com/cd/A60725_05/html/comnls/us/pa/dates06.htm`

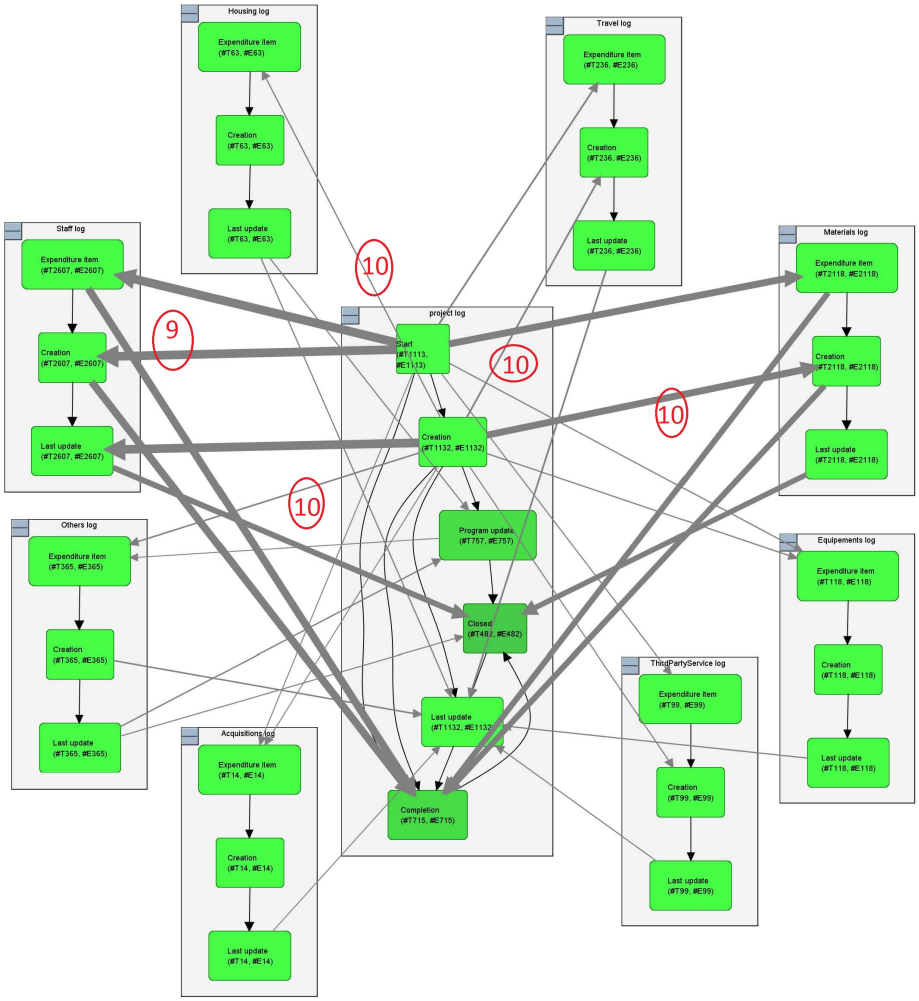

Fig. 39: Interactions between the eight expenditure artifacts and the project life cycle

understand the process model and join in the analysis. A good argument for this claim is that the client was able to observe the unusual flow from the *Closed* event type to the *Last Update* event type of the project which had not been observed by the authors of this paper before the interview. The client indicated that the number of cases are also important to help assess the impact of certain unusual flows.

We conclude this section by summarizing the the results of the two case studies. For both case studies, we are able to successfully create the desired artifacts and identify the desired type level interactions. We were able to discover artifact-centric models using the event logs that were extracted for the artifacts and enriched with interactions. Moreover, for both case studies, we were able to use the discovered model to identify true-positive unusual flows validated by ourselves and by client. While we could achieve our goals, we also noted some limitation. In some cases, domain knowledge might be required to distinguish the "allowed" unusual flows (e.g. expenditures are allowed to be closed after the related tasks are closed but in the same week) from the prohibited unusual flows identified.

8 Conclusion

In this paper, we addressed the problem of discovering a process model from event data stored in a relational data source. We proposed to discover a model that describes the process as a set of interacting data objects (of the process), each following its own life-cycle, also called artifacts. For this, we contributed a semi-automatic technique to identify artifact types in a relational data source, extract a life-cycle log for each identified type. From each log, a life-cycle model of this artifact can be identified using existing process discovery techniques. Second, we provide, for the first time, a family of technique to discover interactions between artifacts at the type level and the event level. This information can be used to visualize the interactions between the extracted artifact life-cycle models. We validated our approach in two case studies using real-life data from ERP systems and showed that the discovered models accurately described the executions of the recorded business processes. We could show that the discovered models provide useful insights into the processes and allowed to identify unusual flows of executions.

Future Research. This paper made a first step towards a fully discovery of artifact-centric process models from a relational data source. Currently, our approach for discovering artifacts still needs manual steps such as indicating a column for splitting the artifacts or splitting the event types. More advanced algorithms can be developed to identify the "perfect" artifact automatically by using, for example, metrics and heuristics. Furthermore, we considered the line level artifacts (e.g. sales order lines) as separate artifacts in our case study and omitted them from the log extraction. It would be interesting to investigate the hierarchy of artifacts; for example, supporting the discovery of sub-artifacts (e.g. sales order lines) within artifacts. A limitation of the current interaction discovery is that it is limited to two artifacts. We would like to discover the interaction flow between multiple artifacts by merging multiple artifacts for example.

Acknowledgments

We thank B.F. van Dongen and H.M.W. Verbeek for their substantial support in writing this paper. We also thank W. van Kessel (KPMG) for his substantial support in analyzing the Oracle case study.

References

1. W. v. d. Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
2. M. van Eck, X. Lu, S. Leemans, and W. van der Aalst, "Pm^2: a process mining project methodology," in *CAiSE 2015 (accepted)*, 2015.
3. M. v. Giessel, "Process Mining in SAP R/3: A method for applying process mining to SAP R/3," Master's thesis, Eindhoven University of Technology, 2004.
4. I. Segers, "Investigating the application of process mining for auditing purposes," Master's thesis, Eindhoven University of Technology, 2007.
5. J. Buijs, "Mapping data sources to xes in a generic way," Master's thesis, Eindhoven University of Technology, 2010.
6. D. Piessens, "Event Log Extraction from SAP ECC 6.0," Master's thesis, Eindhoven University of Technology, 2011.
7. A. Roest, "A Practitioners Guide Towards Process Mining on ERP Systems - Implemented and Tested for SAP Order to Cash," Master's thesis, Eindhoven University of Technology, 2012.
8. A. Nigam and N. Caswell, "Business artifacts: An approach to operational specification," *IBM Systems Journal*, vol. 42, no. 3, pp. 428–445, 2003.
9. D. Cohn and R. Hull, "Business artifacts: A data-centric approach to modeling business operations and processes," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 32, no. 3, pp. 3–9, 2009.
10. W. v. d. Aalst, A. Adriansyah, A. d. Medeiros, F. Arcieri, T. Baier, T. Blickle, J. Bose, P. v. d. Brand, R. Brandtjen, J. Buijs *et al.*, "Process Mining Manifesto," in *Business process management workshops*. Springer, 2012, pp. 169–194.
11. R. Hull, E. Damaggio, R. D. Masellis, F. Fournier, M. Gupta, F. T. Heath, S. Hobson, M. Linehan, S. Maradugu, A. Nigam, P. Sukaviriya, and R. Vaculn, "Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events." in *DEBS, ACM*, 2011, pp. 51–62.
12. R. J. Miller and P. Andritsos, "Schema discovery," *IEEE Data Eng. Bull.*, vol. 26, no. 3, pp. 40–45, 2003. [Online]. Available: http://sites.computer.org/debull/A03sept/toronto.ps
13. J. Turmo, A. Ageno, and N. Català, "Adaptive information extraction," *ACM Comput. Surv.*, vol. 38, no. 2, 2006. [Online]. Available: http://doi.acm.org/10.1145/1132956.1132957
14. S. Sarawagi, "Information extraction," *Foundations and Trends in Databases*, vol. 1, no. 3, pp. 261–377, 2008. [Online]. Available: http://dx.doi.org/10.1561/1900000003
15. V. M. Markowitz and J. A. Makowsky, "Identifying extended entity-relationship object structures in relational schemas," *IEEE Trans. Software Eng.*, vol. 16, no. 8, pp. 777–790, 1990. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/32.57618
16. R. H. L. Chiang, T. M. Barron, and V. C. Storey, "Reverse engineering of relational databases: Extraction of an EER model from a relational database," *Data Knowl. Eng.*, vol. 12, no. 2, pp. 107–142, 1994. [Online]. Available: http://dx.doi.org/10.1016/0169-023X(94)90011-6
17. R. Alhajj, "Extracting the extended entity-relationship model from a legacy relational database," *Inf. Syst.*, vol. 28, no. 6, pp. 597–618, 2003. [Online]. Available: http://dx.doi.org/10.1016/S0306-4379(02)00042-X

18. E. Nooijen, B. v. Dongen, and D. Fahland, "Automatic Discovery of Data-Centric and Artifact-Centric Processes," in *Business Process Management Workshops*. Springer, 2013, pp. 316–327.
19. C. Yu and H. V. Jagadish, "Schema summarization," in *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. ACM, 2006, pp. 319–330. [Online]. Available: http://www.vldb.org/conf/2006/p319-yu.pdf
20. V. Popova, D. Fahland, and M. Dumas, "Artifact lifecycle discovery," *International Journal of Cooperative Information Systems, World Scientific.*, 2014 (to appear).
21. J. E. Ingvaldsen and J. A. Gulla, "Preprocessing support for large scale process mining of sap transactions," in *Business Process Management Workshops*. Springer, 2008, pp. 30–41.
22. A. Ramesh, "Process mining in peoplesoft," Master's thesis, Eindhoven University of Technology, 2006.
23. K. Yano, Y. Nomura, and T. Kanai, "A practical approach to automated business process discovery," in *Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2013 17th IEEE International*. IEEE, 2013, pp. 53–62.
24. W. v. d. Aalst, A. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 16, no. 9, pp. 1128–1142, 2004.
25. A. Weijters and J. Ribeiro, "Flexible heuristics miner (fhm)," in *Computational Intelligence and Data Mining (CIDM), 2011 IEEE Symposium on*. IEEE, 2011, pp. 310–317.
26. A. d. Medeiros, A. Weijters, and W. v. d. Aalst, "Genetic process mining: an experimental evaluation," *Data Mining and Knowledge Discovery*, vol. 14, no. 2, pp. 245–304, 2007.
27. J. v. d. Werf, B. v. Dongen, C. Hurkens, and A. Serebrenik, "Process discovery using integer linear programming," in *Applications and Theory of Petri Nets*. Springer, 2008, pp. 368–387.
28. C. Günther and W. v. d. Aalst, "Fuzzy mining–adaptive process simplification based on multi-perspective metrics," in *Business Process Management*. Springer, 2007, pp. 328–343.
29. S. J. J. Leemans, D. Fahland, and W. M. P. v. d. Aalst, "Discovering block-structured process models from event logs-a constructive approach," in *Application and Theory of Petri Nets and Concurrency*. Springer, 2013, pp. 311–329.
30. ——, "Discovering block-structured process models from non-conforming event logs," in *In 9th International Workshop on Business Process Intelligence 2013 (BPI), Beijing, China*, 2013.
31. J. De Weerdt, M. De Backer, J. Vanthienen, and B. Baesens, "A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs," *Information Systems*, vol. 37, no. 7, pp. 654–676, 2012.
32. D. Lo and S. Khoo, "Smartic: towards building an accurate, robust and scalable specification miner," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, M. Young and P. T. Devanbu, Eds. ACM, 2006, pp. 265–275. [Online]. Available: http://doi.acm.org/10.1145/1181775.1181808
33. M. Gabel and Z. Su, "Javert: Fully automatic mining of general temporal properties from dynamic traces," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 339–349. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453150
34. D. Fahland, D. Lo, and S. Maoz, "Mining branching-time scenarios," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, E. Denney, T. Bultan, and A. Zeller, Eds. IEEE, 2013, pp. 443–453. [Online]. Available: http://dx.doi.org/10.1109/ASE.2013.6693102

35. M. Pradel and T. Gross, "Automatic generation of object usage specifications from large method traces," in *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, Nov 2009, pp. 371–382.
36. A. Zeller, "Specifications for free," in *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 6617. Springer, 2011, pp. 2–12. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-20398-5_2
37. B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *Software Engineering, IEEE Transactions on*, vol. 35, no. 5, pp. 684–702, Sept 2009.
38. J. Lou, Q. Fu, S. Yang, J. Li, and B. Wu, "Mining program workflow from interleaved traces," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010*, B. Rao, B. Krishnapuram, A. Tomkins, and Q. Yang, Eds. ACM, 2010, pp. 613–622. [Online]. Available: http://doi.acm.org/10.1145/1835804.1835883
39. J. Pinggera, P. Soffer, D. Fahland, M. Weidlich, S. Zugal, B. Weber, H. Reijers, and J. Mendling, "Styles in business process modeling: an exploration and a model," *Software & Systems Modeling*, pp. 1–26, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10270-013-0349-1
40. R. Minelli, A. Mocci, M. Lanza, and L. Baracchi, "Visualizing developer interactions," in *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, Sept 2014, pp. 147–156.
41. M. Schur, A. Roth, and A. Zeller, "Mining behavior models from enterprise web applications," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 422–432. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491426
42. D. Fahland, M. d. Leoni, B. v. Dongen, and W. v. d. Aalst, "Behavioral conformance of artifact-centric process models," in *Business Information Systems*. Springer, 2011, pp. 37–49.
43. ——, "Conformance checking of interacting processes with overlapping instances," in *Business Process Management*. Springer, 2011, pp. 345–361.
44. E. Nooijen, "Artifact-Centric Process Analysis–Process discovery in ERP systems." Master's thesis, Eindhoven University of Technology, 2012.
45. A. Petermann, M. Junghanns, R. Muller, and E. Rahm, "Biiig: enabling business intelligence with integrated instance graphs," in *Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on*. IEEE, 2014, pp. 4–11.
46. R. Conforti, M. Dumas, L. García-Bañuelos, and M. La Rosa, "Beyond tasks and gateways: Discovering bpmn models with subprocesses, boundary events and activity markers," in *Business Process Management*. Springer, 2014, pp. 101–117.
47. A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 35–44. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287632
48. A. Mocci and M. Sangiorgio, "Detecting component changes at run time with behavior models," *Computing*, vol. 95, no. 3, pp. 191–221, 2013. [Online]. Available: http://dx.doi.org/10.1007/s00607-012-0214-z
49. A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database system concepts*. McGraw-Hill Hightstown, 1997, vol. 4.

50. J. E. Ingvaldsen and J. A. Gulla, "Preprocessing support for large scale process mining of SAP transactions," in *Business Process Management Workshops, BPM 2007 International Workshops, BPI, BPD, CBP, ProHealth, RefMod, semantics4ws, Brisbane, Australia, September 24, 2007, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 4928. Springer, 2007, pp. 30–41. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78238-4_5
51. X. Lu, "Artifact-Centric Log Extraction and Process Discovery," Master's thesis, Eindhoven University of Technology, 2013.
52. D. Embley and B. Thalheim, *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*. Springer, 2012.