Software & Systems Modeling (2019) 18:1209–1247
https://doi.org/10.1007/s10270-018-0664-7

SPECIAL SECTION PAPER

Connecting databases with process mining: a meta model and toolset

Eduardo González López de Murillas[1] · Hajo A. Reijers[1,2] · Wil M. P. van der Aalst[1,3]

Received: 27 December 2016 / Revised: 8 January 2018 / Accepted: 7 February 2018 / Published online: 19 February 2018

Abstract
Process mining techniques require event logs which, in many cases, are obtained from databases. Obtaining these event logs is not a trivial task and requires substantial domain knowledge. In addition, an extracted event log provides only a single view on the database. To change our view, e.g., to focus on another business process and generate another event log, it is necessary to go back to the source of data. This paper proposes a meta model to integrate both process and data perspectives, relating one to the other. It can be used to generate different views from the database at any moment in a highly flexible way. This approach decouples the data extraction from the application of analysis techniques, enabling the application of process mining in different contexts.

Keywords Process mining · Database · Data schema · Meta model · Event extraction

1 Introduction

The field of process mining offers a wide variety of techniques to analyze event data. Process discovery, conformance and compliance checking, performance analysis, process monitoring and prediction, and operational support are some of the techniques that process mining provides to better understand and improve business processes. However, most of these techniques rely on the existence of an event log.

Obtaining event logs in real-life scenarios is not a trivial task. It is not common to find logs exactly in the right form. In many occasions, such logs simply do not exist and need to be extracted from some sort of storage, like databases. In these situations, when a database exists, several approaches are available to extract events. The most general is the classical extraction in which events are manually obtained from the tables in the database. To do so, substantial domain knowledge is required to select the right data scattered across tables. Some work has been done in this field to assist in the extraction and log generation task [2]. Also, studies have been performed on how to extract events in specific environments like SAP [8,10,18] or other ERP systems [12]. In [5], we presented a more general solution to extract events from databases, regardless of the application under study. The paper describes how to automatically obtain events from database systems that generate *redo logs* as a way to recover from failure. All mentioned approaches aim at, eventually, generating an *event log*, i.e., a set of traces, each of them containing a set of *events*. These events represent operations or actions performed in the system under study and are grouped in traces following some criteria. However, there are multiple ways in which events can be selected and grouped into traces. Depending on the perspective we want to take on the data, we need to extract event logs differently. Also, a database contains a lot more information than just events. The extraction of events and its representation as a plain event log can be seen as a "lossy" process during which valuable information can get lost. Considering the prevalence of databases as a source for event logs, it makes sense to gather as much information as possible, combining the process view with the actual data.

Communicated by Dr. Ilia Bider and Rainer Schmidt.

✉ Eduardo González López de Murillas
e.gonzalez@tue.nl

Hajo A. Reijers
h.a.reijers@tue.nl ; h.a.reijers@vu.nl

Wil M. P. van der Aalst
w.m.p.v.d.aalst@tue.nl ; wvdaalst@pads.rwth-aachen.de

1 Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands

2 Department of Computer Science, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands

3 Department of Computer Science, RWTH Aachen University, Aachen, Germany

We see that process mining techniques grow more and more sophisticated. Yet, the most time-consuming activity, *event log extraction*, is hardly supported. In big industrial database settings, where event data are scattered through hundreds of tables, and many processes coexist in the same environment, the queries used to extract event logs can become very complicated, difficult to write and hard to modify. Ideally, users should be able to find events explicitly defined and stored in a centralized way. These events should be defined in such a way that event correlation and log building could be performed effortlessly and be easily adapted to the business questions to answer in each situation. Also, to discover meaningful data rules, these events should be annotated with enough data attributes.

This paper aims at providing support to tackle the problem of *obtaining, transforming, organizing and deriving data and process information from databases*, abstracting the data to high-level concepts. This means that, after the ETL procedure is applied, users will no longer have to deal with low-level raw data scattered through tables (e.g., timestamps, activity names and case ids as columns of different tables that need to be joined together). Conversely, users will be able to focus on the analysis, dealing only with familiar process elements such as events, cases, logs and activities, or with data elements such as objects, object versions[1], object classes and attributes. Also, the new meta model proposed in this work can be seen as a data warehouse schema that captures all the pieces of information necessary to apply process mining to database environments.

In order to build event logs with events from databases, languages like SQL are the natural choice for many users. The point of this work is not to replace the use of query languages, but to provide a common meta model as a standard abstraction that ensures process mining applicability. Moreover, one of the main advantages of the adoption of a standard meta model has to do with multi-perspective event log building and analysis. Many different event logs can be built from the information stored in a database. Looking at data from different perspectives requires ad hoc queries that extract and correlate events in different ways (e.g., (a) order, delivery and invoice events, versus (b) order, product and supplier events). Our meta model defines the abstract concepts that need to be extracted to enable this multi-perspective event log building in a more intuitive way.

Additionally, as a result of the adoption of the proposed meta model, it becomes easier to connect the event recording system of enterprises with analysis tools, generating different views on the data in a flexible way. Also, this work presents a comprehensive integration of process and data information in a consistent and unified format. All of this is supported by our implementation. Moreover, the provided solution has the benefit of being universal, being applicable regardless of the specific system in use. Figure 1 depicts an environment in which the process information of a company is scattered over several systems from a different nature, like ERPs, CRMs, BPM managers, database systems, redo logs, etc. In such a heterogeneous environment, the goal is to extract, transform and derive data from all sources to a common representation. By putting all pieces together, analysis techniques like process mining can be readily applied.

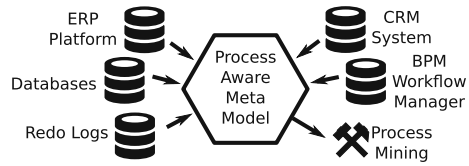

Fig. 1 Data gathering from several systems to a meta model

The remainder of this paper is structured as follows: Section 2 presents a running example used throughout the paper. Section 3 explains the proposed meta model. Implementation details are presented in Sect. 4. The approach is evaluated in Sect. 5 on three real-life environments. The results of the evaluation and the querying of the output of our approach are analyzed in Sect. 6. Section 7 discusses the related work, and finally, Sect. 8 presents conclusions and future work. Additionally, Appendix A provides a formalization of the meta model, and Appendix B extends the formal details of the real-life environments presented in Sect. 5.

2 Running example

In this section, we propose a running example to explain and illustrate our approach. Assume we want to analyze a setting where concerts are organized and concert tickets are sold. To do so, a database is used to store all the information related to concerts, concert halls (*hall*), seats, tickets, bands, performance of bands in concerts (*band_playing*), customers and bookings. Figure 2 shows the data schema of the database. In it we see many different elements of the involved process represented. Let us consider now a complex question that could have been posed from a business point of view: *What is the behavior through the process of customers between 18 and 25 years old who bought tickets for concerts of band X?* This question represents a challenge starting from the given database for several reasons:

[1] In this context, the term "version" or "object version" refers to an instance of an object at a point in time (e.g., the database object corresponding to a specific customer had two different values for the attribute "address" at different points in time, representing two versions of the same customer object). This is different of the usual meaning of "version" in the software context as a way to distinguish different code releases (e.g., version 1.1).

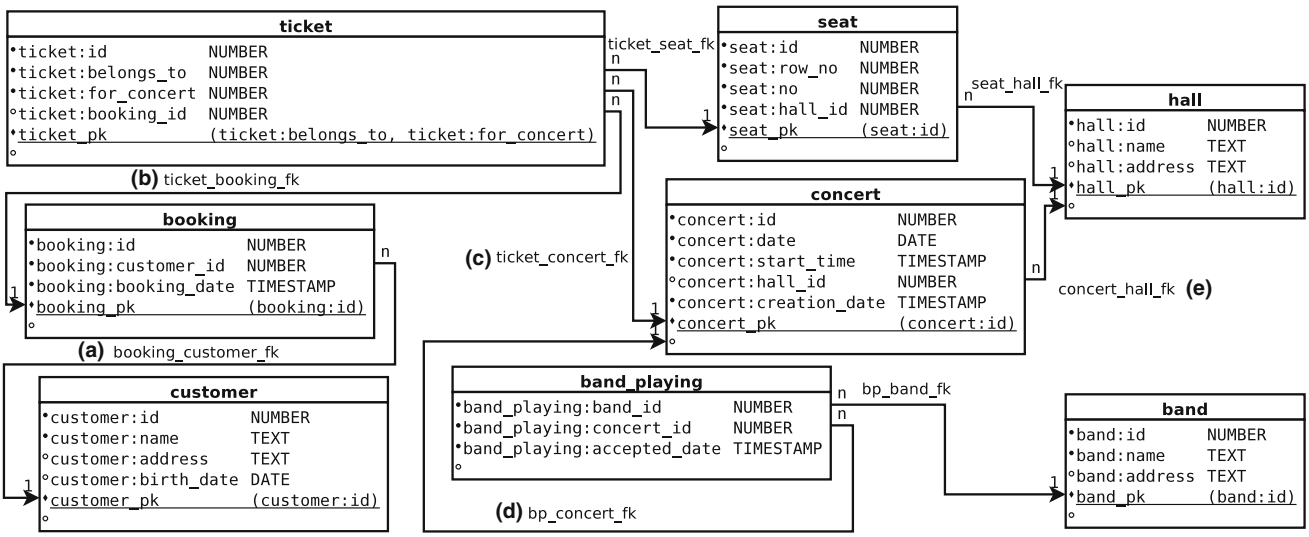

Fig. 2 Data schema of the example database

1. The database does not provide an integrated view of process and data. Therefore, questions related to the execution of the underlying process with respect to some of the elements cannot be directly answered with a query.
2. The current database schema fits the purpose of storing the information in this specific setting, but it does not have enough flexibility to extend its functionality allocating new kinds of data such as events or objects of a different nature.
3. The setting lacks execution information in an accessible way (events, traces and logs are missing so one cannot apply process mining directly), and there is no assistance on how to extract or derive this information from the given data.
4. If we plan to use the data as it is, we need to adapt to the way it is stored for every question we want to answer.

All these reasons make the analysis complex in many settings. At best, such an analysis can only be carried out by the extraction of a highly specialized event log by creating a complex ad hoc query. Besides, the extraction will need to be repeated for new questions that require a new viewpoint on the data.

If we consider that, for the present database, some sort of event recording system is in place, the extraction of additional events becomes feasible. Listing 1 shows an example of an ad hoc query to answer the sample question posted above. This query makes use of the event data scattered through the database (*booking*, *band*, etc.), as depicted in Fig. 2, together with events recorded by a redo log system provided by the RDBMS (*redo_logs_mapping* table). The way to access redo log information has been simplified in this example for the sake of clarity. The first part of the query retrieves *ticket booked* events, making use of the *booking_date* timestamp stored in the table *booking*. These events are united to the ones corresponding to *customer* events. These last ones are obtained from redo logs, due to the lack of such event information in the data schema. Then, events belonging to the same customer are correlated by means of the *caseid* attribute, and the cases are restricted to the ones belonging to customers aged 18–25 at the time the booking was made. Additionally, we need to keep only the bookings made on bands named "X" at the moment of the booking. (A band could change its name at any point in time.) This is an especially tricky step since we need to look into the redo logs to check if the name was different at the time the booking was made.

Listing 1 Sample query to obtain a highly specialized event log from the database

```
SELECT * FROM (
 SELECT
  "ticket booked" as activity,
  BK.booking_date as timestamp,
  BK.customer_id as caseid
 FROM booking as BK
 UNION
 SELECT
  concat(RL.operation," customer profile") as activity,
  RL.timestamp as timestamp,
  rl_value_of('id') as caseid
 FROM redo_logs_mapping as RL
 WHERE
  RL.table = "CUSTOMER"
 ) as E,
 booking as BK,
 customer as CU,
 ticket as T,
 concert as C,
 band_playing as BP,
 band as B
WHERE
 E.caseid = CU.id AND
 CU.id = BK.customer_id AND
 BK.id = T.booking_id AND
 T.for_concert = C.id AND
 C.id = BP.concert_id AND
 BP.band_id = B.id AND
 "0018−00−00 00:00:00" <= (BK.booking_date − CU.birth_date) AND
```

```
"0025−00−00_00:00:00" >= (BK.booking_date − CU.birth_date) AND
(
 (B.name = "X" AND
  B.id NOT IN
   (SELECT rl_value_of('id') as id
    FROM redo_logs_mapping as RL
    WHERE
     RL.table = "BAND"
   )
 )
 OR
 (B.id IN
   (SELECT rl_value_of('id') as id
    FROM redo_logs_mapping as RL
    WHERE
     RL.table = "BAND" AND
     rl_new_value_of('name') = "X" AND
     RL.timestamp <= BK.id AND
    ORDER BY RL.timestamp DESC LIMIT 1
   )
 )
 OR
 (B.id IN
   (SELECT rl_value_of('id') as id
    FROM redo_logs_mapping as RL
    WHERE
     RL.table = "BAND" AND
     rl_old_value_of('name') = "X" AND
     RL.timestamp >= BK.id AND
    ORDER BY RL.timestamp ASC LIMIT 1
   )
 )
)
```

It is evident that extracting specialized event logs that answer very specific questions, while maintaining the original data schema, is possible. However, such queries will have to be adapted or rewritten for every different setting or event recording system. The fact that users, and especially business analysts, need to be knowledgeable about the particularities of how event data are stored represents an important challenge to overcome. This is one of the main reasons why so much time and effort is consumed during the ETL phase, that should be devoted to analysis. This work aims at supporting the ETL phase and ultimately at decoupling it from the analysis by providing a set of familiar abstractions readily available for the business analysts.

3 Meta model

As has been shown before, a need exists for a way to store execution information in a structured way, something that accepts data from different sources and allows building further analysis techniques independently from the origin of this data. Efforts in this field have already been made as can be observed in [7] with the IEEE XES standard. This standard defines a structure to manage and manipulate logs, containing events and traces and the corresponding attributes. Therefore, XES is a good format to represent behavior. However, an XES file is just one view on the data, and despite being an extensible format, it does not provide a predefined structure to store all the linked information we want to consider.

Because of this, it seems necessary to define a structured way to store additional information that can be linked to the classical event log. This new way to generalize and store information must provide sufficient details about the process, the data types and the relations between all the elements, making it possible to answer questions at the business level, while looking at two different perspectives: data and process.

3.1 Requirements

To be able to combine the data and process perspectives in a single structure, it is important to define a set of requirements that a meta model must fulfill. It seems reasonable to define requirements that consider backwards compatibility with well-established standards, support of additional information, its structure and the correlation between process and data views:

1. The meta model must be compatible with the current meta model of XES, i.e., any XES log can be transformed to the new meta model and back without loss of information,
2. It must be possible to store several logs in the new meta model, avoiding event duplication,
3. Logs stored in the same meta model can share events and belong to different processes,
4. It must be possible to store some kind of process representation in the meta model,
5. The meta model must allow storing additional information, like database objects, together with the events, traces and processes, and the correlation between all these elements,
6. The structure of additional data must be precisely modeled,
7. All information mentioned must be self-contained in a single storage format, easy to share and exchange, similarly to the way that XES logs are handled.

The following section describes the proposed meta model which complies with these requirements, providing a description of the concepts. A formalization of the meta model can be found in Appendix A.

3.2 Description

Considering the typical environments subject to study in the process mining field, we can say that it is common to find systems backed up by some sort of database storage system. Regardless of the specific technology behind these databases, all of them have in common some kind of structure for data. We can describe our meta model as a way to integrate process and data perspectives, providing flexibility on its inspection and assistance to reconstruct the missing parts. Figure 3 shows a high-level representation of the meta model. On the right-hand side, the data perspective is considered, while the left models the process view. Assuming that the starting point

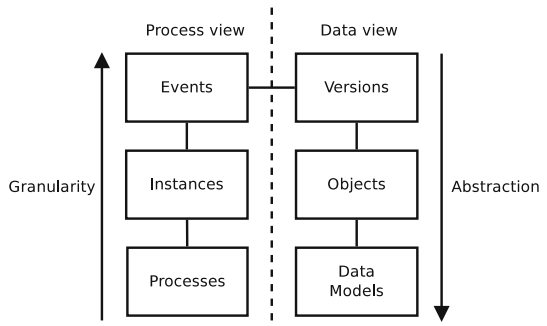

Fig. 3 Diagram of the meta model at a high level

of our approach is data, we see that the less abstract elements of the meta model, *events* and *versions*, are related, providing the connection between the process and data view. These are the basic blocks of the whole structure and, usually, the rest can be derived from them.

The data side considers three elements: **data models**, **objects** and **versions**. The data models provide a schema describing the objects of the database. The objects represent the unique entities of data that ever existed or will exist in our database, while the versions represent the specific values of the attributes of an object during a period of time. Versions represent the evolution of objects through time. The process side considers **events**, **instances** and **processes**. Processes describe the behavior of the system. Instances are traces of execution for a given process, being sets of events ordered through time. These events represent the most granular kind of execution data, denoting the occurrence of an activity or action at a certain point in time.

Figure 4 depicts the entity-relation diagram of the meta model. Some elements have been omitted in the diagram for the sake of simplicity. A full version of the ER diagram is available online[2], and a formalization is provided in Appendix A. Each of the entities in the diagram, represented by a square, corresponds to the basic *entities* of the meta model. Also, these entities, together with their relations (diamond shapes), have been grouped in areas that we call *sectors* (delimited by dashed lines). These sectors: *data models*, *objects*, *versions*, *events*, *cases* and *process models* contain tightly related concepts and provide an abbreviated representation of the meta model. For the sake of clarity, the "sectorized" representation of the meta model will be used in further sections. As can be observed, the entity-relation diagram is divided into six sectors. The purpose of each of them is described below:

- **Data models**: this sector is formed by concepts needed to describe the structure of any database system. Many data models can be represented together in this sector, whose main element is the *data model* entity. For each data model, several *classes* can exist. These classes are abstractions of the more specific concept of table, which is commonly found in RDBMSs. Looking at the database provided in Sect. 2, the tables *customer* and *booking* are examples of classes. These classes contain *attributes*, which are equivalent to table columns in modern databases (e.g., *id*, *name*, *address*, etc.). The references between classes of the same data model are represented with the *relationship* entity. This last entity holds links between a source and a target class (e.g., *booking_customer_fk* which relates the source class *booking* to the target class *customer*).
- **Objects**: the *object* entity, part of the objects sector, represents each of the unique data elements that belong to a class. An example of this can be a hypothetical customer with *customer_id* = 75. Additional details of this object are omitted, given that they belong to the next sector.
- **Object versions (Versions)**: for each of the unique object entities described in the previous sector, one or many *versions* can exist. A version is an instantiation of an object during a certain period of time, e.g., the customer object with id 75, existed in the database, during a certain period of time, for example from "2015-08-01 14:45:00" to "2016-09-03 12:32:00". During that period of time, the object had specific values for the attributes of the customer class that it belongs to. Therefore, there is a version of customer 75, valid between the mentioned dates, with name "John Smith", address "45, 5th Avenue", and birth date "1990-01-03". If at some point, the value of one of the attributes changed (e.g., a new address), the end timestamp of the previous version would be set to the time of the change, and a new version would be created with the updated value for that attribute, and a start timestamp equal to the end of the previous version, e.g., version_1 = {object_id = 75, name = "John Smith", address = "45, 5th Avenue", birth_date = "1990-01-03", start_timestamp = "2015-08-01 14:45:00", end_timestamp = "2016-09-03 12:32:00"}, and version_2 = {object_id = 75, name = "John Smith", address = "floor 103, Empire State Building", birth_date = "1990-01-03", start_timestamp = "2016-09-03 12:32:00", end_timestamp = NONE }. Note that the value of *end_timestamp* for the newly created object version (*version_2*) is NONE. That means that it is the current version for the corresponding object (*object_id* = 75). Another entity reflected in this sector is the concept of *relation*. A relation is an instantiation of a relationship and holds a link between versions of objects that belong to the source and target classes of the relationship. For example, a version of a booking object can be related to another version of a customer object

[2] https://github.com/edugonza/OpenSLEX/blob/master/doc/meta-model.png.

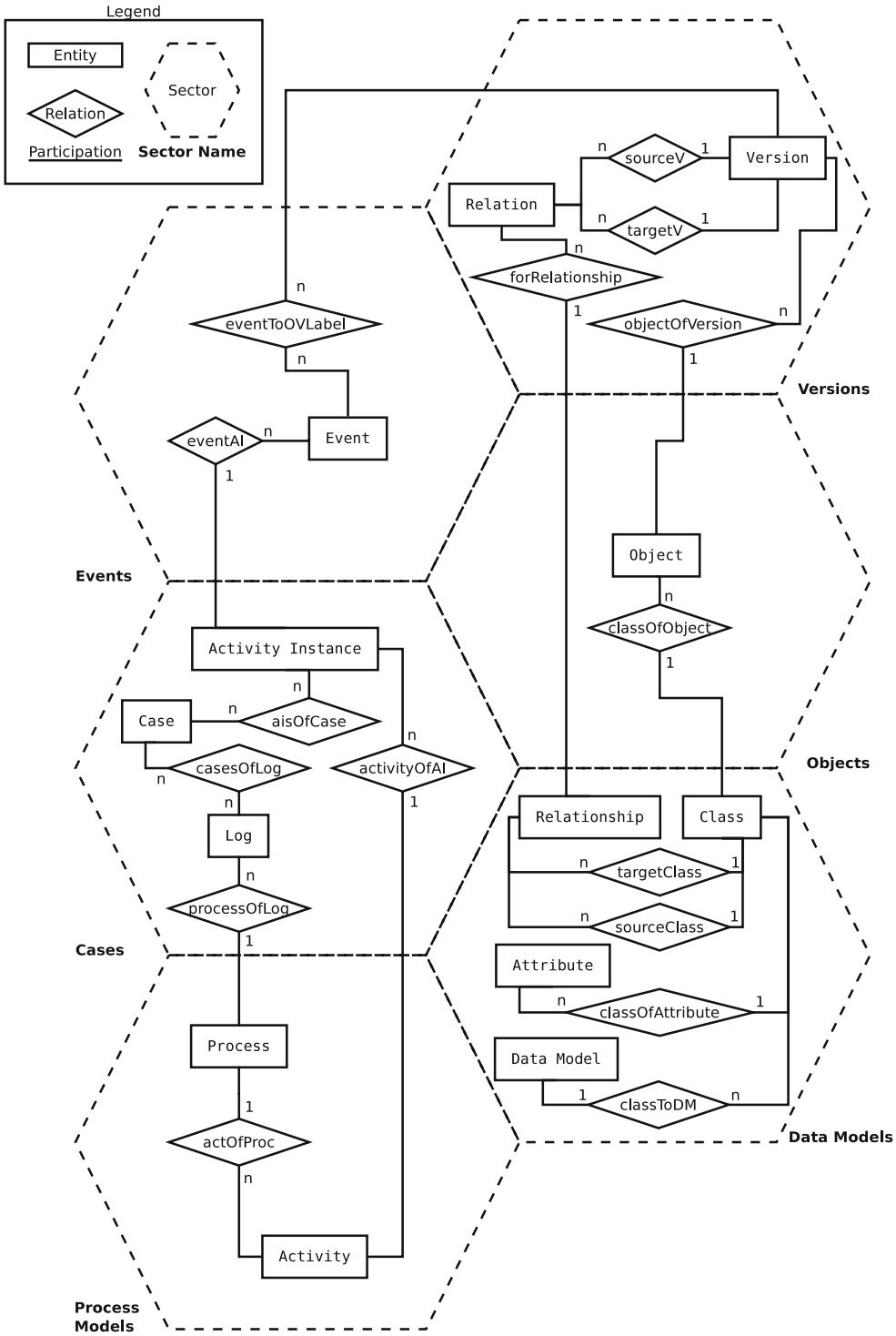

Fig. 4 ER diagram of the meta model. The entities have been grouped in sectors, delimited by the dashed lines

by means of a relation instance, given that a relationship (*booking_customer_fk*) exists from class *booking* to class *customer*.

– **Events**: this sector collects a set of events, obtained from any available source (database tables, redo logs, change records, system logs, etc.). In this sector, events appear as a collection, not grouped into traces (such grouping is reflected in the next sector). In order to keep process information connected to the data side, each event can be linked to one or many object versions by means of

a label (*eventToOVLabel*). This label allows specifying what kind of interaction exists between the event and the referred object version, e.g., *insert*, *update*, *delete*, *read*, etc. Events hold details such as *timestamp*, *life cycle* and *resource* information, apart from an arbitrary number of additional event attributes.

– **Cases and instances**: the entities present in this sector are very important from the process mining point of view. The events by themselves do not provide much information about the control flow of the underlying process, unless they are correlated and grouped into traces (or cases). First, the *activity instance* entity must be explained. This entity is used to group events that refer to the same instance of a certain activity with different values for its life cycle, e.g., the execution of the activity *book_tickets* generates one event for each phase of its life cycle: *book_tickets+start* and *book_tickets+complete*. Both events, referring to the same execution of an activity, are grouped in the same activity instance. Then, as in any other event log format, activity instances can be grouped in *cases*, and these cases, together, form a *log*.
– **Processes**: the last sector contains information about *processes*. Several processes can be represented in the same meta model. Each process is related to a set of *activities*, and each of these activities can be associated with several activity instances, contained in the corresponding *cases and instances* sector.

An instantiation of this meta model fulfills the requirements set in Sect. 3.1 in terms of storage of data and process view. Some characteristics of this meta model that enable full compatibility with the XES standard have been omitted in this formalization for the sake of brevity. Additionally, an implementation of this meta model has been made. This was required in order to provide tools that assist in the exploration of the information contained within the populated meta model. More details on this implementation are explained in the following section.

4 Implementation

The library Open SQL Log Exchange (OpenSLEX[3]), based on the meta model proposed in this work has been implemented in Java. This library provides an interface to insert data in an instantiation of this meta model and to access it in a similar way to how XES Logs are managed by the IEEE XES Standard [7]. However, under the hood it relies on SQL technology. Specifically, the populated meta model is stored in a SQLite[4] file. This provides some advantages like an SQL query engine, a standardized format as well as storage in self-contained single data files that benefits its exchange and portability. Figure 4 shows an ER diagram of the internal structure of the meta model. However, it represents a simplified version to make it more understandable and easy to visualize. The complete class diagram of the meta model can be accessed in the OpenSLEX's website[3]. In addition to the library mentioned earlier, an inspector has been included in the Process Aware Data Suite (PADAS)[5] tool. This inspector, depicted in Fig. 5, allows exploring the content of OpenSLEX files by means of a GUI in an exploratory fashion, which lets the user dig into the data and apply some basic filters on each element of the structure. The tool presents a series of blocks that contain the *activities*, *logs*, *cases*, *activity instances*, *events*, *event attribute values*, *data model*, *objects*, *object versions*, *object version attribute values* and *relations* entities in the meta model. Some of the lists in the inspector (*logs*, *cases*, *activity instances*, *events* and *objects*) have tabs that allow one to filter the content they show. For instance, if the tab *"Per Activity"* in the *cases* list is clicked, only cases that contain events of such activity will be shown. In the same way, if the tab *"Per Case"* in the *events* list is clicked, only events contained in the selected case will be displayed. An additional block in the tool displays the attributes of the selected event.

The goal of providing these tools is to assist in the task of populating the proposed meta model, in order to query it in a posterior step. Because the meta model structure and the data inserted into it are stored in a SQLite file, it is possible to execute SQL queries in a straightforward way. In particular, the whole process of extracting, transforming and querying data has been implemented in a RapidProM[6] workflow. RapidProM is an extension that adds process mining plugins from ProM to the well-known data mining workflow tool RapidMiner[7]. For our task, an OpenSLEX meta model population operator has been implemented in a development branch[8] of RapidProM. This operator, together with the data handling operators of RapidMiner (including database connectors), lets us extract, transform and query our populated meta model automatically in a single execution of the workflow. More details on this workflow and the extraction and transformation steps are provided in Sect. 5, showing how the technique can be applied in different real-life environments. A workflow was designed for each of the evaluation environments with the purpose of extracting and transforming the content of the corresponding databases to fit the structured of the OpenSLEX meta model. Each of these workflows, also referred to as adapters, is extensible using the collec-

[3] http://www.win.tue.nl/~egonzale/projects/openslex/.

[4] http://www.sqlite.org/.

[5] http://www.win.tue.nl/~egonzale/projects/padas/.

[6] http://rapidprom.org/.

[7] http://rapidminer.com/.

[8] https://github.com/rapidprom/rapidprom-source/tree/egonzalez.

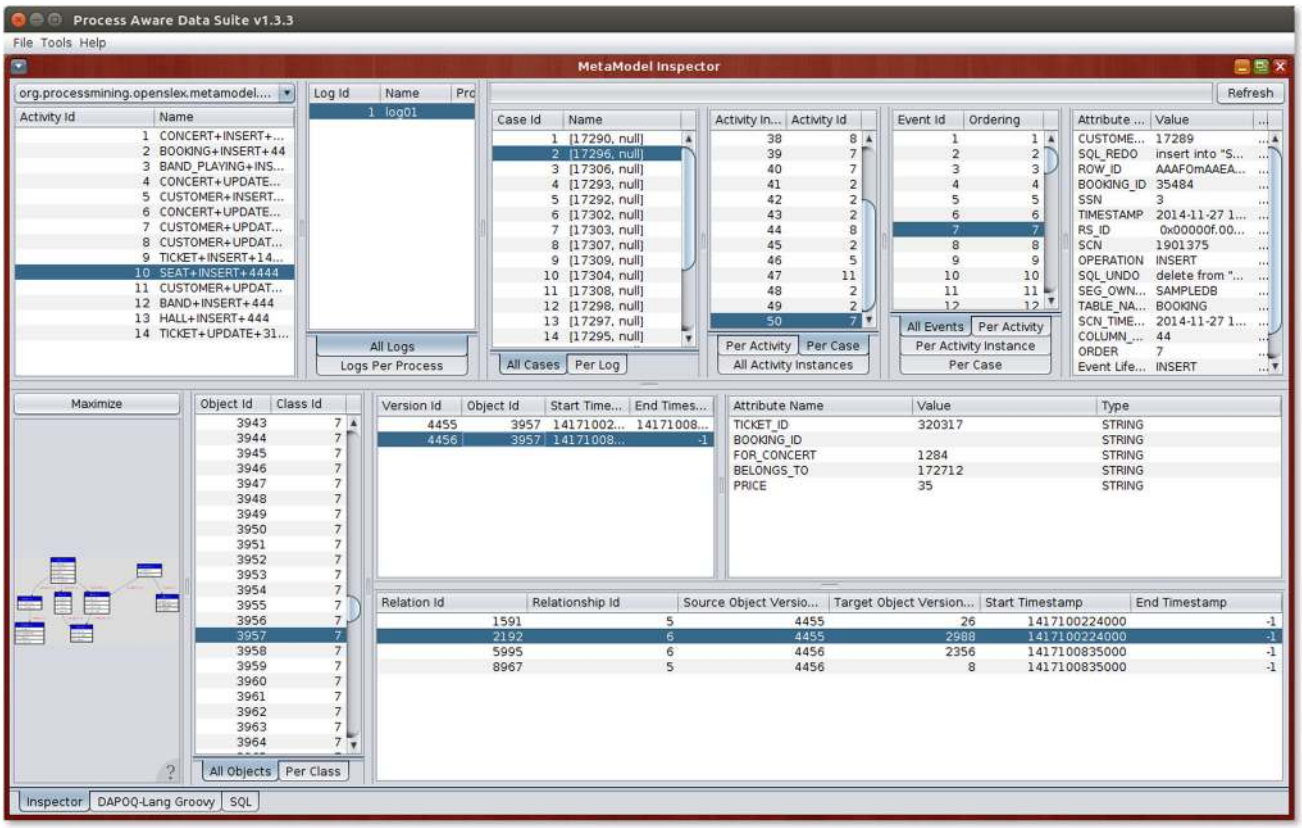

Fig. 5 Screenshot of the meta model inspector tool

tion of operators available in RapidMiner, RapidProM and other plugins, and can be modified to fit new environments. Moreover, Sect. 6 demonstrates how querying of the resulting OpenSLEX populated meta model can be standardized for all the proposed environments.

The provided implementation of the OpenSLEX meta model is a general and extensible platform for data extraction and transformation. It provides the means to abstract raw data to high-level concepts. These concepts (events, cases, objects, etc.) are easier to deal with for business analysts. On the other hand, limitations exist to the current implementation. A general method does not exist at the moment to perform this extraction and transformation independently of the raw data structure. This means that an ad hoc adapter needs to be designed for different data architectures (SAP data schema, Oracle redo logs, in-table versioning, etc.) in order to properly extract and interpret events and object versions. However, we believe that the three adapters provided with the implementation should suffice in most cases and serve as a template for other environments. It is important to note that, despite the need to design an extraction adapter for each type of environment, it supposes an advantage with respect to writing ad hoc queries for event log extraction. This is due to the fact that, once the extraction and transformation to OpenSLEX are performed, automated methods (implemented as operators in the RapidProM platform) become available for generating multiple event logs from different perspectives in a single dataset. This saves time and effort and is less prone to errors than designing ad hoc queries for each specific event log perspective.

5 Application in real-life environments

The development of the meta model presented in this paper has been partly motivated by the need of a general way to capture the information contained in different systems combining the data and process views. Such systems, usually backed up by a database, use very different ways to internally store their data. Therefore, in order to extract these data, it is necessary to define a translation mechanism tailored to the wide variety of such environments. Because of this, *the evaluation aims at demonstrating the feasibility of transforming information from different environments to the proposed meta model*. Specifically, three real-life source environments are analyzed:

1. *Database redo logs*: files generated by the RDBMS in order to maintain the consistency of the database in case of failure or rollback operations. The data were

obtained from real redo logs generated by a running Oracle instance. A synthetic process was designed and run in order to insert and modify data in the database and trigger the generation of redo logs. Because of the simplicity of the data model, this environment inspired the running example of Sect. 2.

2. *In-table version storage*: Application-specific schema to store new versions of objects as a new row in each table. The data of this analysis was obtained from a real-life instance of a Dutch financial organization.
3. *SAP-style change tables*: changes in tables are recorded in a "redo log" style as a separate table, the way it is done in SAP systems. For this analysis, real SAP data, generated by an external entity, were used. Such data are often used for training purposes by the third-party organization.

The benefit of transforming data to a common representation is that it allows for decoupling the application of techniques for the analysis from the sources of data. In addition, a centralized representation allows linking data from different sources. However, the source of data may be incomplete in itself. In some real-life scenarios, explicitly defined events might be missing. Other scenarios do not have a record of previous object versions. Something common is the lack of a clear case notion, therefore the absence of process instances. In all these cases, when transforming the data to only fit in our meta model is not enough, we need to apply some automated inference techniques. This allows one to derive the missing information and create a complete and fully integrated view. Figure 6 shows these environments, and which sectors of our meta model can be populated right away, only extracting what is available in the database. Then, following a series of automated steps like *version inference* and *event inference*, *case derivation*, and *process discovery*, all the sectors can be populated with inferred or derived data.

The first part of this evaluation (Sect. 5.1) presents the different scenarios that we can find when transforming data. Each of these scenarios starts from data that correspond to different sectors of the meta model. Then, we show how to derive the missing sectors from the given starting point. Sections 5.2, 5.3 and 5.4 analyze the three real-life common environments mentioned before. We will demonstrate that data extraction is possible and that the meta model can be populated from these different sources. Section 5.5 presents a method to merge data from two different systems into a single meta model structure, providing an end-to-end process view. Section 6 demonstrates that, from the three resulting populated meta models, it is possible to standardize the process mining analysis and perform it automatically, adapting only a few parameters specific to each dataset. In addition to that, an example of the corresponding resulting populated meta model for each one of the environments is shown.

5.1 Meta model completion scenarios

Experience teaches us that it is rare to find an environment that explicitly provides the information to fill every *sector* of our meta model. This means that additional steps need to be taken to evolve from an incomplete meta model to a complete one. To do so, Fig. 7 presents several scenarios in which, starting from a certain input (gray sectors), it is possible to infer the content of other elements (dashed sectors). Depending on the starting point that we face, we must start inferring the

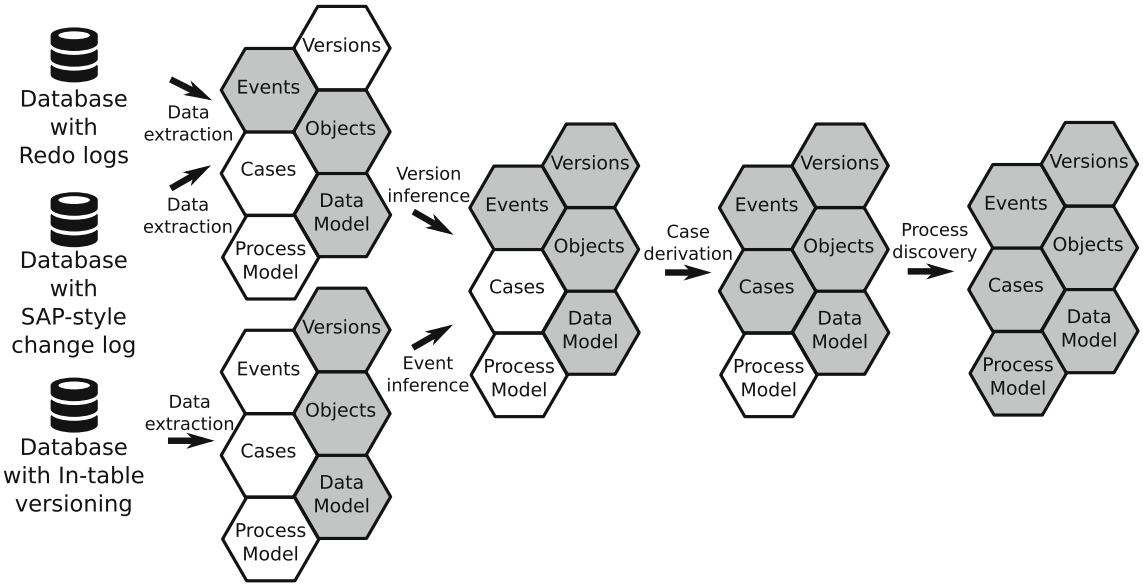

Fig. 6 Meta model completion in the three evaluated environments

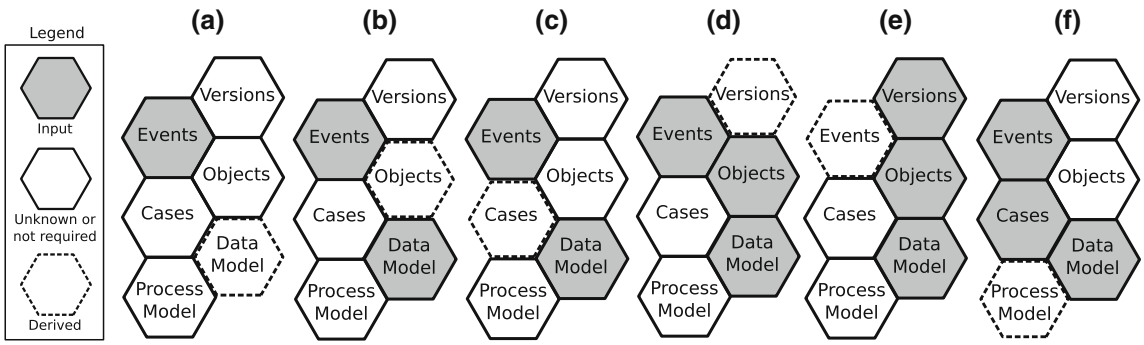

Fig. 7 Input scenarios to complete meta model sectors

missing elements consecutively in the right order, which will lead us, in the end, to a completely populated meta model:

a *Schema discovery*: One of the most basic elements we require in our meta model to be able to infer other elements is the *events sector*. Starting from this input and applying schema, primary key and foreign key discovery techniques [17,25], it is possible to obtain a data model describing the structure of the original data.

b *Object identification*: If the events and a data model are known, we can infer the objects that these events represent. To do so, it is necessary to know the attributes of each class that identify the objects (primary keys). Finding the unique values for such attributes in the events corresponding to each class results in the list of unique objects of the populated meta model.

c *Case derivation*: Another possible scenario is the one in which we derive cases from the combination of events and a data model. The event splitting technique described in [5], which uses the transitive relations between events defined by the data model, allows generating different sets of cases, or event logs. This requires, similarly to scenario *b*, to match the attributes of each event to the attributes of each class. Then, we must use the foreign key relations to correlate events between them. Selecting the desired combination of relationships between events will tell us how to group these events in cases to form a log.

d *Version inference*: The events of each object can be processed to infer the *object versions* as results of the execution of each event. To do so, events must contain the values of the attributes of the object they relate to at a certain point in time or, at least, the values of the attributes that were affected (modified) by the event. Then, ordering the events by (descending) timestamp and applying them to the last known value of each attribute allow us to reconstruct the versions of each object.

e *Event inference*: The inverse of scenario *d* is the one in which events are inferred from object versions. Looking at the attributes that differ between consecutive versions it is possible to create the corresponding event for the modification.

f *Process discovery*: Finally, a set of cases is required to discover a process model using any of the multiple miners available in the process mining field.

The following three sections consider real-life environments that can act as a source for event data. These environments are used to illustrate the scenarios in Fig. 7 and demonstrate that the complete meta model structure can be derived for each of them. In each scenario, the goal is to create an integrated view of data and process, even when event logs are not directly available.

5.2 Database redo logs

The first environment focuses on database *redo logs*, a mechanism present in many DBMSs to guarantee consistency, as well as providing additional features such as rollback and point-in-time recovery. Redo logs have already been considered in our earlier work [5,21] as a source of event data for process mining. This environment corresponds to the scenario depicted in Fig. 7d, where data model, objects and events are available, but object versions need to be inferred. Table 1 shows an example fragment of a redo log obtained from an Oracle DBMS. After its processing, explained in [5], these records are transformed into events. Table 2 shows the derived values for attributes of these events according to the changes observed in the redo log. For instance, we see that the first row of Table 2 corresponds to the processed redo log record observed in the first row in Table 1. It corresponds to a *Customer* insertion; therefore, all the values for each attribute were inexistent before the event was executed (4th column). The second column holds the values for each attribute right after the event occurred. The rest of the events are interpreted in the same way.

Figure 6 shows a general overview of how the meta model sectors are completed according to the starting input data and

Table 1 Fragment of a redo log: each line corresponds to the occurrence of an event

#	Time + Op + Table	Redo	Undo
1	2016-11-27 15:57:08.0 + INSERT + CUSTOMER	insert into "SAMPLEDB". "CUSTOMER" ("ID", "NAME", "ADDRESS", "BIRTH_DATE") values ('17299', 'Name1', 'Address1', TO_DATE('01-AUG-06', 'DD-MON-RR'));	delete from "SAMPLEDB". "CUSTOMER" where "ID" = '17299' and "NAME" = 'Name1' and "ADDRESS" = 'Address1' and "BIRTH_DATE" = TO_DATE('01-AUG-06', 'DD-MON-RR') and ROWID = '1';
2	2016-11-27 16:07:02.0 + UPDATE + CUSTOMER	update "SAMPLEDB". "CUSTOMER" set "NAME" = 'Name2' where "NAME" = 'Name1' and ROWID = '1';	update "SAMPLEDB". "CUSTOMER" set "NAME" = 'Name1' where "NAME" = 'Name2' and ROWID = '1';
3	2016-11-27 16:07:16.0 + INSERT + BOOKING	insert into "SAMPLEDB". "BOOKING" ("ID", "CUSTOMER_ID") values ('36846', '17299');	delete from "SAMPLEDB". "BOOKING" where "ID" = '36846' and "CUSTOMER_ID" = '17299' and ROWID = '2';
4	2016-11-27 16:07:16.0 + UPDATE + TICKET	update "SAMPLEDB". "TICKET" set "BOOKING_ID" = '36846' where "BOOKING_ID" IS NULL and ROWID = '3';	update "SAMPLEDB". "TICKET" set "BOOKING_ID" = NULL where "BOOKING_ID" = '36846' and ROWID = '3';
5	2016-11-27 16:07:17.0 + INSERT + BOOKING	insert into "SAMPLEDB". "BOOKING" ("ID", "CUSTOMER_ID") values ('36876', '17299');	delete from "SAMPLEDB". "BOOKING" where "ID" = '36876' and "CUSTOMER_ID" = '17299' and ROWID = '4';
6	2016-11-27 16:07:17.0 + TICKET + UPDATE	update "SAMPLEDB". "TICKET" set "ID" = '36876' where "BOOKING_ID" IS NULL and ROWID = '5';	update "SAMPLEDB". "TICKET" set "ID" = NULL where "BOOKING_ID" = '36876' and ROWID = '5';

the steps taken to derive the missing sectors. In this case, the analysis of database redo logs allows one to obtain a set of events, together with the objects they belong to and the data model of the database. These elements alone are not sufficient to do process mining without the existence of an event log (cases). In addition, the versions of the objects of the database need to be inferred from the events as well. Therefore, the starting point is a meta model in which the only populated sectors are *Events*, *Objects* and *Data Model*. We need to infer the remaining ones: *Versions*, *Cases* and *Process Models*.

Fortunately, a technique to build logs using different perspectives (Trace ID Patterns) is presented in [5]. The existence or definition of a data model is required for this technique to work. Figure 6 shows a diagram of the data transformation performed by the technique, and how it fits in the proposed meta model structure. A more formal description of the mapping between redo logs and our meta model can be consulted in "Appendix B.2". The data model is automatically extracted from the database schema and is the one included in the meta model. This data model, together with the extracted events, allows us to generate both cases (c)

Table 2 Fragment of a redo log: each row corresponds to the occurrence of an event

#	Attribute name	Value after event	Value before event
1	Customer:id	17299	–
	Customer:name	Name1	–
	Customer:address	Address1	–
	Customer:birth_date	01-AUG-06	–
	RowID	=	1
2	Customer:id	=	{17299}
	Customer:name	Name2	Name1
	Customer:address	=	{Address1}
	Customer:birth_date	=	{01-AUG-06}
	RowID	=	1
3	Booking:id	36846	–
	Booking:customer_id	17299	–
	RowID	=	2
4	Ticket:booking_id	36846	NULL
	Ticket:id	=	(317132)
	Ticket:belongs_to	=	(172935)
	Ticket:for_concert	=	(1277)
	RowID	=	3
5	Booking:id	36876	–
	Booking:customer_id	17299	–
	RowID	=	4
6	Ticket:booking_id	36876	NULL
	Ticket:id	=	(317435)
	Ticket:belongs_to	=	(173238)
	Ticket:for_concert	=	(1277)
	RowID	=	5

and object versions (d). Then, process discovery completes the meta model with a process (f). Once the meta model structure is populated with data, we can make queries on it taking advantage of the established connections between all the entities and apply process mining to do the analysis.

5.2.1 Database redo logs: transformation

In the previous section, we have described the nature of redo logs and how they can be extracted and transformed into event collections first and used to populate the meta model afterward. The goal of this section is to sketch the content of this resulting populated meta model and, in further sections, the kind of analysis that can be done on it.

Table 3 represents quantitatively the input and output data for our meta model population process for each of the entities introduced in Sect. 3. These entities match the ones depicted in Fig. 4. Moreover, this table provides qualitative properties on the nature of the transformation for each entity. In this case, we observe that all the data are obtained from the original database through SQL (except for the redo logs, that require some specific-to-the-job tools from Oracle). The input entities (*data model*, *class*, *attribute*, *relationship* and *event*) are extracted automatically (5th column), directly transformed given that they are explicitly defined in the source data (6th column), and maintaining their quantity (3th vs. 4th columns). However, the remaining entities of our meta model need to be inferred, i.e., derived from the original entities. In this case, this derivation can be done automatically by our tool (PADAS). The data model that describes this dataset is depicted in Fig. 8 and clearly inspired the running example provided in Sect. 2.

5.2.2 Database redo logs: adapter implementation

In order to assist in the task of extracting and processing redo logs, the tool PADAS[9] has been developed. This tool is able to connect to an existing Oracle RDBMS, load a collection of redo logs, extract their content and fill the missing data from a database snapshot, obtain the data model automatically and export the resulting collection of events to an intermediate

[9] https://www.win.tue.nl/~egonzale/projects/padas/.

Table 3 Redo log dataset transformation to populate the OpenSLEX meta model

Entity	Format	# Input Els.	# Output Els.	Aut/Manual	Derivation
Data Model	SQL	1	1	Aut	Explicit
Class	SQL	8	8	Aut	Explicit
Attribute	SQL	27	27	Aut	Explicit
Relationship	SQL	8	8	Aut	Explicit
Object	–	0	6740	Aut	Inferred
Version	–	0	8424	Aut	Inferred
Relation	–	0	13384	Aut	Inferred
Event	Redo Log	8512	8512	Aut	Explicit
Activity Instance	–	0	8512	Aut	Inferred
Case	–	0	21	Aut	Inferred
Log	–	0	1	Aut	Inferred
Activity	–	0	14	Aut	Inferred
Process	–	0	1	Aut	Inferred

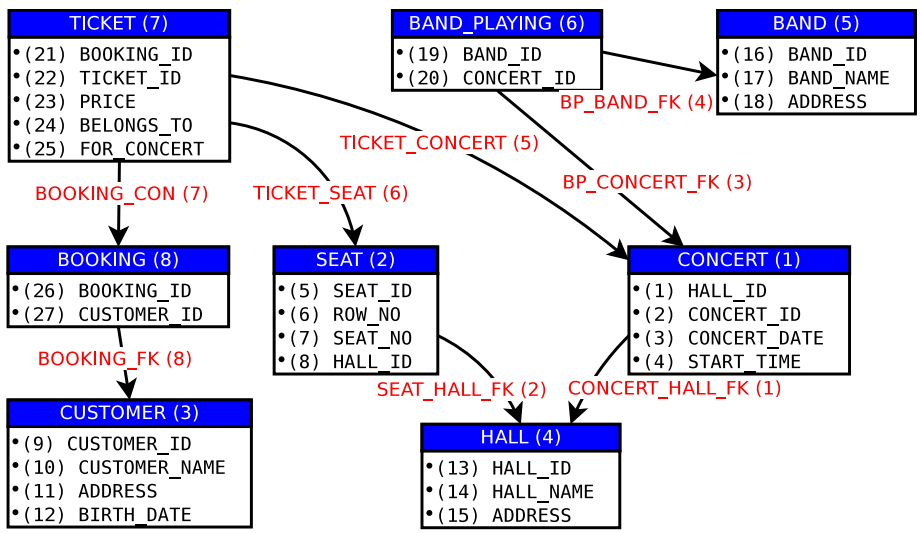

Fig. 8 Data model of the redo log dataset as obtained from the populated meta model

format. Then, the rest of the processing can be done offline from the Oracle database.

In addition to this, the tool assists in selecting case notions from the data model and building logs for a specific view. As can be observed in Fig. 9, once the three main sectors have been extracted, i.e., data model, events and cases, the meta model population can begin. This is an automatic process that does not require any further user interaction.

5.3 In-table versioning

It is not always possible to get redo logs from databases. Sometimes they are disabled or not supported by the DBMS. Also, we simply may not be able to obtain credentials to access them. Whatever the reason, it is common to face the situation in which events are not explicitly stored. This seriously limits the analysis that can be performed on the data. The challenge in this environment is to obtain, somehow, an event log to complete our data.

It can be the case that in a certain environment, despite lacking events, versioning of objects is kept in the database, i.e., it is possible to retrieve the old value for any attribute of an object at a certain point in time. This is achieved by means of duplication of the modified versions of rows. This environment corresponds to the scenario depicted in Fig. 7e, where data model, objects and object versions are available, but events need to be inferred. The table at the bottom left corner of Fig. 10 shows an example of in-table versioning of objects. We see that the primary key of *Customer* is formed by the fields *id* and *load_timestamp*. Each row represents a version of an object, and every new reference to the same *id* at a later *load_timestamp* represents an update. Therefore, if we order rows (ascending) by *id* and *load_timestamp*, we get sets of versions for each object. The first one (with older *load_timestamp*) represents an insertion, and the rest

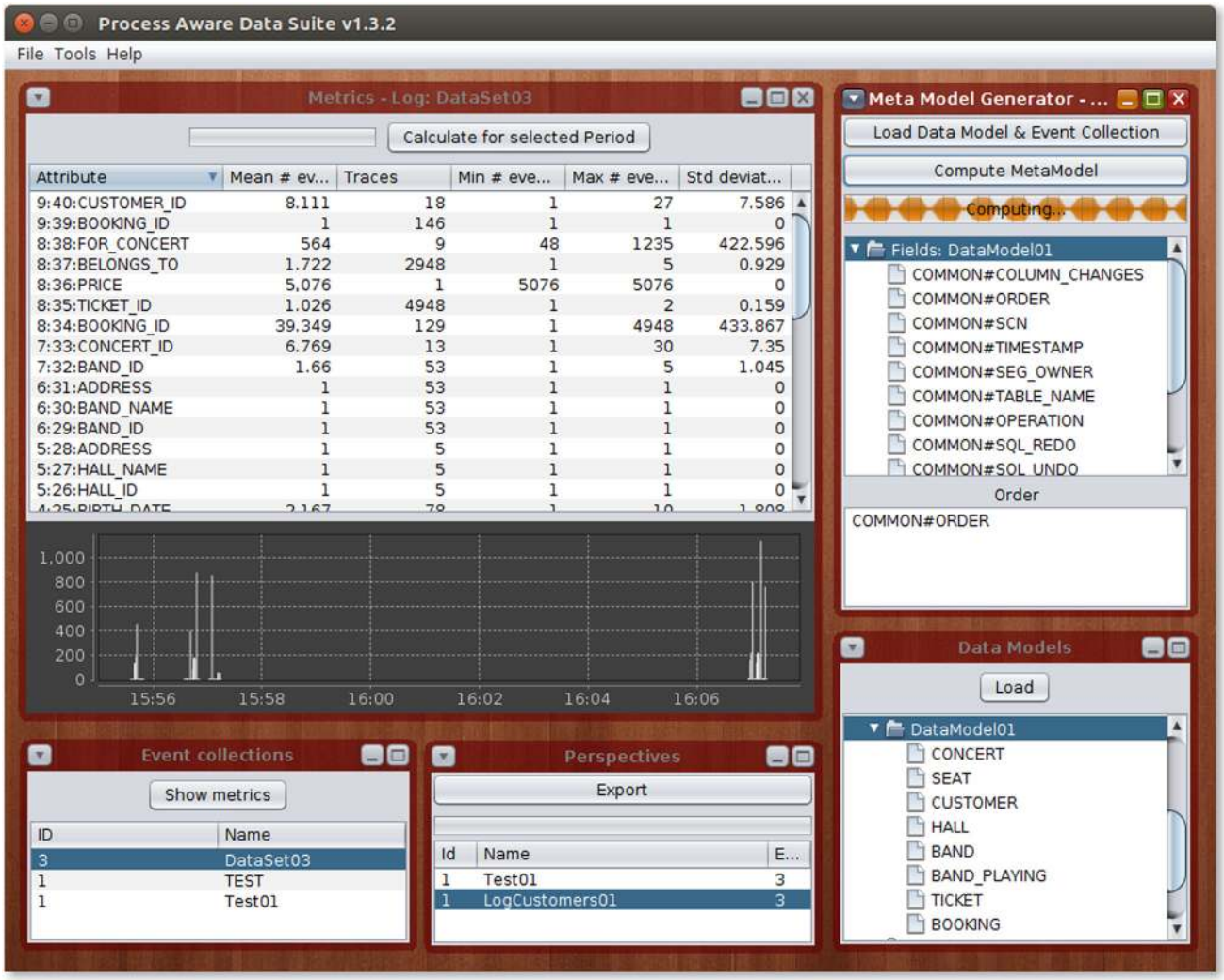

Fig. 9 PADAS tool settings to convert a redo log dataset into a populated meta model

are updates on the values. A more formal description of the mapping from in-table versioning environments to our meta model can be found in Appendix B.3.

5.3.1 In-table versioning: transformation

Looking at Fig. 10, it is clear that, ordering by timestamp the versions in the original set (bottom left), we can reconstruct the different states of the database (right). Each new row in the original table represents a change in the state of the database. Performing this process for all the tables allows inferring the events in a setting where they were not explicitly stored. Figure 6 shows that, thanks to the meta model proposed, it is possible to derive events starting from a data model, a set of objects, and their versions as input (Fig. 7e). The next step is to obtain cases from the events and data model applying the technique from [5] to split event collections into cases selecting an appropriate *Trace ID Pattern* (scenario *c*). Finally, process discovery will allow us to obtain a process model to complete the meta model structure (scenario *f*). A more formal description of the mapping between in-table versioning sources and our meta model can be found in Appendix B.3.

As a result of the whole procedure, we have a meta model completely filled with data (original and derived) that enables any kind of analysis available nowadays in the process of mining field. Moreover, it allows for extended analysis combining the data and process perspectives. Table 4 shows the input/output of the transformation process, together with details of its automation and derivation. We see that, in this environment, the input data correspond to the following entities of the meta model: *data model*, *class*, *attribute* and *version*. These elements are automatically transformed from the input data, where they are explicitly defined. However, the rest of elements need to be either manually obtained from domain knowledge (relationships between classes), or

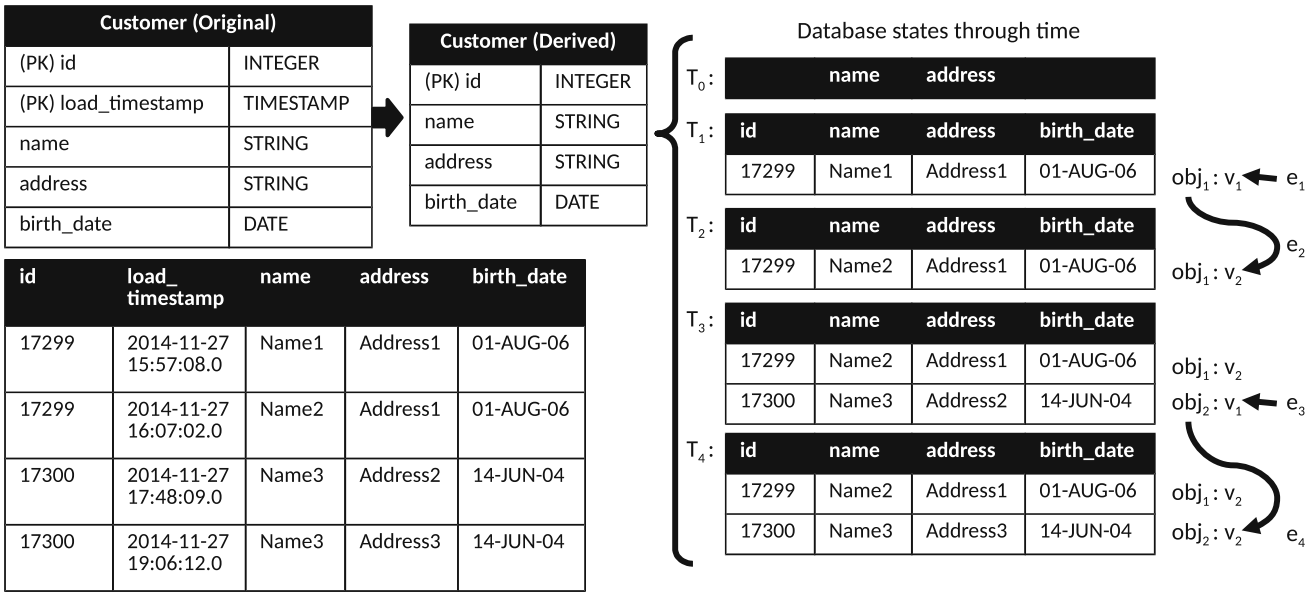

Fig. 10 Example of in-table versioning and its transformation into objects and versions

Table 4 In-table versioning dataset transformation to populate the OpenSLEX meta model

Entity	Format	# Input Els.	# Output Els.	Aut/manual	Derivation
Data Model	CSV	1	1	Aut	Explicit
Class	CSV	8	8	Aut	Explicit
Attribute	CSV	65	65	Aut	Explicit
Relationship	–	0	15	Manual	dom know
Object	–	0	162287	Aut	Inferred
Version	CSV	277107	277094	Aut	Explicit
Relation	–	0	274359	Aut	Inferred
Event	–	0	277094	Aut	Inferred
Activity Instance	–	0	267236	Aut	Inferred
Case	–	0	9858	Aut	Inferred
Log	–	0	1	Aut	Inferred
Activity	–	0	62	Aut	Inferred
Process	–	0	1	Aut	Inferred

automatically inferred from the rest of elements. This is so because, given the configuration of the database, there was no way to query the relationships between tables, as they were not explicitly defined in the DBMS but managed at the application level. Therefore, these relationships needed to be specified by hand after interviewing domain experts on the process at hand.

The resulting data model is depicted in Fig. 11. It is based on the database of a financial organization, focusing on the claim and service processing for a specific financial product. As can be observed, this dataset links information corresponding to customers (*CUSTOMER (3)*), their products (*Product (8)* and *Service_Provition (6)*), incoming and outgoing phone calls (*Call_IN (2)* and *Call_OUT (7)*), letters being sent to customers (*Letter_OUT (4)*), monthly statements from customers (*MONTHLY_STATEMENT (1)*) and customer details change forms (*User_Details_Change_Form_IN (5)*). In further sections, we will show how to obtain a view on this dataset, exploiting the structure of the meta model.

5.3.2 In-table versioning: adapter implementation

The task of transforming the data from an environment that presents an in-table versioning structure into our meta model can be automated by means of an adapter. In this case, we make use of the RapidMiner[10] platform together with the RapidProM[11] extension to implement a workflow that auto-

[10] http://www.rapidminer.com.

[11] http://www.rapidprom.org/.

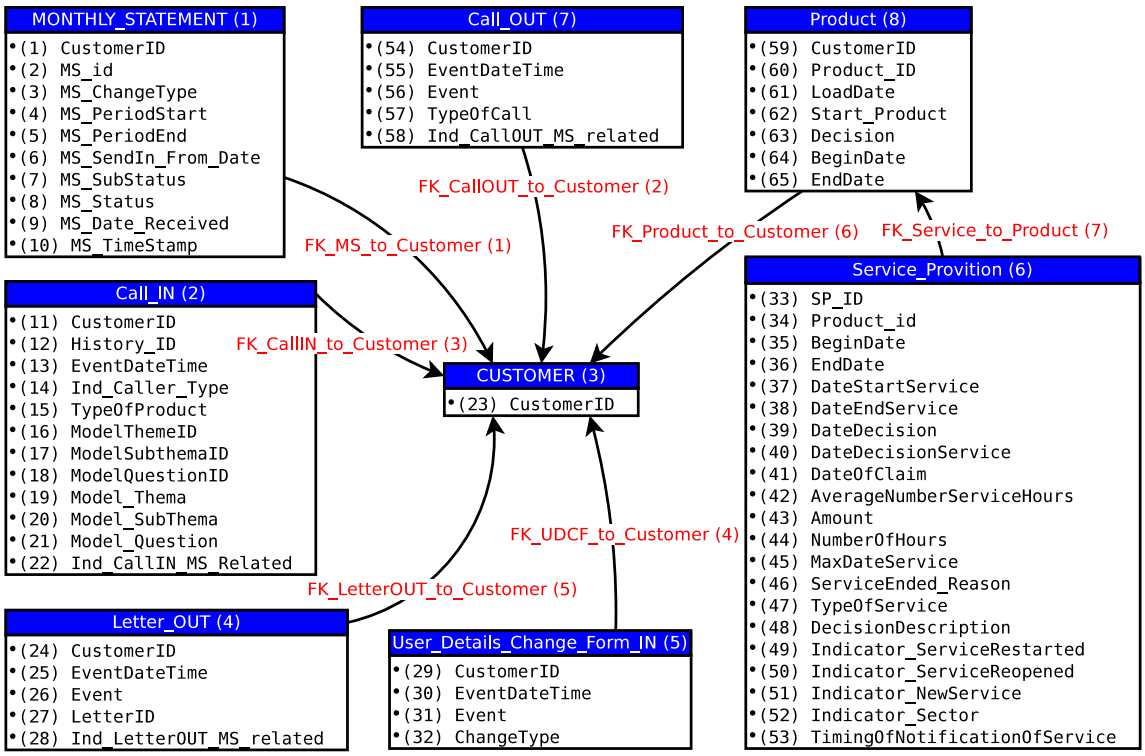

Fig. 11 Data model of the in-table versioning dataset as obtained from the populated meta model

mates this transformation. Specifically, the operator needed for this task can be found in one of the development branches of the RapidProM repository[12]. This adapter, as depicted in Fig. 12, takes as input the set of CSV files that contain the unprocessed dump of the tables to be analyzed directly from the source database. Also, if we get direct access to such database, the data can be queried directly, skipping the transformation to CSV files.

5.4 SAP-style change table

The last environment we will consider for our feasibility study is related to widespread ERP systems such as SAP. These systems provide a huge amount of functionalities to companies by means of configurable modules. They can run on various platforms and rely on databases to store all their information. However, in order to make them as flexible as possible, the implementation tries to be independent of the specific storage technology running underneath. We can observe SAP systems running on MS SQL, Oracle or other technologies, but they generally do not make intensive use of the features that the database vendor provides. Therefore, data relations are often not defined in the database schema, but managed at the application level. This makes the life of the analyst who would be interested in obtaining event logs rather complicated. Fortunately, SAP implements its own redo-log-like mechanism to store changes in data, and it represents a valid source of data for our purposes. In this setting, we lack event logs, object versions, a complete data model and processes. Without some of these elements, performing any kind of process mining analysis becomes very complicated. For instance, the lack of an event log does not allow for the discovery of a process, and without it, performance or conformance analyses are not possible. To overcome this problem, we need to infer the lacking elements from the available information in the SAP database.

First, it must be noted that, despite the absence of an explicitly defined data model, SAP uses a consistent naming system for their tables and columns, and there are lots of documentation available that describe the data model of the whole SAP table landscape. Therefore, this environment corresponds to the scenario depicted in Fig. 7d, where data model, objects and events are available, but object versions need to be inferred. To extract the events, we need to process the change log. This SAP-style change log, as can be observed in Fig. 13, is based on two change tables: *CDHDR* and *CDPOS*. The first table (*CDHDR*) stores one entry per change performed on the data with a unique change id (*CHANGENR*,*OBJECTCLAS*,*OBJECTID*) and other additional details. The second table (*CDPOS*) stores one entry per field changed. Several fields in a data object can be changed at the same time and will share the same change id. For

[12] https://github.com/rapidprom/rapidprom-source/tree/egonzalez.

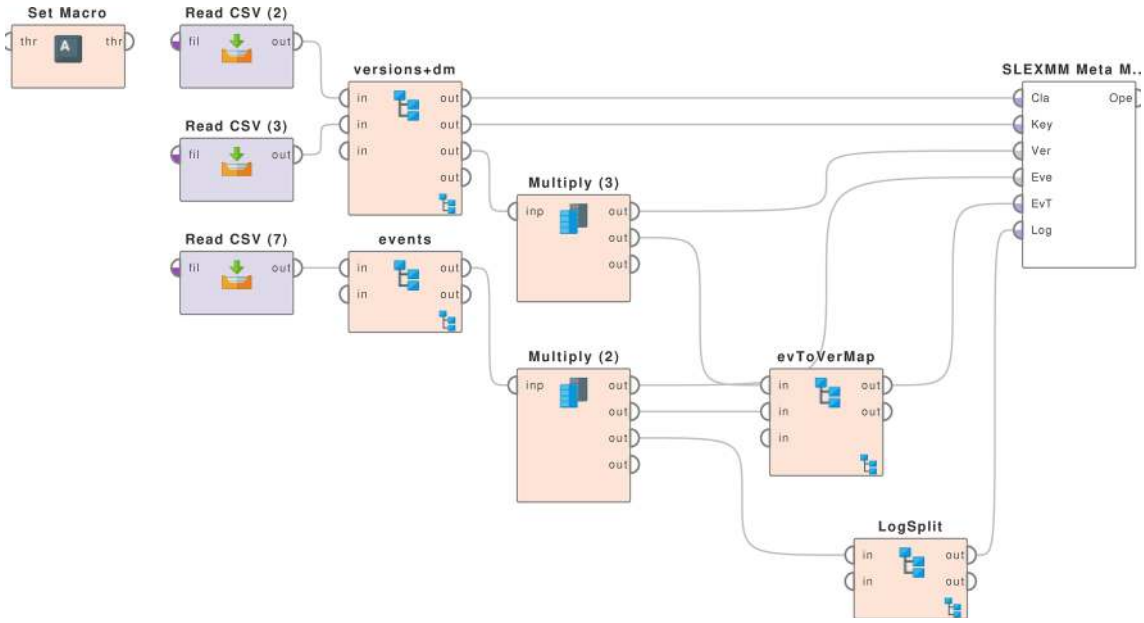

Fig. 12 RapidMiner workflow to convert an in-table versioning dataset in order to populate the OpenSLEX meta model

CDHDR	
OBJECTCLAS	STRING
CHANGENR	INTEGER
OBJECTID	INTEGER
USERNAME	STRING
UDATE	DATE
UTIME	TIME
TCODE	INTEGER

OBJECTCLAS	CHANGENR	OBJECTID	USERNAME	UDATE	UTIME	TCODE
CUST	0001	0000001	USER1	2014-11-27	15:57:08.0	0001
CUST	0002	0000001	USER2	2014-11-27	16:07:02.0	0002
CUST	0003	0000002	USER2	2014-11-27	17:48:09.0	0003
CUST	0004	0000002	USER1	2014-11-27	19:06:12.0	0004
...	...	...	...	...	...	...

Customer	
(PK) id	INTEGER
name	STRING
address	STRING
birth_date	DATE

CDPOS	
OBJECTCLAS	STRING
CHANGENR	INTEGER
TABNAME	STRING
TABKEY	STRING
FNAME	STRING
VALUE_NEW	STRING
VALUE_OLD	STRING

OBJECTCLAS	CHANGENR	TABNAME	TABKEY	FNAME	VALUE_NEW	VALUE_OLD
CUST	0001	CUSTOMER	17299	name	Name1	
CUST	0001	CUSTOMER	17299	address	Address1	
CUST	0001	CUSTOMER	17299	birth_date	01-AUG-06	
CUST	0002	CUSTOMER	17299	name	Name2	Name1
CUST	0003	CUSTOMER	17300	name	Name3	
CUST	0003	CUSTOMER	17300	address	Address2	
CUST	0003	CUSTOMER	17300	birth_date	14-JUN-04	
CUST	0004	CUSTOMER	17300	address	Address3	Address2
...	...	...	...	...	...	...

Fig. 13 Example of SAP change tables CDHDR and CDPOS

each field changed, the table name is recorded (*TABNAME*) together with the field name (*FNAME*), the key of the row affected by the change (*TABKEY*) and the old and new values of the field (*VALUE_OLD*, *VALUE_NEW*). A more formal description of the mapping between SAP change tables and our meta model can be found in "Appendix B.4". This structure is very similar to the one used for redo logs. However, one of the differences is that changes on different fields of the same record are stored in different rows of the *CDPOS* table, while in the redo logs they are grouped in a single operation.

5.4.1 SAP-style change table: transformation

As can be seen in Fig. 6, after processing the change log and providing a SAP data model, we are in a situation in which the events, objects and data model are known. Therefore, we can infer the versions of each object (d) split the events in cases (c) and finally discover a process model (f). With all these ingredients, it becomes possible to perform any process mining analysis and answer complex questions combining process and data perspectives.

Table 5 SAP dataset transformation to populate the OpenSLEX meta model

Entity	Format	# Input Els.	# Output Els.	Aut/Manual	Derivation
Data Model	SQL	1	1	Aut	Explicit
Class	SQL	87	87	Aut	Explicit
Attribute	SQL	4305	4305	Aut	Explicit
Relationship	–	0	296	Aut	Dom know
Object	SQL	7340011	7339985	Aut	Explicit
Version	–	0	7340650	Aut	Inferred
Relation	–	0	7086	Aut	Inferred
Event	SQL	26106	26106	Aut	Explicit
Activity Instance	–	0	5577	Aut	Inferred
Case	–	0	22	Aut	Inferred
Log	–	0	1	Aut	Inferred
Activity	–	0	172	Aut	Inferred
Process	–	0	1	Aut	Inferred

Table 5 shows that, in order to populate our meta model, what we obtain from SAP are the following entities: *data model*, *class*, *attribute*, *object* and *event*. All these elements are explicitly defined and can be automatically transformed. However, the rest need further processing to be inferred from the input data. Only the relationships cannot be inferred, but can be obtained automatically from domain knowledge. The difference here between in-table versioning and SAP is that, despite the need for domain knowledge to obtain the relationships in both scenarios, in the case of SAP we can query the online documentation that specifies the connections between different tables of their data model. To do so, a script[13] has been developed which, from a set of table names, finds and obtains the relationships and stores then in a CSV file.

To get an idea of the complexity of the data model of this dataset, Fig. 14 shows a full picture of it. As can be noticed, it is a very complex structure. However, the tool allows one to zoom in and explore the interesting areas. Figure 15 shows a zoomed area of the data model, where the *EKPO* table points to the *EKKO* table.

5.4.2 SAP-style change table: adapter implementation

As has been mentioned before, SAP systems often use relational databases to store the documents and information they manage. Despite the wide variety of database systems used, the extraction of the relevant information is always possible, as long as a JDBC Driver exists. This allows one to connect to the source database directly from RapidMiner, as shown in the workflow in Fig. 16. In this specific case, the database was SAP Adaptive Server Enterprise (ASE), originally known as Sybase SQL Server. The process is as simple as providing the list of table names we are interested in. The workflow will loop through them (Fig. 17) and extract their content into CSV files that will be processed in a further step by a different workflow.

Once we have the CSV files of the SAP tables, we need to process them. To do so, the RapidProM workflow in Fig. 18 has been developed, which loops through these files (including the SAP change tables) to build our meta model.

5.5 Distributed system merge

The three previous environments have been considered in isolation, showing the ETL process on individual systems. On the other hand, as depicted in Fig. 1, it is not rare that companies' IT landscape is constructed by several systems such as ERPs, CRMs, BPM workflow managers, and so on, which operate separately. These systems store data related to different aspects of the functioning of the company. It is possible that, in order to perform our analysis, we need to make use of data from several of these sources combined, e.g., to link user requests made through a website with internal request handling managed by an internal SAP system. In that case, we need to merge the data coming from these independent systems into a single structure. In order to achieve this goal, we require, at least, one common connecting concept to be shared by these systems.

Figure 19 shows an example of two systems being merged in a single data model. On the left side of the figure, the data model described in Sect. 5.3 is shown as an abstract group of tables from which one of them, the customer table, has been selected. On the right side, the ticket selling platform described in Sect. 5.2 is represented, from which the customer table has been selected as well. Both customer tables (*CUSTOMER_A* and *CUSTOMER_B*) have a customer identification field. Both could share the same ids to represent the same concept. However, that is not needed to be correlated.

[13] https://www.win.tue.nl/~egonzale/createkeysfile-sh/.

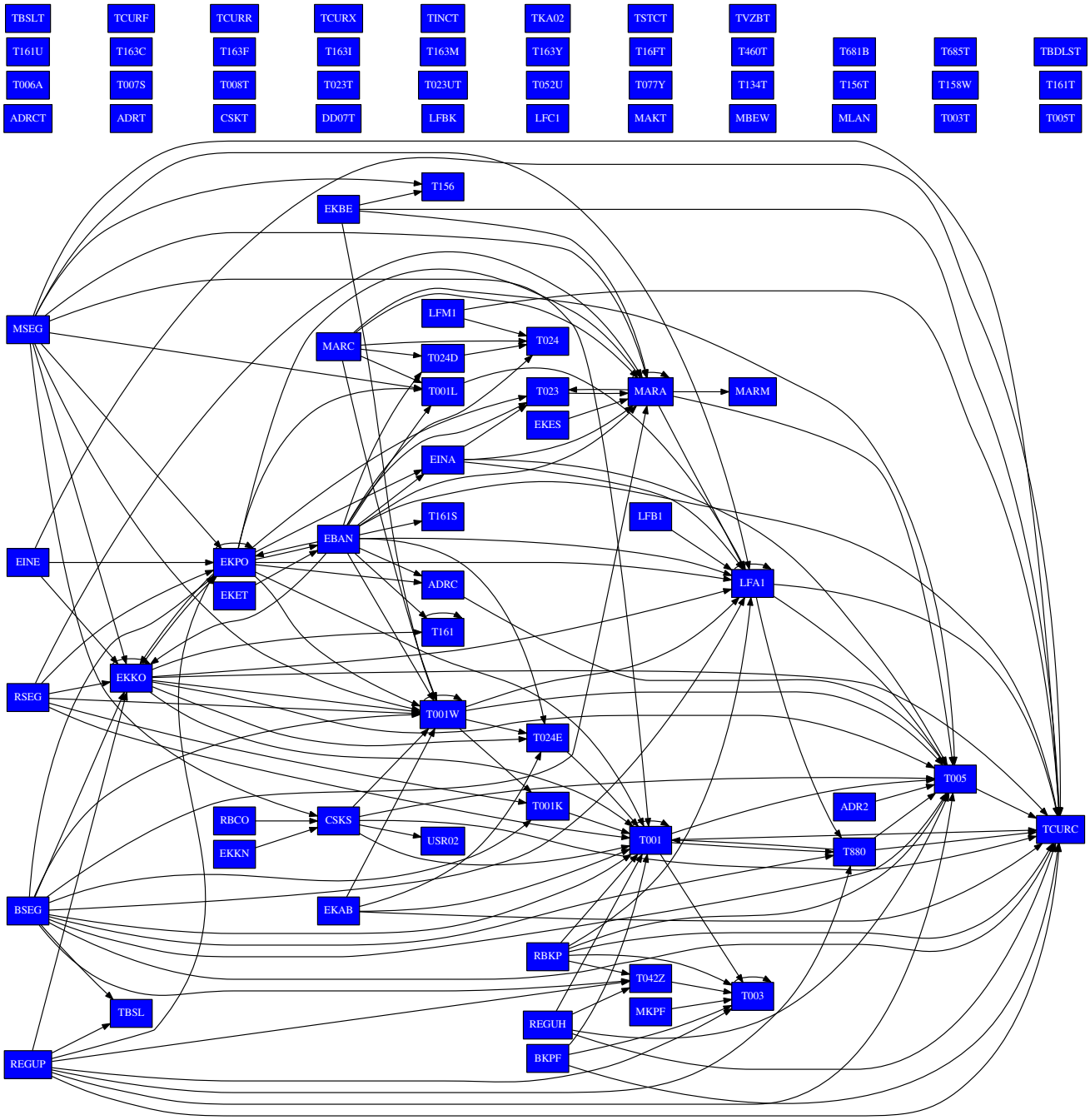

Fig. 14 General view of the data model of the SAP dataset as obtained from the populated meta model. Due to the size of the data model, the attributes or the tables have been omitted from this graph

The merging process requires the creation of an additional linking table (*CUSTOMER_LINK*) which holds the relation between customer ids for both separate data models.

To make the combination of these data models fit the structure of our meta model, it is necessary to make the connection between objects at the object version level. This can be done automatically as long as we know (a) which customer id from table A is related to which customer id from table B, and (b) during which period of time did this relation exist. It can be that the correlation is permanent, for example, when we map customer ids of two systems which always represent the same entity (e.g., social security numbers on one system with national identification numbers on the other). In such a case, the connecting object will always relate to versions of the same two connected objects. This situation is represented in Fig. 20, where an object of class A (top line) is connected

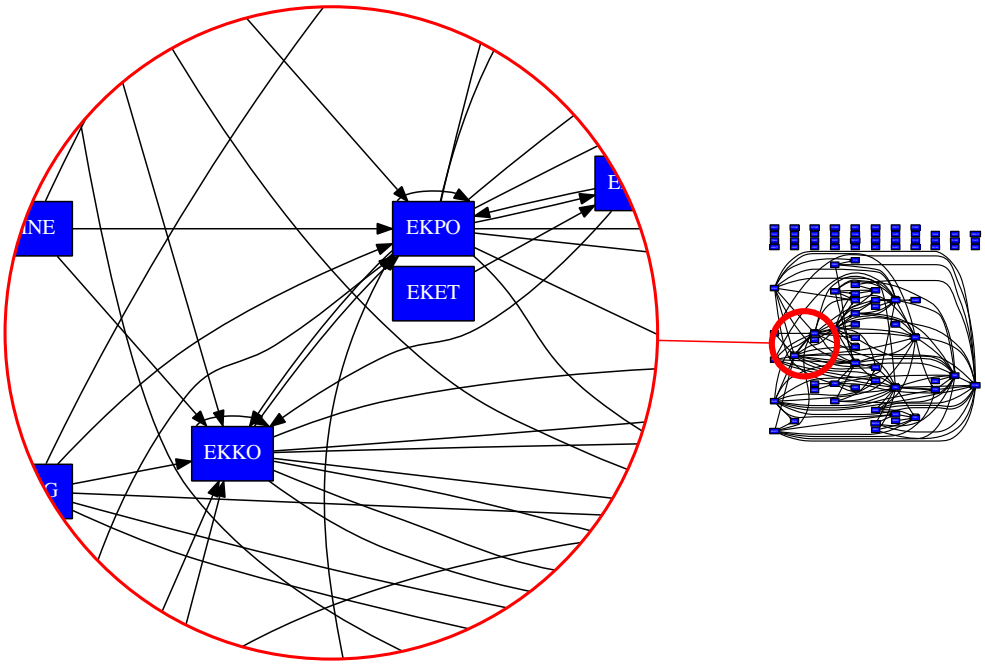

Fig. 15 Detail of the data model of the SAP dataset as obtained from the populated meta model

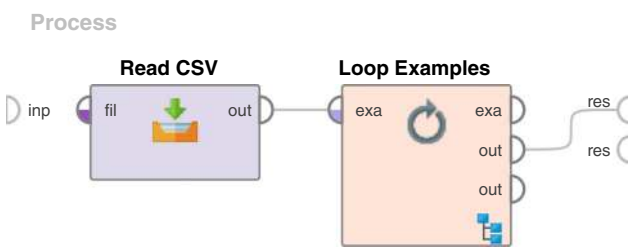

Fig. 16 RapidMiner workflow to connect to a SAP database and extract the content of all the tables

Fig. 17 RapidMiner subprocess to extract each table when connecting to a SAP database

to an object of class B (bottom line) by means of a linking object (middle line). For each pair of coexisting object versions of *objA* and *objB*, a new version of *objL* must exist to relate them. This method has the benefit of providing a great flexibility, being also possible to hold relations between objects that change through time. An example of this is the case in which the connection between two tables represents employer-to-employee relations. An employer can be related to many employees, and employees can change employers at any time. Therefore, the mapping between employer and employee objects will not be permanent through time, but will evolve and change and only exist during a specific period. Such a case is supported by the proposed mapping method, as presented in Fig. 21, by means of new object versions for the linking object.

As demonstrated with this example, the proposed meta model is able to merge information from different systems into a single structure, enabling the analysis of process and data in a holistic way, beyond the boundaries of IT systems infrastructure. Throughout all this section, we have seen how data extraction and transformation into the proposed meta model (Fig. 22) can be performed when dealing with environments of very different nature. It is important to note that, despite its apparent complexity, all the steps previously mentioned are carried out automatically by the provided implementations of the adapters. These adapters can be modified and extended to add support for new environments.

6 Analysis of the resulting populated meta model

The main advantage of transforming all our source information into the proposed meta model structure is that, regardless of the origin of data, we can pose questions in a standard way. Let us consider the three environments described in previous sections: redo logs, in-table versions and SAP systems. In the examples we chose to illustrate the transformation, it is evident that they belong to very different processes and businesses. The redo log example corresponds to a concert ticket selling portal. The second example, based on in-table ver-

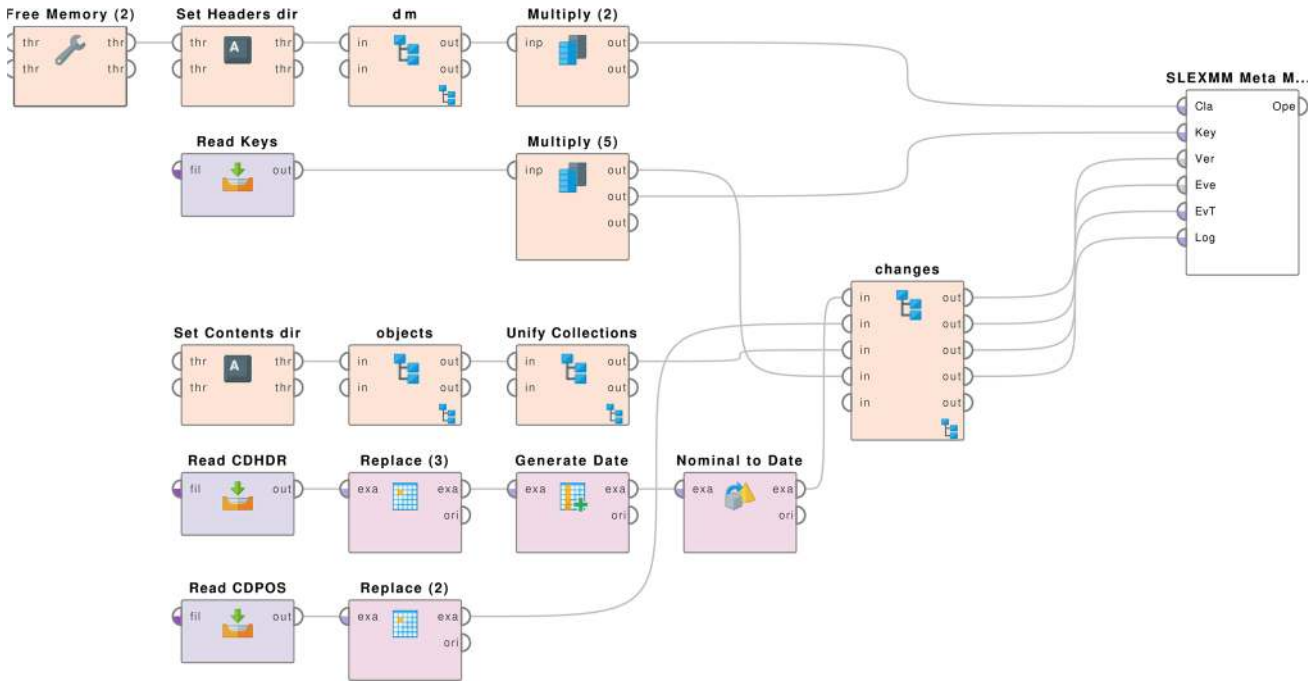

Fig. 18 RapidMiner workflow to populate the meta model based on SAP dataset

sioning, corresponds to the claim and service processing for a financial product within an organization. The third example, based on SAP system, represents the procurement process of a fictitious company. Due to the diversity of data formats and designs of the systems, in a normal situation these three environments would require very different approaches in order to be queried and analyzed. However, we claim that our meta model provides the standardization layer required to tackle these systems in a similar manner.

6.1 Standardized querying

Our goal is to query the data from different sources in a standardized fashion. To do so, as demonstrated in Sect. 5, we extracted the raw input data from the databases of each system. Then, our automated workflows were used to transform these data and obtaining, as a result, a structure compliant with the proposed meta model. In this way, we hope to make plausible that, given a specific business question, it is possible to make a SQL query that answers this question on any populated meta model, regardless of the process it represents or the original format of the data. We demonstrate our claim with the following example. Let's assume we want to find the group of cases in our logs that comply with a specific rule, or represent a part of the behavior we want to analyze. As an example, for the redo log dataset we could define the following Business Question (BQ1):

BQ1: In which cases, the address of a customer was updated?

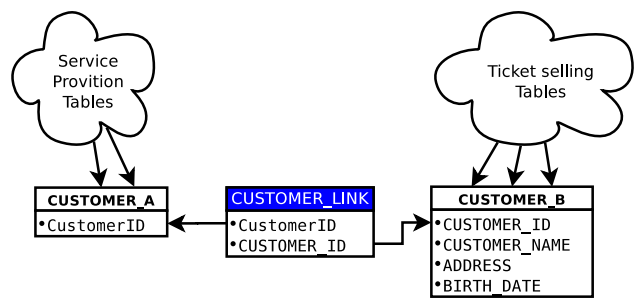

Fig. 19 Link between different data models through tables representing a common concept (*CUSTOMER_A* and *CUSTOMER_B*) by means of a link table (*CUSTOMER_LINK*)

To do this in the original data would require to go through the redo log, looking for events on the table *CUSTOMER*, link them to the specific *customer_id*, correlate the other events of the same customer, group them in traces and build the log. This is not an easy task, especially if such a complicated query needs to be built specifically for each new business question that comes to mind. Let us consider another example, now for the in-table versioning dataset (BQ2):

BQ2: In which cases, a service request was resolved?

To answer this, we need to go through the table *product* and check in which row the value of the field *decision* changed respect to the previous one, having the same *customer_id* in common. Then, correlate events from any other table through the same *customer_id* and build the log again. This requires a sequence of non-trivial and time-consuming

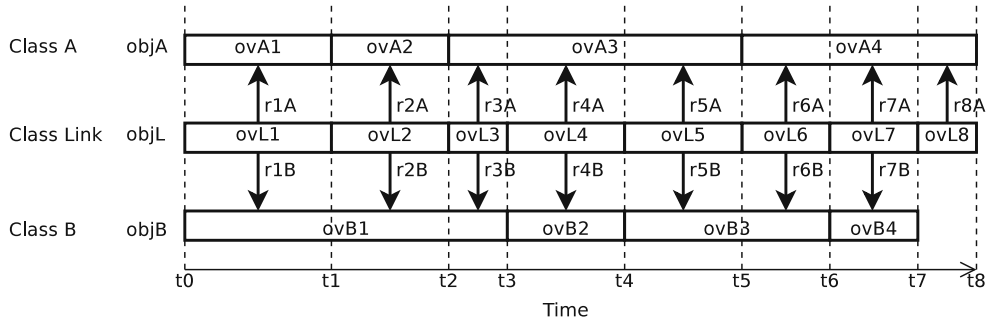

Fig. 20 Merging method to map versions of objects belonging to classes of different data models (class A and class B) through a linking class (class Link)

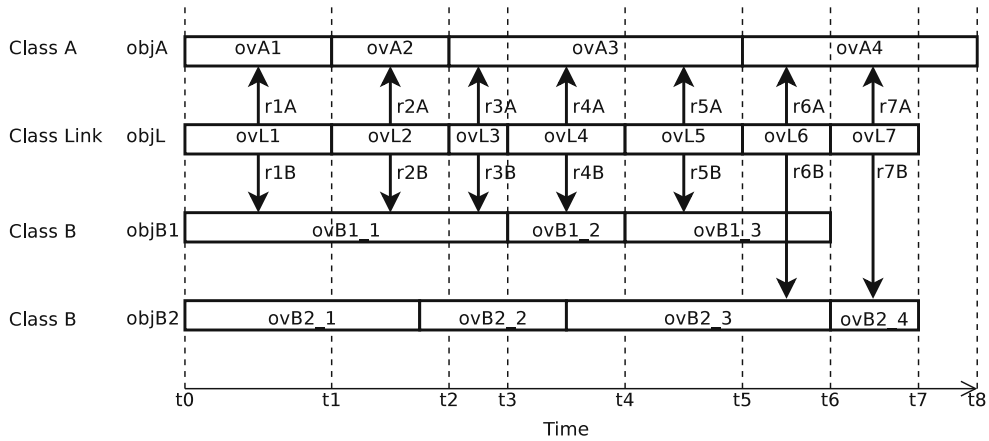

Fig. 21 Merging method to map versions of multiple objects belonging to classes of different data models (class A and class B) through a linking class (class Link). In this case, the related objects change through time

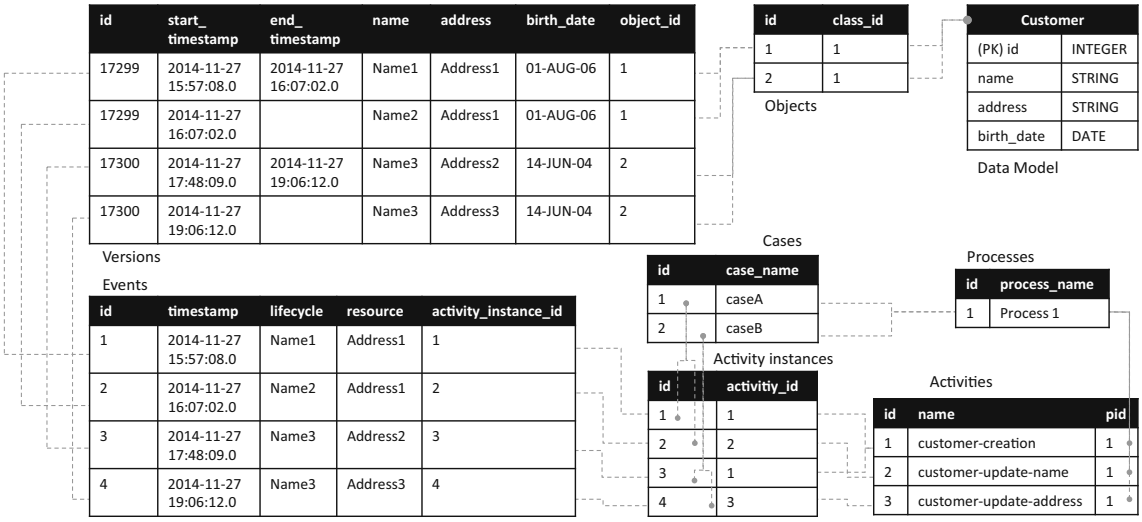

Fig. 22 Fragment of resulting populated meta model

steps. As a final example, we could think of the following question to ask for the SAP dataset (BQ3):

> **BQ3:** In which cases, the quantity of a purchase requisition order was modified?

This time we need to go to the *EKPO*[14] table, which contains information about purchasing document items. However, unlike the in-table versioning case, the previous values of the items are not in the original table anymore. To find historical values for each field, we need to go through the CDHDR and CDPOS tables to find a row that corresponds to the modification of the field *MENGE* (purchase order quantity). To do so, we must look at the value of the field *FNAME*, that should be equal to the string *"MENGE"*, and the value of the field *TABNAME*, that should be equal to the string *"EKPO"*. Then, match it with the correct Purchase Order (sharing the same *TABKEY*) and correlate it to any other event (from the *CDHDR* and *CDPOS* tables) that affected the same purchase orders and group them in cases to form a log. Doing so we can see the changes in context. We see that, as the complexity of data increases, the process to query it increases as well.

Therefore, to summarize, we want to ask three different questions in three different environments, and each question needs to be answered in a specific way. However, we claim that we can derive a standard way to answer questions of the same nature on different datasets, as long as the data have been previously normalized into our meta model. If we look at the three proposed business questions (BQ1, BQ2 and BQ3), all of them have something in common: they rely on identifying cases in which a value was modified for a specific field. To complicate things a bit more, we can even filter the cases to obtain only the ones that cover a specific period of time. This means that we can generalize these questions into one common general question (GQ):

> **GQ:** In which cases (a) there was an event that happened between time T1 and T2, (b) that performed a modification in a version of class C (c) in which the value of field F changed from X to Y?

This is a question we can answer on the basis of our meta model, regardless of the dataset it represents. All we need to do is to specify the values of each parameter (T1, T2, C, X and Y) according to the data model at hand. We translated the general question GQ into the SQL query in Listing 2.

Listing 2 Standard query executed on the three populated meta models

```
SELECT C.id as "T:concept:name",
    E.timestamp as "E:time:timestamp",
    AC.name as "E:concept:name",
    E.resource as "E:org:resource",
    E.lifecycle as "E:lifecycle:transition",
    E.ordering
FROM
    event as E,
    "case" as C,
    activity_instance as AI,
    activity_instance_to_case as AITC,
    activity as AC
WHERE
    C.id = AITC.case_id AND
    AITC.activity_instance_id = AI.id AND
    E.activity_instance_id = AI.id AND
    AI.activity_id = AC.id AND
    C.id IN
    (
        SELECT C.id
        FROM
            "case" as C,
            class as CL,
            object as O,
            object_version as OV,
            object_version as OVP,
            event as E,
            activity_instance as AI,
            activity_instance_to_case as AITC,
            event_to_object_version as ETOV,
            attribute_name as AT,
            attribute_value as AV,
            attribute_value as AVP
        WHERE
            E.activity_instance_id = AI.id AND
            AITC.activity_instance_id = AI.id AND
            AITC.case_id = C.id AND
            ETOV.event_id = E.id AND
            ETOV.object_version_id = OV.id AND
            OV.object_id = O.id AND
            O.class_id = CL.id AND
            CL.name = "%{CLASS_NAME}" AND
            E.timestamp > "%{TS_LOWER_BOUND}" AND
            E.timestamp < "%{TS_UPPER_BOUND}" AND
            AT.name = "%{ATTRIBUTE}" AND
            AV.attribute_name_id = AT.id AND
            AV.object_version_id = OV.id AND
            AV.value LIKE "%{NEW_VALUE}" AND
            AVP.attribute_name_id = AT.id AND
            AVP.object_version_id = OVP.id AND
            AVP.value LIKE "%{OLD_VALUE}" AND
            OVP.id IN
            (
                SELECT OVP.id
                FROM object_version as OVP
                WHERE
                    OVP.start_timestamp < OV.start_timestamp AND
                    OVP.object_id = OV.object_id
                ORDER BY OVP.start_timestamp DESC LIMIT 1
            )
    )
ORDER BY C.id, E.ordering;
```

This query is standard and independent of the dataset thanks to the use of macros (RapidMiner macros) denoted by *%{MACRO_NAME}*. We just need to instantiate their values according to the dataset to process. Table 6 shows the values for each macro to be replaced in each case. Notice that the timestamps (*TS_LOWER_BOUND* and *TS_UPPER_BOUND*) are expressed in milliseconds, and the string % in *NEW_VALUE* and *OLD_VALUE* will match any value.

This query can be easily executed in the RapidMiner environment, using the *Query Database* operator. Figure 23 shows the workflow executed to set the macro-values and make the query for each dataset. Some details of the logs obtained for each of the three datasets can be observed in Table 7.

[14] http://www.sapdatasheet.org/abap/tabl/EKPO.html.

Table 6 Parameters to query the three different populated meta models with the same query

Variable	Value-RL	Value-ITV	Value-SAP
CLASS_NAME	CUSTOMER	Product	EKPO
TS_LOWER_BOUND	527292000000	527292000000	527292000000
TS_UPPER_BOUND	1480531444303	1480531444303	1480531444303
ATTRIBUTE	ADDRESS	Decision	MENGE
NEW_VALUE	%	Toekenning	%
OLD_VALUE	%	Nog geen beslissing	%

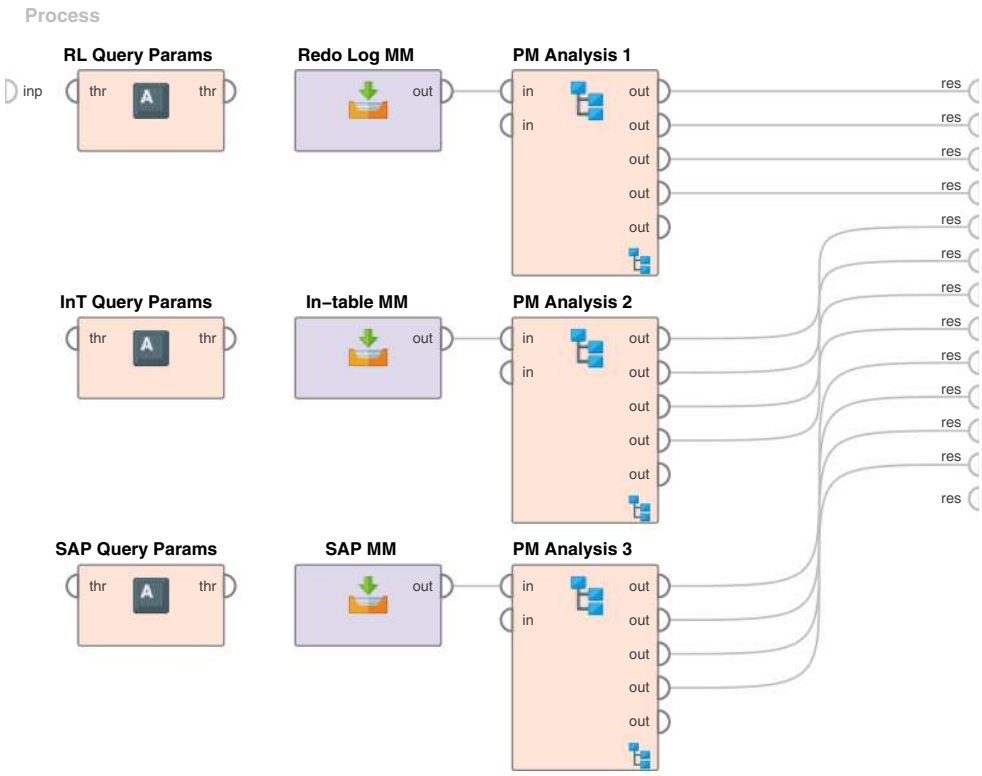

Fig. 23 Analysis workflow to process the three resulting populated meta models

Table 7 Query results for the three different populated meta models

Properties	Redo log	In-table Ver.	SAP
# Cases	17	3177	1
# Events	301	69410	12
# Event classes	5	50	11

It is important to notice that this technique does not exclude the classical way to analyze logs. On the contrary, it enables the use of all the existing process mining techniques. A proof of it is the workflow in Fig. 23, which executes the subprocess in Fig. 24 for each of the resulting logs. As can be observed, this process mining task transforms the log table into a XES log. Then, a process tree is discovered using the Inductive Miner, which transforms it into a Petri Net. Then, a performance analysis is done on the discovered model and the log, together with a conformance checking. In addition to that, a more interactive view of the process is obtained with the Inductive Visual Miner. A social network (when the resource field is available) can be discovered using the Social Network Miner. All these analyses are performed automatically and with exactly the same parameters for each resulting log. It is completely independent of the source, which means that, just by modifying the values of the macros of our query, we can standardize the process of analyzing sublogs for value modification cases.

6.2 Process mining results

To continue with the demonstration of viability of this technique, in this section we show the results of the automated analysis performed on the three different datasets by means of the same query and same workflow.

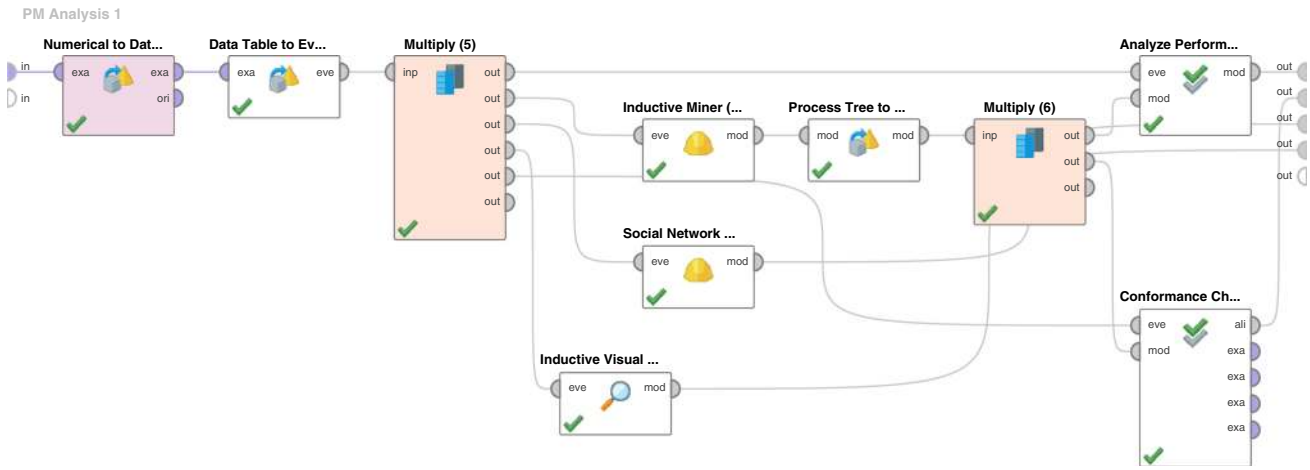

Fig. 24 Process mining analysis subprocess included in the analysis workflow of the three populated meta models. It illustrates that queries on the populated meta models can be converted into event logs that can be analyzed by a range of process mining algorithms

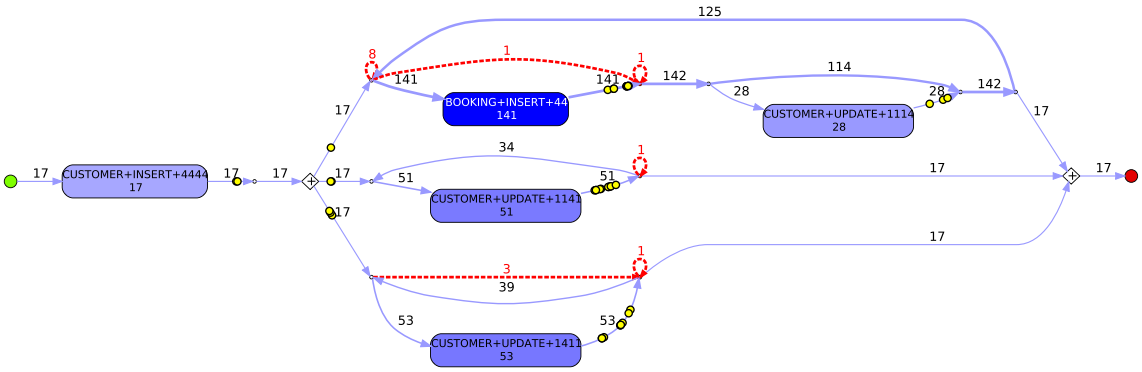

Fig. 25 Discovered model and deviations for the redo log dataset

6.2.1 The redo log environment: ticket selling process

In the case of the ticket selling process, we are interested in the traces that modified the address of a customer. First, the discovered model is shown in Fig. 25. We see five main activities in this model: customer insertions, booking insertions, and three different kinds of customer updates. One can notice that the activity *CUSTOMER+UPDATE+1411*, which corresponds to updates in the address field, is the most frequent of the three modifications, only behind the activity *BOOKING+INSERT+44* activity. One can also see that, as expected, a customer record can only be modified when it has been inserted before. The dashed lines in the model represent deviations from the discovered model, usually representing infrequent cases that were filtered out during the mining phase.

Another technique that can be used during the execution of the automated workflow is the performance analysis. To do so, the log is replayed on the discovered model, and time differences are computed between consecutive events. Then, this information is displayed on top of the original model to visualize bottlenecks and performance issues. In this case, Fig. 26 shows that the most time-consuming activity is one of the customer updates. However, this information can be misleading since these modifications were atomic, while data about start and complete events are missing. To have a better view on performance metrics and accurate task duration, the life-cycle attribute should be properly assigned to pair events in activity instances. This is done automatically by the transformation adapter. However, in this case, there are no activities to be paired.

To finalize the analysis of this process, Fig. 27 shows the result of checking the conformance of the log respect to the process. Here we see that, framed in red, activities *CUSTOMER+UPDATE+1411* and *BOOKING+INSERT+44* show move on log deviations. However, as the green bar at the bottom and the number in parentheses show, these deviations occur in a very small share of the total cases. The rest of the activities are always executed synchronously according to the model.

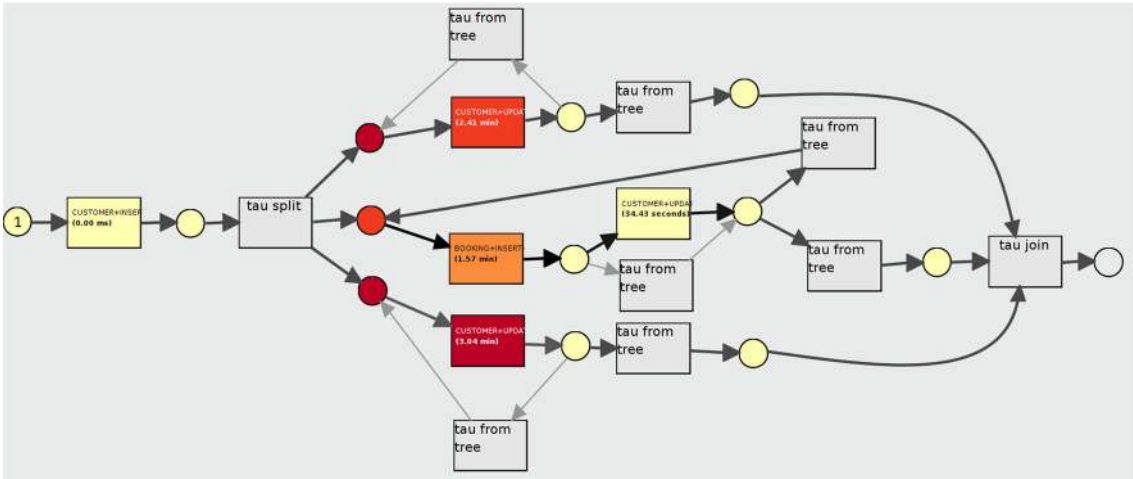

Fig. 26 Performance analysis of the model for the redo log dataset

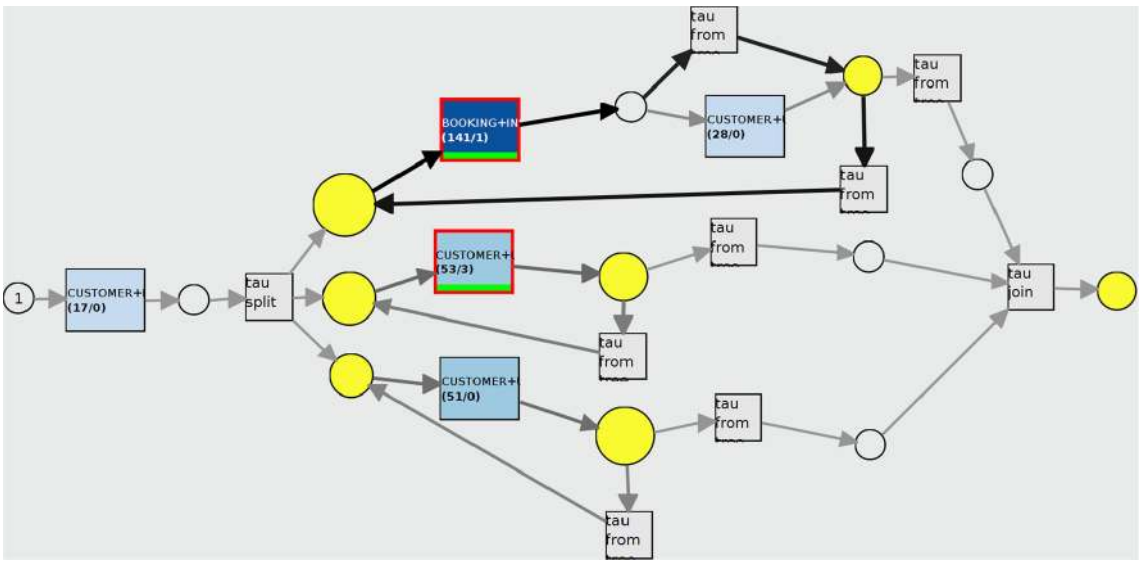

Fig. 27 Conformance analysis of the model for the redo log dataset

6.2.2 The in-table versioning environment: claim management process

The second environment corresponds to the claim management process of a financial organization. The result of our query represents the cases in which the value of the *Decision* field of product service claim changed from *Undecided* (*Nog geen beslissing*) to *Granted* (*Toekenning*). Figure 28 shows the model as obtained from Inductive Visual Miner. It can be observed that the process starts with a *Monthly Statement* being initialized. Then, for most of the following steps three possibilities exist: the activity is executed, the activity is re-executed (in a loop), or the activity is skipped. The frequencies in the arcs demonstrate that, in most of the cases, each subsequent activity happens at least once. These activities consist of the *Reception* and *Checking* of *Monthly Statements*, *Incoming calls*, *Outgoing letters*, *Changes in user details* and *Outgoing calls*. Finally, the claim is resolved and three outcomes are possible: *Not decided*, *Accepted* or *Rejected*.

When looking at the performance view in Fig. 29, we notice that most of the places of the net are colored in red. This means that the waiting time in this places is higher than the rest. It makes sense since, for some automatic activities, the waiting time between tasks will be very low, of the order of milliseconds or seconds, and those places will set the lower bound for the performance scale. On the other hand, for most of the human-driven activities, the times will be much longer, of the order of days. With respect to activities, we see some variability, having some very costly activities, especially at the beginning. These activities are mainly the ones related to

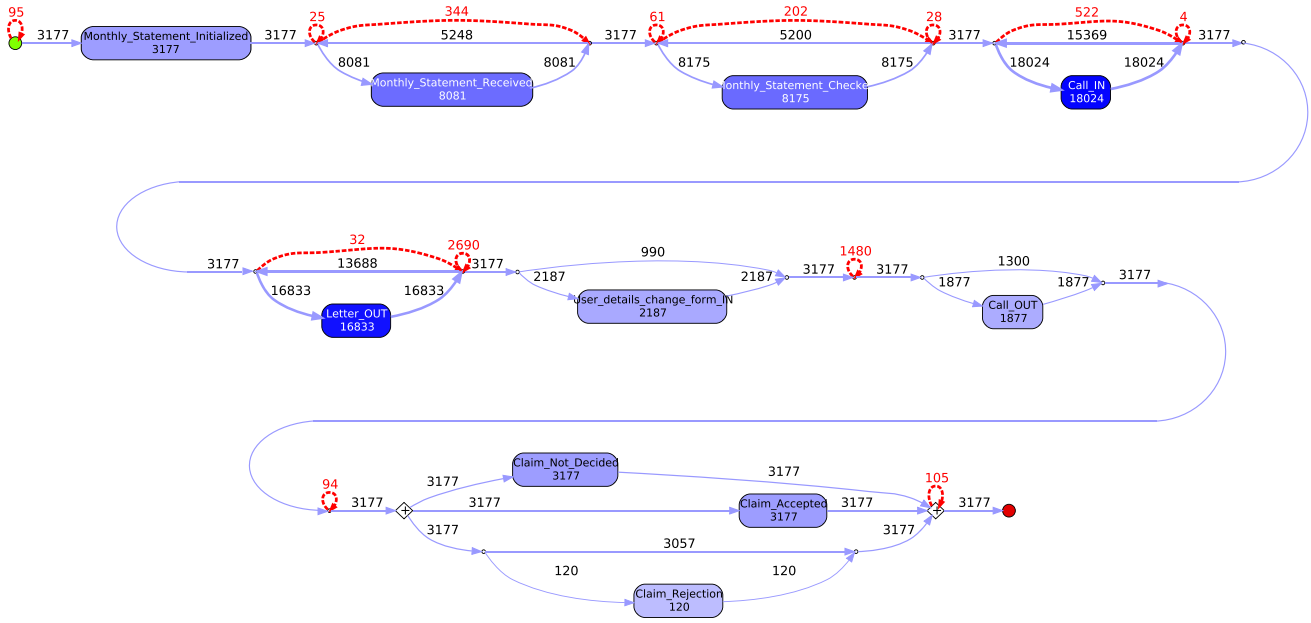

Fig. 28 Discovered model and deviations for the in-table versioning dataset

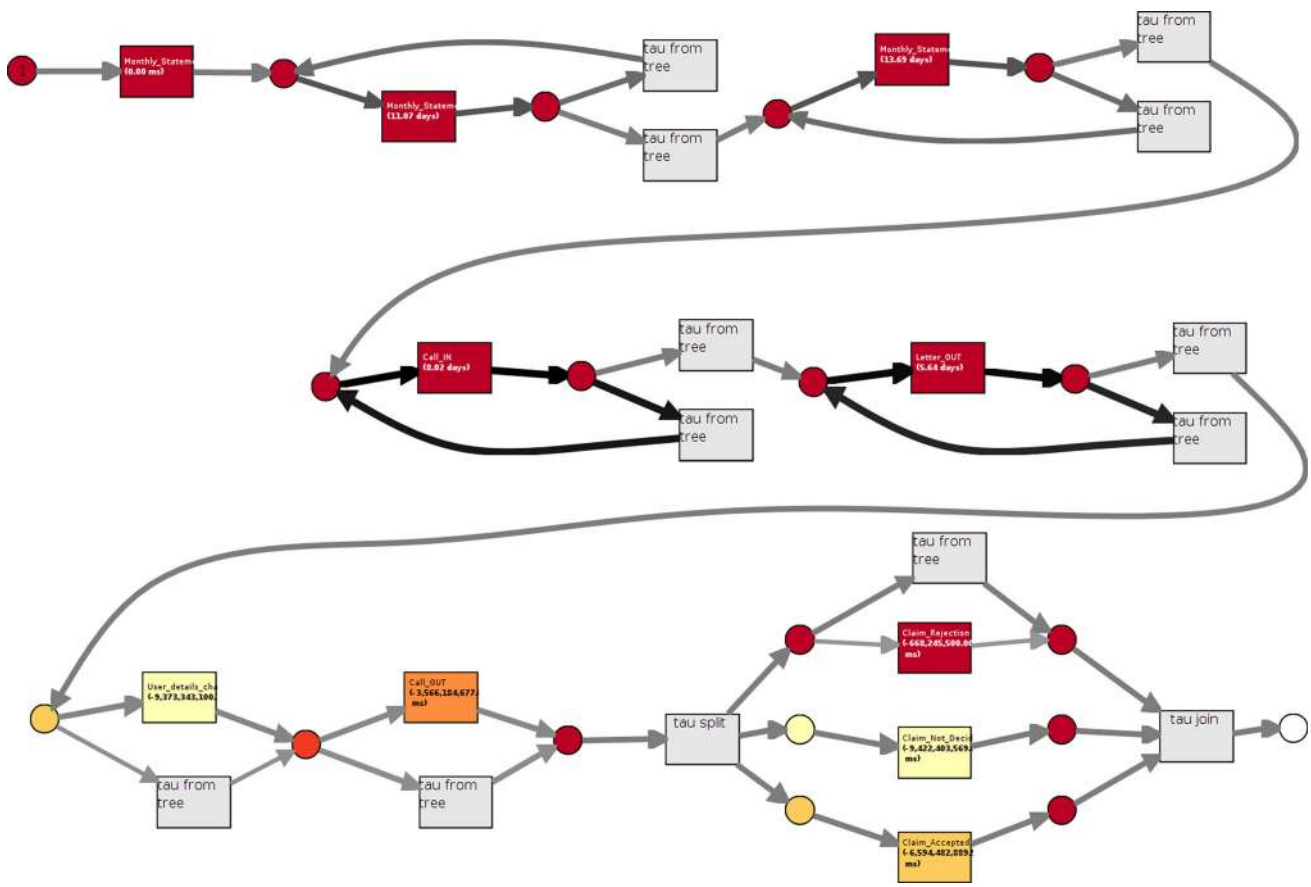

Fig. 29 Performance analysis of the model for the in-table versioning dataset

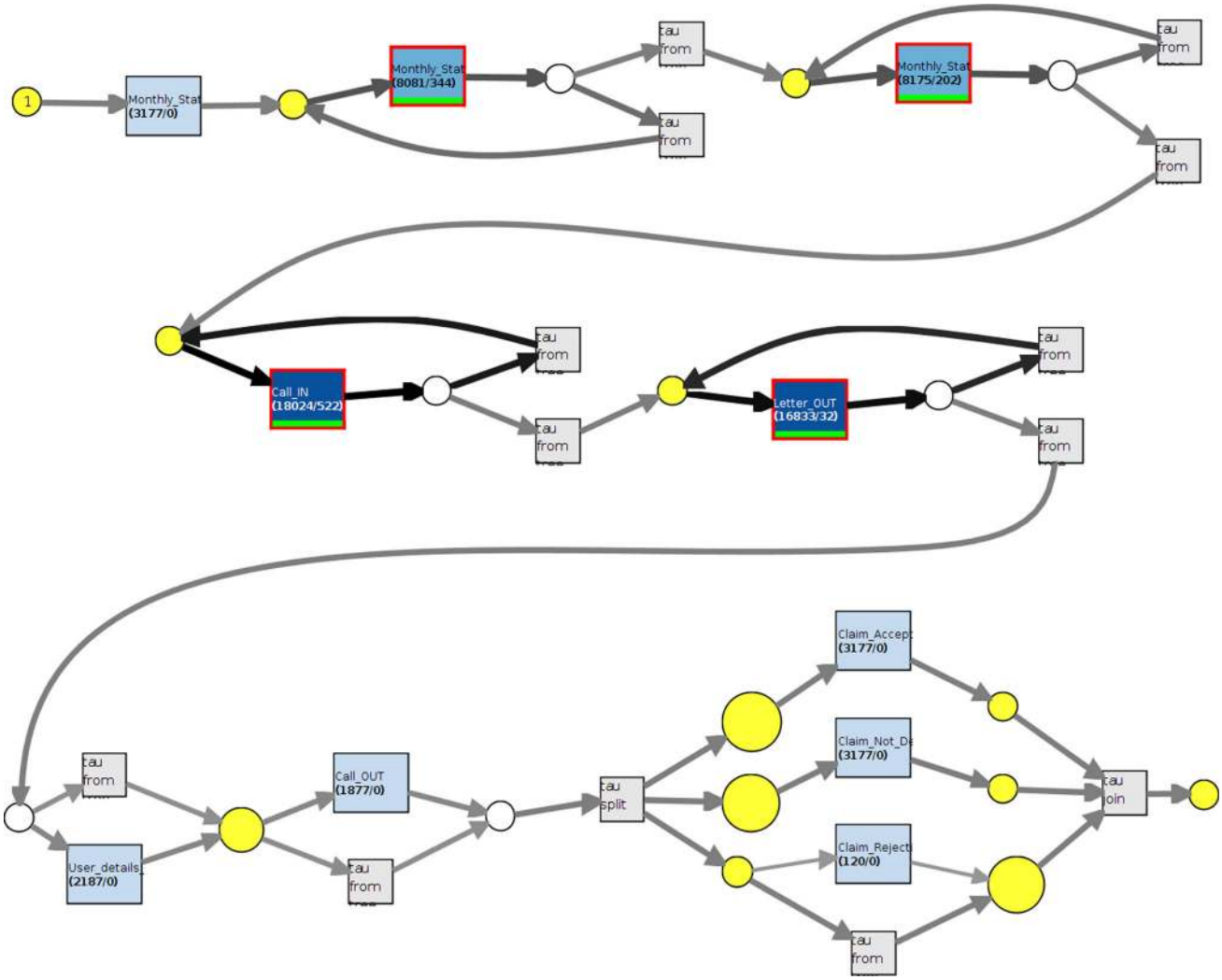

Fig. 30 Conformance analysis of the model for the in-table versioning dataset

processing of *Monthly Statements*, *Incoming calls* and *Outgoing letters*.

Figure 30 shows some conformance results. Here we can observe that most of the activities happen without significant deviations. However, some model moves are observed, but in a very low rate, mainly because of the infrequent skips explained previously in Fig. 28.

6.2.3 The SAP environment: procurement process

Finally, we focus on the Procurement process of an SAP environment. We are particularly interested in the cases in which the quantity of a purchase order was modified. After querying the populated meta model and obtaining the relevant cases, we used Inductive Visual Miner to get a model of the process. Figure 31 shows a very structured process, which is not strange given the few cases that fulfilled our criteria.

It seems clear that some activities are executed repeatedly because of the loops. The process always starts with an update in the value of the *Effective Price in Purchasing Info Record* (*INFOSATZ_U_EFFPR*). This is followed by updates in the *Purchasing Document* (*EINKBELEG* object class), specifically in the *Purchase order not yet complete* field (*MEMORY*). Only then, the *Purchase Order Quantity* (*MENGE*) field is updated.

According to Fig. 32, the performance seems evenly distributed, except for the update in *Purchase Order Quantities*, which seems to take a shorter time than the rest. From the conformance point of view (Fig. 33), we see that the process does not show deviations. This was expected, given the size of the log and the existence of only one case with this particular behavior.

Something interesting to note in this dataset with respect to the other environments is the existence of *resource* information. This means that we know who performed a change

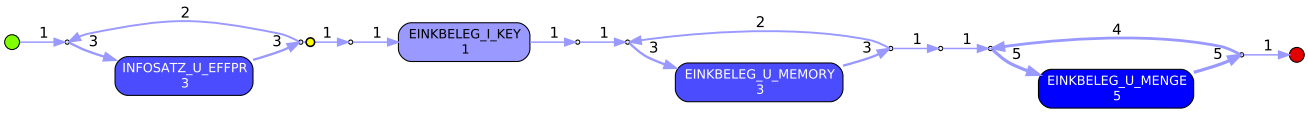

Fig. 31 Discovered model and deviations for the SAP dataset

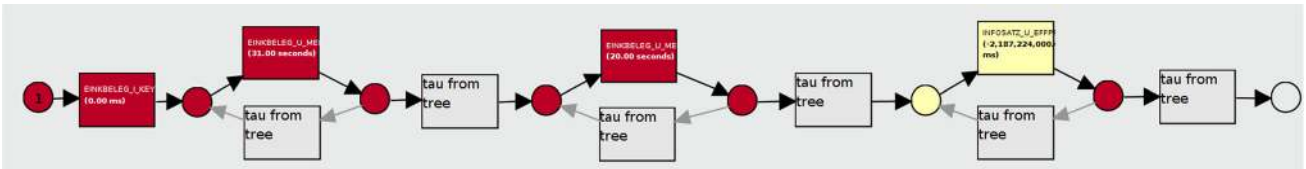

Fig. 32 Performance analysis of the model for the SAP dataset

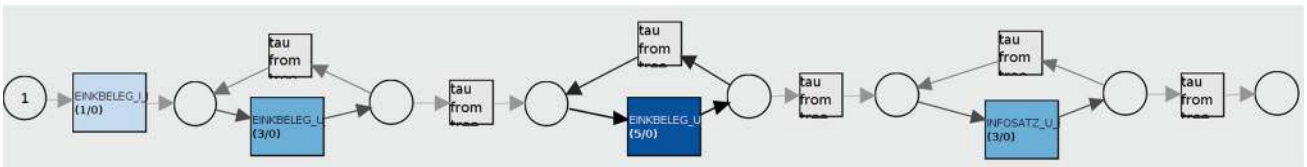

Fig. 33 Conformance analysis of the model for the SAP dataset

Fig. 34 Social network for the SAP dataset

in the documents in SAP. This can be exploited to analyze the data discovering the social network behind the interactions. In this case, Fig. 34 represents the discovered social network, showing the handout of work between resources *UJBSSEN* and *EHANI*.

7 Related work

This paper is based on our previous work [4] in which we already proposed the same meta model. However, in this paper, a special emphasis is put on the applicability of the approach and the standardization of the analysis. For this, we provide a formal description of the mapping between real-life system and our meta model. Moreover, we provide an implementation of all our techniques, making possible to transform the data to populate our meta model, capturing the data and process perspectives of the system under analysis. This makes possible to query it in a standard way, obtaining the relevant cases for the business question at hand, in order to proceed with the analysis. Finally, a demonstration of the analysis is made, covering all the phases of the process, starting from the data extraction, and continuing with data transformation, querying, process discovery, conformance and performance analysis.

Despite the efforts made by the community, we can identify a few areas in which the problem of modeling, collecting and analyzing process execution data remains a challenge. **Business process management** and **workflow management** are areas in which the community has focused on providing models to describe their functioning and make possible the analysis of their behavior. Papers like [16,27] provide meta models to give structure to audit trails on workflows. However, they focus mainly on the workflow or process perspective.

Process mining has different needs and the desire to store event data in a unified way is obvious. In [19], the authors provide a meta model to define event logs, which would evolve later in the IEEE XES Standard format [7]. This format represents a great achievement from the standardization point of view and allows exchanging logs and developing mining techniques assuming a common representation of the data. However, the current definition of XES is not able to effectively and efficiently store the data perspective of our meta model. From our point of view, the XES format is, in fact, a target format and not a source of information. We aim at, from a richer source, generating different views on data in XES format to enable process mining.

Artifact-centric approaches provide a different point of view. These approaches give more relevance to the data entities in business management systems. Data artifacts are identified within business process execution data, in order to discover how changes and updates affect their life cycle [15]. Techniques like the one proposed in [9] are able to combine

the individual life cycles of different artifacts to show the interrelation of the different steps. However, a limitation of these techniques is the difficulty to interpret correctly the resulting models, in which clear execution semantics are not always available. The application of other techniques like conformance and performance analysis to the resulting models has not been solved yet.

Data warehousing is an area in which there is a great interest to solve the problem of gathering business process information from different sources. Work has been done on this aspect [3,14,26,28,29], proposing ways to centralize the storage of heterogeneous data, with the purpose of enabling process monitoring and workflow analysis. However, some of these approaches, besides being process-oriented, do not allow applying process mining analysis. This is due to the lack of event data, when measures of process performance are stored instead. On the other hand, when they allow storing more detailed information [13], force an ad hoc structure, only relevant for the process at hand. The main distinction between our approach and existing work on data warehousing for process mining is the generality and standardization of our meta model, being independent of the specific process, while combining both data and process perspectives.

Process cubes [20] are techniques, closely related to data warehousing, that aim at combining aspects of multi-dimensional databases (OLAP cubes) with event logs, in order to enable the application of process mining techniques. Further work, with functional implementations, has been presented in [1,22–24]. These approaches allow one to slice, dice, roll up and drill-down event data using predefined categories and dimensions. It represents a great improvement for the analysis of multi-dimensional data. However, these techniques still rely on event data as their only source of information, and they require any additional data to be part of each individual event. Therefore, given that it is impossible to efficiently represent in an event log all the aspects we cover in our meta model, process cubes as they are right now do not represent an alternative for our purposes.

SAP Process Observer is a component of the wide spectrum of SAP utilities. It makes possible to monitor the behavior of some SAP Business Suite processes (e.g., order to cash). This component monitors Business Object (BO) events and creates logs correlating them. The tool provides insights such as deviation detection, real-time exception handling, service level agreement tracking, etc. These event logs can be used as well to perform other kinds of analytics. One of the main advantages of this solution is the fact that it enables real-time monitoring, without the need to deal with raw data. However, it needs to be configured specifying the activities that must be monitored. The kind of logs that it generates is no different of the event logs provided by other tools or techniques, in the sense that they use the same structure, having events grouped in sets of traces, and data only represented as plain attributes of events, traces and logs. Also, it lacks genericity, being only applicable in SAP environments to monitor Business Suite processes.

Nowadays, several **commercial process mining** tools are available on the market. These tools import either XES event logs (e.g., Disco[15]), or event tables (e.g., Celonis[16]), supporting attributes at the event, trace and log levels. Most of them provide advanced filtering features based on these attributes, as is the case of Disco. Additionally, Celonis has two interesting features. First, it implements OLAP capabilities, making possible to perform some operations like slicing and dicing event data. Second, Celonis provides its own language to express formulas. These formulas are used to compute statistics and key performance indicators (KPIs) and to express complex conditions for OLAP operations. All these features are remarkable. However, they are restricted by the flat structure of their input event log format. It is not possible to keep complex data relations in such a restricted representation. These complex relations are needed to perform advanced data querying, like query GQ in Sect. 6.1, where relevant event data must be selected based on the evolution of database objects of a specific class.

The main flaw of most of these approaches resides in the way they force the representation of complex systems by means of a flat event log. The data perspective is missing, only allowing one to add attributes at the event, trace or log level. More recent works try to improve the situation, analyzing data dependencies [11] in business models with the purpose of improving them, or even observing changes on object states to improve their analysis [6]. However, none of the existing approaches provides a generic and standard way of gathering, classifying, storing and connecting process and data perspectives on information systems, especially when dealing with databases where the concept of structured process can be fuzzy or nonexistent.

8 Conclusion

In this paper, a meta model has been proposed that provides a way to capture a more descriptive image of the reality of business information systems. This meta model aligns the data and process perspectives and enables the application of existing process mining techniques. At the same time, it unleashes a new way to query data and historical information. This is possible thanks to the combination of data and process perspectives on the analysis of information systems. The applicability of the technique has been demonstrated by means of the analysis of several real-life environments. Also, an implementation of the proposed solution has been

[15] https://fluxicon.com/disco/.

[16] https://www.celonis.com.

developed and tested, including adapters for each of these well-known environments. However, from our point of view, the main contribution of this work is the universality of the proposed solution. Its applicability to so many different environments provides a common ground to separate data extraction and analysis as different problems, in this way generating an interface that is much richer and powerful than the current existing standards. To summarize, it provides a standardization layer to simplify and generalize the analysis phase.

Nevertheless, several challenges remain open. For instance, a system that enhances the query building experience, allowing for a more natural and user-friendly way, is desirable. Also, mechanisms to exploit the benefits of the process side (control flow, performance information, conformance, etc.) of the meta model when combined with the data (e.g., data rules discovery) will make the solution really beneficial in comparison with the regular queries that can be performed on the source database systems.

Appendix A: Meta model formalization

This section proposes a formalization of the elements in this meta model, starting from the data and continuing with the process side. As has been mentioned before, we can assume a way to classify elements in types or *classes* exists. Looking at our running example, we can distinguish between a *ticket* class and a *customer* class. This leads to the definition of data model as a way to describe the schema of our data.

Definition 1 (Data Model) A data model is a tuple $DM = (CL, AT, classOfAttribute, RS, sourceClass, targetClass)$ such that

- CL is a set of class names,
- AT is a set of attribute names,
- $classOfAttribute \in AT \rightarrow CL$ is a function that maps each attribute to a class,
- RS is a set of relationship names,
- $sourceClass \in RS \rightarrow CL$ is a function that maps each relationship to its source class,
- $targetClass \in RS \rightarrow CL$ is a function that maps each relationship to its target class

Each of these elements belonging to a *Class* represents a unique entity, something that can be differentiated from the other elements of the same class, e.g., *Customer A* and *Customer B*. We will call them *Objects*, being unique entities according to our meta model.

Definition 2 (Object Collection) Let OBJ be the set of all possible objects. An object collection OC is a set of objects such that $OC \subseteq OBJ$.

Something we know as well is that, during the execution of a process, the nature of these elements can change over time. Modifications can be made on the attributes of these *objects*. Each of these represents *mutations* of an object, modifying the values of some of its attributes, e.g., modifying the address of a customer. As a result, despite being the same object, we will be looking at a different *version* of it. The notion of *object version* is therefore introduced to show the different stages in the life cycle of an *object*.

During the execution of a process, operations will be performed, and many times, links between elements are established. These links allow relating *tickets* to *concerts*, or *customers* to *bookings*, for example. These relationships are of a structured nature and usually exist at the data model level, being defined between *classes*. Therefore, we know upfront that elements of the class *ticket* can be related somehow to elements of the class *concert*. *Relationships* is the name we use to call the definition of these links at the data model level. However, the actual instances of these *relationships* appear at the *object version* level, connecting specific versions of objects during a specific period of time. These specific connections are called *relations*.

Definition 3 (Version Collection) Let V be some universe of values, TS a universe of timestamps and $DM = (CL, AT, classOfAttribute, RS, sourceClass, targetClass)$ a data model. A version collection is a tuple $OVC = (OV, attValue, startTimestamp, endTimestamp, REL)$ such that

- OV is a set of object versions,
- $attValue \in (AT \times OV) \nrightarrow V$ is a function that maps a pair of object version and attribute to a value,
- $startTimestamp \in OV \rightarrow TS$ is a function that maps each object version to a start timestamp,
- $endTimestamp \in OV \rightarrow TS$ is a function that maps each object version to an end timestamp such that $\forall ov \in OV : endTimestamp(ov) \geq startTimestamp(ov)$,
- $REL \subseteq (RS \times OV \times OV)$ is a set of triples relating pairs of object versions through a specific relationship.

At this point, it is time to consider the *process* side of the meta model. The most basic piece of information we can find in a process event log is an event. These are defined by some attributes, among which we find a few typical ones like *timestamp*, *resource* or *life cycle*.

Definition 4 (Event Collection) Assume V to be some universe of values and TS a universe of timestamps. An event collection is a tuple $EC = (EV, EVAT, eventAttributeValue, eventTimestamp, eventLifecycle, eventResource)$ such that

- EV is a set of events,
- $EVAT$ is a set of event attribute names,
- $eventAttributeValue \in (EV \times \text{EVAT}) \nrightarrow V$ is a function that maps a pair of an event and event attribute name to a value,
- $eventTimestamp \in EV \rightarrow TS$ is a function that maps each event to a timestamp,
- $eventLifecycle \in EV \rightarrow \{start, complete, \ldots\}$ is a function that maps each event to a value for its *life-cycle* attribute,
- $eventResource \in EV \rightarrow V$ is a function that maps each event to a value for its resource attribute.

When we consider events of the same activity but relating to a different life cycle, we gather them under the same *activity instance*. For example, two events that belong to the activity *make booking* could have different *life-cycle* values, being *start* the one denoting the beginning of the operation (first event) and *complete* the one denoting the finalization of the operation (second event). Therefore, both events belong to the same *activity instance*. Each of these activity instances can belong to different *cases* or *traces*. At the same time, *cases* can belong to different *logs*, that represent a whole set of *traces* on the behavior of a process.

Definition 5 (Instance Collection) An instance collection is a tuple $IC = (AI, CS, LG, aisOfCase, casesOfLog)$ such that

- AI is a set of activity instances,
- CS is a set of cases,
- LG is a set of logs,
- $aisOfCase \in CS \rightarrow \mathcal{P}(AI)$ is a function that maps each case to a set of activity instances[17],
- $casesOfLog \in LG \rightarrow \mathcal{P}(CS)$ is a function that maps each log to a set of cases.

The last piece of our meta model is the *process model collection*. This part stores *process models* on an abstract level, i.e., as sets of *activities*, ignoring details about the control flow or how these activities relate between them. An *activity* can belong to different *processes* at the same time.

Definition 6 (Process Model Collection) A process model collection is a tuple $PMC = (PM, AC, actOfProc)$ such that

- PM is a set of processes,
- AC is a set of activities,
- $actOfProc \in PM \rightarrow \mathcal{P}(AC)$ is a function that maps each process to a set of activities.

Now we have all the pieces of our meta model, but it is still necessary to wire them together. A *connected meta model* defines the connections between these blocks. Therefore, we see that *versions* belong to *objects* (*objectOfVersion*) and *objects* belong to a class (*classOfObject*). In the same way, *events* belong to *activity instances* (*eventAI*), *activity instances* belong to *activities* (*activityOfAI*) and can belong to different *cases* (*aisOfCase*), *cases* to different *logs* (*casesOfLog*) and *logs* to process (*processOfLog*). Connecting both data and process views, we find *events* and *versions*. They are related (*eventToOVLabel*) in a way that can be interpreted as a causal relation between *events* and *versions*, i.e., when *events* happen they trigger the creation of *versions* as a result of modifications on data (the update of an attribute for instance). Another possibility is that the event represents a read access or query of the values of a *version*.

Definition 7 (Connected Meta Model) A connected meta model is defined as a tuple $CMM = (DM, OC, classOfObject, OVC, objectOfVersion, EC, eventToOVLabel, IC, eventAI, PMC, activityOfAI, processOfLog)$ such that

- $DM = (CL, AT, classOfAttribute, RS, sourceClass, targetClass)$ is a data model,
- OC is an object collection,
- $classOfObject \in OC \rightarrow CL$ is a function that maps each object to a class,
- $OVC = (OV, attValue, startTimestamp, endTimestamp, REL)$ is a version collection,
- $objectOfVersion \in OV \rightarrow OC$ is a function that maps each object version to an object,
- $EC = (EV, EVAT, eventAttributeValue, eventTimestamp, eventLifecycle, eventResource)$ is an event collection,
- $eventToOVLabel \in (EV \times OV) \nrightarrow V$ is a function that maps pairs of an event and an object version to a label. If $(ev, ov) \in dom(eventToOVLabel)$, this means that both event and object version are linked. The label itself defines the nature of such link, e.g., *"insert"*, *"update"*, *"read"*, *"delete"*, etc.,
- $IC = (AI, CS, LG, aisOfCase, casesOfLog)$ is an instance collection,
- $eventAI \in EV \rightarrow AI$ is a function that maps each event to an activity instance,
- $PMC = (PM, AC, actOfProc)$ is a process model collection,
- $activityOfAI \in AI \rightarrow AC$ is a function that maps each activity instance to an activity,
- $processOfLog \in LG \rightarrow PM$ is a function that maps each log to a process.

[17] $\mathcal{P}(X)$ is the powerset of X, i.e., $Y \in \mathcal{P}(X)$ if $Y \subseteq X$.

To define validity, we introduce some shorthands:

Definition 8 (Notations I) Let $CMM = (DM, OC, classOfObject, OVC, objectOfVersion, EC, eventToOVLabel, IC, eventAI, PMC, activityOfAI, processOfLog)$ be a connected meta model. We define the following shorthands:

- $E_{ai} = \{e \in EV \mid eventAI(e) = ai\}$,
- $AI_{cs} = \{ai \in AI \mid ai \in aisOfCase(cs)\}$,
- $OV_{obj} = \{ov \in OV \mid objectOfVersion(ov) = obj\}$,
- $OC_{cl} = \{obj \in OC \mid classOfObject(obj) = cl\}$,
- $REL_{rs} = \{(rs', ov, ov') \in REL \mid rs' = rs\}$,
- $AIAC_{ac} = \{ai \in AI \mid activityOfAI(ai) = ac\}$.

However, in order to consider a meta model as valid, the data and connections in it must fulfill certain criteria set in the following definition:

Definition 9 (Valid Connected Meta Model) Let $CMM = (DM, OC, classOfObject, OVC, objectOfVersion, EC, eventToOVLabel, IC, eventAI, PMC, activityOfAI, processOfLog)$ be a connected meta model. A valid connected meta model VCMM is a connected meta model such that:

- *attValue* is only defined for attributes of the same class of the object version, that is, $(at, ov) \in domain(attValue) \iff classOfObject(objectOfVersion(ov)) = classOfAttribute(at)$,
- none of the object versions of a same object overlap in time, that is, $\forall obj \in OC : \forall ov_a \in OV_{obj} : \forall ov_b \in OV_{obj} : endTimestamp(ov_a) \leq startTimestamp(ov_b) \vee endTimestamp(ov_b) \leq startTimestamp(ov_a)$,
- $\forall log \in LG : \forall cs \in casesOfLog(lg) : \forall ai \in AI_{cs} : activityOfAI(ai) \in actOfProc(processOfLog(lg))$, that is, all the cases of a log contain only activity instances that refer to activities of the same process of the log.

Appendix B: Extended formalizations

The meta model proposed in this paper has been described in Sect. 3 and formalized in "Appendix A". Then, Sect. 5 presents three environments in which data were extracted from their corresponding databases and transformed to adapt to the structure of our meta model. This section presents a formal description of these three environments, and their mapping to the meta model.

B.1 Common definitions to the three environments

The three environments we are dealing with use a relational database to store all the relevant information. Therefore, the three of them share some characteristics. In this subsection, formalizations common to the three environments are presented. We start with the *source data model* in Definition 10.

Definition 10 (Source Data Model) Assume V to be some universe of values. A source data model is a tuple $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$ such that

- C is a set of class names,
- A is a set of attribute names,
- $classAttr \in C \rightarrow \mathcal{P}(A)$ is a function mapping each class onto a set of attribute names. A_c is a shorthand denoting the set of attributes of class $c \in C$, i.e., $A_c = classAttr(c)$,
- $val \in A \rightarrow \mathcal{P}(V)$ is a function mapping each attribute onto a set of values. $V_a = val(a)$ is a shorthand denoting the set of possible values of attribute $a \in A$,
- PK is a set of primary key names,
- FK is a set of foreign key names,
- PK and FK are disjoint sets, that is $PK \cap FK = \emptyset$. To facilitate further definitions, the shorthand K is introduced, which represents the set of all keys: $K = PK \cup FK$,
- $keyClass \in K \rightarrow C$ is a function mapping each key name to a class. K_c is a shorthand denoting the set of keys of class $c \in C$ such that $K_c = \{k \in K \mid keyClass(k) = c\}$,
- $keyRel \in FK \rightarrow PK$ is a function mapping each foreign key onto a primary key,
- $keyAttr \in K \rightarrow \mathcal{P}(A)$ is a function mapping each key onto a set of attributes, such that $\forall k \in K : keyAttr(k) \subseteq A_{keyClass(k)}$,
- $refAttr \in (FK \times A) \nrightarrow A$ is a function mapping each pair of a foreign key and an attribute onto an attribute from the corresponding primary key, i.e., $\forall k \in FK : \forall a, a' \in keyAttr(k) : (refAttr(k, a) \in keyAttr(keyRel(k)) \wedge (refAttr(k, a) = refAttr(k, a') \implies a = a')$.

The following notations in Definition 11 will help us to express mappings and objects in a more natural way.

Definition 11 (Notations) Let $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$ be a source data model.

- $M^{SDM} = \{map \in A \nrightarrow V \mid \forall a \in dom(map) : map(a) \in V_a\}$ is the set of mappings,
- $O^{SDM} = \{(c, map) \in C \times M^{SDM} \mid dom(map) = classAttr(c)\}$ is the set of all possible objects of SDM.

Another common aspect of all the environments is the concept of *source object model*. Definition 12 describes this

concept, which corresponds to a snapshot of the database at a certain moment in time.

Definition 12 (Source Object Model) Let $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$ be a source data model. A source object model of SDM is a set $SOM \subseteq O^{SDM}$ of objects. $\mathcal{U}^{SOM}(SDM) = \mathcal{P}\left(O^{SDM}\right)$ is the set of all object models of SDM.

To ensure the validity of the source object model, i.e., the fulfillment of all the constraints such as primary and foreign keys, we introduce the concept of *valid source object model* in Definition 13.

Definition 13 (Valid Source Object Model) Assume $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$ to be a source data model. $VSOM \subseteq \mathcal{U}^{SOM}(SDM)$ is the set of valid source object models. We say that $SOM \in VSOM$ if the following requirements hold:

- $\forall (c, map) \in SOM : \big(\forall k \in K_c \cap FK : \big(\exists (c', map') \in SOM :$ $keyClass(keyRel(k)) = c' \wedge (\forall a \in keyAttr(k) :$ $map(a) = map'(\text{refAttr}(k, a))\big)\big)\big)$, i.e., referenced objects must exist,
- $\forall (c, map), (c, map') \in SOM : \big(\forall k \in K_c \cap PK :$ $((\forall a \in keyAttr(k) :$ $map(a) = map'(a)\big) \implies map = map'\big)$, i.e., PK and UK values must be unique.

In one or another form, we find events when extracting our data. Something in common between the events we find in the three mentioned environments is that three types can be distinguished: additions, updates and deletions. We call these types *source event types*, as explained in Definition 14.

Definition 14 (Source Event Types) Let $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$ be a source data model and VSOM the set of valid source object models. $SET = ET_{add} \cup ET_{upd} \cup ET_{del}$ is the set of source event types composed of the following pairwise disjoint sets:

- $ET_{add} = \{(\oplus, c) \mid c \in C\}$ are the event types for adding objects,
- $ET_{upd} = \{(\oslash, c) \mid c \in C\}$ are the event types for updating objects,
- $ET_{del} = \{(\ominus, c) \mid c \in C\}$ are the event types for deleting objects.

Each of the three environments we are trying to formalize presents different characteristics when recording execution events. Definition 15 provides a common, base concept of *source events*. However, in further sections we will see what are the particularities of this concept in each of the environments.

Definition 15 (Source Events) Let $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$ be a source data model, VSOM the set of valid source object models, SET the set of source event types, and $map_{null} \in \{\varnothing\} \rightarrow V$ a function with the empty set as domain. SE is the set of source events such that $\forall e \in SE : \exists et \in SET : e = (et, map_{old}, map_{new})$.

Finally, something needed for the mapping between these systems and our meta model is the existence of a concept of *source event occurrence* and *source change log*. Definition 16 provides a description of these concepts. However, we will see how each environment interprets this differently, and how a change log can be inferred from each of them.

Definition 16 (Source Event Occurrence and Source Change Log) Let $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$ be a source data model, VSOM the set of valid source object models and SE the set of source events. Assume some universe of timestamps TS. $eo = (e, ts, resource, lifecycle) \in SE \times TS \times V \times V$ is a source event occurrence. $SEO(SDM, SE) = SE \times TS \times V \times V$ is the set of all possible source event occurrences. A source change log $SCL = \langle eo_1, eo_2, \ldots, eo_n \rangle$ is a sequence of source event occurrences such that time is non-decreasing, i.e., $SCL = \langle eo_1, eo_2, \ldots, eo_n \rangle \in (SEO(SDM, SE))^*$ and $ts_i \leq ts_j$ for any $eo_i = (e_i, ts_i)$ and $eo_j = (e_j, ts_j)$ with $1 \leq i < j \leq n$.

Finally, Definition 17 establishes some useful notations for further definitions, with respect to source event occurrences and source object ids.

Definition 17 (Notations II) Assume a universe of timestamps, a source event occurrence $eo = (((evT, c), map_{old}, map_{new}), ts, resource, lifecycle)$, and a source data model SDM. We define the following shorthand: $objectId(c, map) = \{(a, v) \in (A, V) | a \in keyAttr(PK_c) \wedge map(a) = v\}$, i.e., it returns a set of pairs (*attribute*, *value*) for a mapping *map* according to the attributes of the primary key of such class *c* in the source data model.

Now, all the common elements of the three environments have been formalized. These represent the common ground needed in order to make the mapping to our meta model. The three following sections (Sects. B.2, B.3, and B.4) formalize some of the concepts that differ between the three environments. Finally, Sect. B.5 proposes a mapping to our meta model.

B.2 Database redo logs: formalization

The redo log environment presents some particularities with respect to how the events are represented. Definition 18 for-

malizes this concept while maintaining compatibility with the common description of source events in Definition 15.

Definition 18 (Redo Log Events) Let $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$ be a source data model, VSOM the set of valid source object models and $map_{null} \in \{\varnothing\} \rightarrow V$ a function with the empty set as domain. $SE = E_{add} \cup E_{upd} \cup E_{del}$ is the set of source events composed of the following pairwise disjoint sets:

- $E_{add} = \{((\oplus, c), map_{old}, map_{new})) \mid (c, map_{new}) \in O^{SDM} \wedge map_{old} = map_{null}\}$
- $E_{upd} = \{((\oslash, c), map_{old}, map_{new})) \mid (c, map_{old}) \in O^{SDM} \wedge (c, map_{new}) \in O^{DM}\}$
- $E_{del} = \{((\ominus, c), map_{old}, map_{new})) \mid (c, map_{old}) \in O^{SDM} \wedge map_{new} = map_{null}\}$

This is, mainly, the only difference between the redo log environment and the common description provided in the previous section. However, for the other two environments, some additional details need to be taken into account.

B.3 In-table versioning: formalization

In the case of in-table versioning environments, change logs are not explicitly recorded. However, we find object versions implicitly recorded within the tables of the database. Definition 19 formalizes this structure.

Definition 19 (Implicit Versions Record) Let $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$ be a source data model. An implicit versions record is a tuple $IVR = (TABNAME, OBJECTID, TIMESTAMP, VERID, FNAMES, value)$ such that:

- *TABNAME* is a set of table names,
- $OBJECTID \subseteq \mathbb{N}$ is a set of object identifiers,
- *TIMESTAMP* is a set of timestamps of versions,
- $VERID \subseteq TABNAME \times OBJECTID \times TIMESTAMP$ is a set of unique version identifiers formed by the combination of a table name, an object id and a time stamp,
- $FNAMES \subseteq A$ is the set of attributes of the object version,
- $value \in (VERID \times A) \nrightarrow V$ is a mapping between a pair (*versionId*, *attributeName*) and its new value after the change.

Now that we know how object versions are being defined in this specific environment, we can show how it affects our previous definition of *source object model*. Definition 20 shows the compatibility in this particular case.

Definition 20 (ITV Source Object Model) Given a source data model $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$ and an implicit versions record IVR, an ITV source object model SOM is a set of objects such that $SOM \subseteq O^{SDM}$ such that: $\forall o = (c, map) \in SOM : c \in TABNAME \wedge \forall a \in domain(map) : a \in FNAMES \wedge \exists v = (tab, ob, ts) \in VERID : map(a) = value(v, a) \wedge \nexists v' = (tab, ob, ts') \in VERID : ts' > ts$, i.e., the ITV source object model SOM is formed by all the most recent object versions for each object id.

Definition 21 (Notations III) Let $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$ be a source data model, and $IVR = (TABNAME, OBJECTID, TIMESTAMP, VERID, FNAMES, value)$ an implicit versions record. Given a version id $vid = (tab, ob, ts) \in VERID$, we define the following shorthands:

- itvEvType(id)
- $itvEvType \in VERID \rightarrow \{\oplus, \oslash\}$ is a function mapping object version ids to types of change, such that $itvEvType(vid) = \oplus \iff \nexists vid' = (tab', ob', ts') \in VERID : tab' = tab \wedge ob' = ob \wedge ts' < ts$, i.e., if a previous version of the same object does not exist, it is considered to be an addition change. Otherwise, $itvEvType(vid) = \oslash$, i.e., it is considered to be an update,
- $itvTs \in VERID \rightarrow TIMESTAMP$ is a function mapping object version ids to timestamps such that, $itvTs(vid) = ts$,
- $itvTabName \in VERID \rightarrow TABNAME$ is a function mapping object version ids to table names such that, $itvTabName(vid) = tab$,
- $itvObjId \in VERID \rightarrow OBJECTID$ is a function mapping object version ids to object ids such that, $itvObjId(vid) = ob$,
- $itvPrevVerId \in VERID \rightarrow (VERID \cup \{\varnothing\})$ is a function mapping object version ids to their predecessor object version id (or null id if a predecessor does not exist) such that, $itvPrevVerId(vid) = vid' \wedge (vid' = (tab', ob', ts') \wedge tab = tab' \wedge ob = ob' \wedge ts > ts' \wedge \nexists (tab', ob', ts'') \in VERID : ts > ts'' > ts') \vee (itvEvType(vid) = \oplus \wedge vid' = \varnothing)$,

Finally, the only building block left to define in this case is the correlation between the previous definition of source event occurrences (Definition 16) with this environment. Definition 22 shows that equivalence and provides the key to translate implicit object versions into source event occurrences.

Definition 22 (ITV Source Event Occurrences) Let TS be a universe of timestamps, V a universe of values and *SDM* a source data model $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$. We say that $SEO(SDM, IVR) = SE \times TS \times V \times V$ is a

set of ITV source event occurrences such that $\forall eo = (((evT, c), map_{old}, map_{new}), ts, resource, lifecycle) \in SEO(SDM, IVR) : \exists id \in VERID$:

- $evT = itvEvType(id) \wedge$
- $c = itvTabName(id) \wedge$
- $(\{itvPrevVerId(id)\} \times domain(map_{old})) \subseteq domain(value) \wedge$
- $(\{id\} \times domain(map_{new})) \subseteq domain(value) \wedge$
- $\forall a \in domain(map_{old}) : map_{old}(a) = value(itvPrevVerId(id), a) \wedge$
- $\forall a \in domain(map_{new}) : map_{new}(a) = value(id, a) \wedge$
- $ts = itvTs(id) \wedge$
- $itvObjId(id) = objectId(c, map_{new}) \wedge$
- $resource = \emptyset \wedge$
- $lifecycle = itvEvType(id)$,

That is, for each source event occurrence in $SEO(SDM, IVR)$ exists an implicit version record in *IVR* that shares values for all its properties.

B.4 SAP-style change table: formalization

SAP systems are a different kind of environment. They are closely related to the redo log environments, with the difference that the change record contains the relevant information in a slightly different way. Definition 23 provides details on this *SAP change record*.

Definition 23 (SAP Change Record) Assume a universe of values V, a universe of date values *DATE* and a universe of time values *TIME*. Given a source data model $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$, a SAP change record is a tuple $SCR = (CHNR, OBJECTCLAS, OBJECTID, CHNID, uName, uDate, uTime, change\_ind, tabName, tabKey, fName, value\_new, value\_old)$ such that:

- $CHNR \subseteq \mathbb{N}$ is a set of SAP change numbers,
- *OBJECTCLAS* is a set of SAP object classes,
- $OBJECTID \subseteq \mathbb{N}$ is a set of SAP object ids,
- $CHNID \subseteq CHNR \times OBJECTCLAS \times OBJECTID$ is a set of unique change identifiers formed by the combination of a change number (*CHNR*), an object class (*OBJECTCLAS*) and an object id (*OBJECTID*),
- $uName \in CHNID \rightarrow V$ is a mapping between change ids and user name strings,
- $uDate \in CHNID \rightarrow DATE$ is a mapping between change ids and date values,
- $uTime \in CHNID \rightarrow TIME$ is a mapping between change ids and time values,
- $change\_ind \in CHNID \rightarrow \{\oplus, \oslash, \ominus\}$ is a mapping between change ids and a change type,
- $tabName \in CHNID \rightarrow C$ is a mapping between change ids and a class name,
- $tabKey \in CHNID \rightarrow V$ is a mapping between change ids and the primary key of the modified object,
- $fName \in CHNID \rightarrow \mathcal{P}(A)$ is a mapping between change ids and a set of changed attributes,
- $value\_new \in (CHNID \times A) \nrightarrow V$ is a mapping between a pair (change id,attribute name) and its new value after the change,
- $value\_old \in (CHNID \times A) \nrightarrow V$ is a mapping between a pair (change id,attribute name) and its old value before the change.

The previous definition gives us the ground to build the mapping to source event occurrences. Definition 24 describes how to obtain the source event occurrences previously introduced from the SAP change record. This allows inferring the change log necessary to build our meta model.

Definition 24 (SAP Source Event Occurrences) Assume a universe of timestamps TS, a universe of values V, and a function $convertDateTime \in (DATE \times TIME) \rightarrow TS$ that maps pairs of date and time values into a timestamp. Given a source data model $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$ and a SAP change record $SCR = (CHNR, OBJECTCLAS, OBJECTID, CHNID, uName, uDate, uTime, change\_ind, tabName, tabKey, fName, value\_new, value\_old)$, we define $SEO(SDM, SCR) = SE \times TS \times V \times V$ as a set of source event occurrences such that $\forall eo = (((evT, c), map_{old}, map_{new}), ts, resource, lifecycle) \in SEO(SDM, SCR) : \exists id \in CHNID$:

- $evT = change\_ind(id) \wedge$
- $c = tabName(id) \wedge$
- $map_{old}(fName(id)) = value\_old(fName(id)) \wedge$
- $map_{new}(fName(id)) = value\_new(fName(id)) \wedge$
- $ts = convertDateTime(uDate(id), uTime(id)) \wedge$
- $tabKey(id) = objectId(c, map_{new}) \wedge$
- $resource = uName(id) \wedge$
- $lifecycle = change\_ind(id)$,

That is, for each event occurrence in $SEO(SDM, SCR)$ exists an event record in *SCR* that shares values for all its properties.

B.5 Common meta model mapping for the three environments

In the previous sections of this Appendix, we have defined the common aspects of the three environments under study, together with the particularities of each of them. Now, we have defined the necessary notions to map each concept from the original sources to our meta model. In this section, we will define this mapping for each of the main elements of

the meta model, except for the *cases* and *process models* sectors, which are independent from the source data and can be inferred from the extracted information once it has been already mapped to our meta model. We will start with the mapping of the *source data model* in Definition 25.

Definition 25 (Mapped Data Model) Given a source data model $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$, a mapped data model is a tuple $MDM = (CL, AT, classOfAttribute, RS, sourceClass, targetClass)$ such that:

- $CL = C$ is a set of class names,
- $AT = A$ is a set of attribute names,
- $classOfAttribute \in AT \rightarrow CL$ is a function that maps each attribute to a class, such that $\forall at \in AT : at \in classAttr(classOfAttribute(at))$,
- $RS = FK$ is a set of relationship names,
- $sourceClass \in RS \rightarrow CL$ is a function that maps each relationship to its source class, such that $\forall rs \in RS : sourceClass(rs) = keyClass(rs)$,
- $targetClass \in RS \rightarrow CL$ is a function that maps each relationship to its target class, such that $\forall rs \in RS : targetClass(rs) = keyClass(keyRel(rs))$.

The same can be done with the *source object model*. Its mapping is formalized in Definition 26. Also, in Definition 27 we redefine the concept of *timestamps* to include the situations in which the beginning or end of a period is unknown. Then, some useful notations for further definitions are expressed in Definition 28.

Definition 26 (Mapped Object Collection) Assume $SOM \in VSOM$ to be a valid source object model, MDM a mapped data model, and OBJ the set of all possible objects for MDM. A mapped object collection MOC is a set of objects such that $MOC \subseteq OBJ$, and $mappedObj \in MOC \leftrightarrow SOM$ is a bijective function that maps every mapped object to a source object and vice versa.

Definition 27 (Universe of Global Timestamps) Assume TS to be a universe of timestamps. A universe of global timestamps is a set GTS such that $GTS = TS \cup \{-Œ\} \cup \{+Œ\}$, where $-\phi$ represents an undefined timestamp in the past, and $+\phi$ an undefined timestamp in the future. These timestamps fulfill the following condition: $\forall ts \in TS : -Œ < ts < +Œ$.

Definition 28 (Notations IV) Assume a universe of timestamps TS. Given a source change log SCL, and a mapped data model $MDM = (CL, AT, classOfAttribute, RS, sourceClass, targetClass)$, we define the following shorthands:

- $SCL_a^b(c, oid) = \{eo_i \in SCL | eo_i = ((evT, c), map_{old}, map_{new}, t) \wedge b > i > a \wedge objectId(c, map_{new}) = oid\}$, is the set of event occurrences in the source change log such that all of them occurred after the a-th and before the b-th element of the sequence, and they correspond to objects of class c and with the object id oid,
- $tseo \in \mathcal{P}(SCL) \rightarrow \mathcal{P}(TS)$ is a function that returns a set of timestamps corresponding to the provided set of event occurrences.

One of the key elements of this mapping is the version collection. In Definition 29, we use the source change log and object models to infer the content of the versions part of our meta model.

Definition 29 (Mapped Version Collection) Assume a universe of global timestamps GTS, a mapped object collection MOC, a source data model $SDM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr)$, a mapped data model $MDM = (CL, AT, classOfAttribute, RS, sourceClass, targetClass)$, and a source change log $SCL =< eo_1, eo_2, \ldots, eo_n >$. A mapped object version collection is a tuple $MOVC = (OV, attValue, startTimestamp, endTimestamp, REL)$ such that:

- $OV = \{(c, map, t_s, t_e)\}$ is a set of object versions for which the following holds:
  - $\forall o = (c, map) \in MOC : \exists ov \in OV : ov = (c, map, t_s, t_e) \wedge t_s = max(tseo(SCL_1^n(c, id)), -\phi) \wedge t_e = +\phi$, i.e., for every object in the mapped object collection exists a version for which the start timestamp is either unknown or the timestamp of the last source event occurrence that affected that object, and the end timestamp is unknown,
  - $\forall i \in 1..n : eo_i = ((evT, c), map_{old}, map_{new}, t_{ev}) \in SCL : \exists ov \in OV : ov = (c, map, t_s, t_e) \wedge (((evT = \oplus \vee evT = \oslash) \wedge map_{new} = map \wedge t_s = t_{ev} \wedge t_e = min(ts(SCL_{i+1}^n(c, id)), +\phi)) \vee (evT = \ominus \wedge map_{old} = map \wedge t_e = t_{ev} \wedge t_s = max(ts(SCL_1^{i-1}(c, id)), -\phi)))$, i.e., for every source event occurrence that represents an addition or a modification, there is an object version with the values after the event occurrence, start timestamp equal to the event occurrence timestamp, and end timestamp equal to the one of the next event occurrence that affected the same object. In case it is the first version of the object, the start timestamp will be unknown. If it is the last version of the object, the end timestamp will be unknown instead.
- $attValue \in (AT \times OV) \nrightarrow V$ is a function that maps values to pairs of attributes and object versions such that, given an attribute $at \in AT$ and an object version $ov = (c, map, t_s, t_e)$, $attValue(at, ov) = map(at)$,

- $startTimestamp \in OV \rightarrow GTS$ is a function that returns the start timestamp of an object version such that, given an obj. version $ov = (c, map, t_s, t_e), startTimestamp(ov) = t_s$,
- $endTimestamp \in OV \rightarrow GTS$ is a function that returns the end timestamp of an object version such that, given an object version $ov = (c, map, t_s, t_e), endTimestamp(ov) = t_e$,
- $REL \subseteq (RS \times OV \times OV)$ is a set of triples relating pairs of object versions through specific relationships such that, given a relationship $rs \in RS$, and two object versions $ov_a = (c_a, map_a, t_{sa}, t_{ea}) \in OV$ and $ov_b = (c_b, map_b, t_{sb}, t_{eb}) \in OV$, it is always true that $(rs, ov_a, ov_b) \in REL \iff rs \in FK \wedge sourceClass(rs) = c_a \wedge targetClass(rs) = c_b \wedge (t_{sa} \leq t_{sb} \leq t_{ea} \vee t_{sb} \leq t_{sa} \leq t_{eb}) \wedge \forall at \in keyAtt(rs) : map_a(at) = map_b(refAttr(rs, at))$, i.e., two object versions are related through a relationship if they belong to the source and target classes of the relationship, respectively, and if there is a mapped foreign key from the source data model such that both versions share the same values for the key attributes. In addition, both versions must have coexisted in time.

Finally, in Definition 30 we describe how the events of the meta model are mapped to the source change log.

Definition 30 (Mapped Event Collection) Assume V to be some universe of values, TS a universe of timestamps and SCL a mapped change log $SCL = < eo_1, eo_2, \ldots, eo_n >$. A mapped event collection is a tuple $MEC = (EV, EVAT, eventAttributeValue, eventTimestamp, eventLifecycle, eventResource)$ such that:

- EV is a set of events,
- $EVAT$ is a set of event attribute names,
- $eventAttributeValue \in (EV \times EVAT) \nrightarrow V$ is a function that maps a pair of an event and event attribute name to a value,
- $eventTimestamp \in EV \rightarrow TS$ is a function that maps each event to a timestamp,
- $eventLifecycle \in EV \rightarrow \{start, complete, \ldots\}$ is a function that maps each event to a value for its life-cycle attribute,
- $eventResource \in EV \rightarrow V$ is a function that maps each event to a value for its resource attribute.

And $\forall ev \in EV : \exists eo \in SCL : eo = (((evT, c), map_{old}, map_{new}), ts, resource, lifecycle) \wedge ts = eventTimestamp(ev) \wedge resource = eventResource(ev) \wedge lifecycle = eventLifecycle(ev) \wedge \forall (ev, at) \in domain(eventAttributeValue) : map_{new}(at) = eventAttributeValue(ev, at)$, i.e., for each event in the mapped event collection, there is an event occurrence in the mapped change log that shares timestamp, resource, life cycle and attribute values.

The presented mapping specifies how data are represented in each of the considered environments, and how these data can be transformed in order to populate with content our meta model, with the purpose of performing further analysis in a more standardized and automated way.

References

1. Bolt, A., van der Aalst, W.M.P.: Multidimensional process mining using process cubes. In: Enterprise, Business-Process and Information Systems Modeling—16th International Conference, BPMDS 2015, 20th International Conference, EMMSAD 2015, Held at CAiSE 2015, Stockholm, Sweden, June 8–9, 2015, Proceedings, pp. 102–116 (2015)
2. Buijs, J.: Mapping data sources to xes in a generic way. Master's thesis, Technische Universiteit Eindhoven, The Netherlands (2010)
3. Eder, J., Olivotto, G.E., Gruber, W.: A data warehouse for workflow logs. In: Engineering and Deployment of Cooperative Information Systems, pp. 1–15. Springer (2002)
4. González López de Murillas, E., Reijers, H.A., van der Aalst, W.M.P.: Connecting databases with process mining: a meta model and toolset. In: International Workshop on Business Process Modeling, Development and Support, pp. 231–249. Springer (2016)
5. González-López de Murillas, E., van der Aalst, W.M.P., Reijers, H.A.: Process mining on databases: Unearthing historical data from redo logs. In: Business Process Management. Springer (2015)
6. Herzberg, N., Meyer, A., Weske, M.: Improving business process intelligence by observing object state transitions. Data Knowl. Eng. **98**, 144–164 (2015)
7. IEEE Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams. IEEE Std 1849-2016, pp. 1–50 (2016). https://doi.org/10.1109/IEEESTD.2016.7740858
8. Ingvaldsen, J.E., Gulla, J.A.: Preprocessing support for large scale process mining of SAP transactions. In: Business Process Management Workshops, pp. 30–41. Springer (2008)
9. Lu, X., Nagelkerke, M., van de Wiel, D., Fahland, D.: Discovering interacting artifacts from ERP systems. IEEE Trans. Serv. Comput. **8**(6), 861–873 (2015)
10. Mahendrawathi, E., Astuti, H.M., Wardhani, I.R.K.: Material movement analysis for warehouse business process improvement with process mining: a case study. In: Asia Pacific Business Process Management, pp. 115–127. Springer (2015)
11. Meyer, A., Pufahl, L., Fahland, D., Weske, M.: Modeling and enacting complex data dependencies in business processes. In: Proceedings of the 11th international conference on Business Process Management, pp. 171–186. Springer (2013)
12. Mueller-Wickop, N., Schultz, M.: ERP event log preprocessing: timestamps vs. accounting logic. In: Design Science at the Intersection of Physical and Virtual Design, Lecture Notes in Computer Science, vol. 7939, pp. 105–119. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-38827-9_8
13. Neumuth, T., Mansmann, S., Scholl, M.H., Burgert, O.: Data warehousing technology for surgical workflow analysis. In: 21st IEEE International Symposium on Computer-Based Medical Systems, 2008, CBMS'08, pp. 230–235. IEEE (2008)
14. Niedrite, L., Solodovnikova, D., Treimanis, M., Niedritis, A.: Goal-driven design of a data warehouse-based business process analysis system. In: Proceedings of the 6th Conference on 6th WSEAS

Interanational Conference on Artificial Intelligence, Knowledge Engineering and Data Bases, pp. 243–249 (2007)
15. Popova, V., Fahland, D., Dumas, M.: Artifact lifecycle discovery. Int. J. Cooper. Inf. Syst. **24**(01), 1550,001 (2015)
16. Rosemann, M., Zur Muehlen, M.: Evaluation of workflow management systems-a meta model approach. Aust. J. Inf. Syst. **6**(1), 103–116 (1998)
17. Sismanis, Y., Brown, P., Haas, P.J., Reinwald, B.: Gordian: efficient and scalable discovery of composite keys. In: Proceedings of the 32nd International Conference on Very Large Data Bases, pp. 691–702. VLDB Endowment (2006)
18. Štolfa, J., Kopka, M., Štolfa, S., Koběrskỳ, O., Snášel, V.: An application of process mining to invoice verification process in sap. In: Innovations in Bio-Inspired Computing and Applications, pp. 61–74. Springer (2014)
19. van Dongen, B.F., van der Aalst, W.M.P.: A meta model for process mining data. EMOI-INTEROP **160**, 30 (2005)
20. van der Aalst, W.M.P.: Process cubes: Slicing, dicing, rolling up and drilling down event data for process mining. In: Asia Pacific Business Process Management—First Asia Pacific Conference, AP-BPM 2013, Beijing, China, 29–30 August , 2013. Selected papers, pp. 1–22 (2013)
21. van der Aalst, W.M.P.: Extracting event data from databases to unleash process mining. BPM–Driving Innovation in a Digital World, Management for Professionals, pp. 105–128. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-14430-6_8
22. Vogelgesang, T., Appelrath, H.: A relational data warehouse for multidimensional process mining. In: Proceedings of the 5th International Symposium on Data-Driven Process Discovery and Analysis (SIMPDA 2015), Vienna, Austria, 9–11 December, 2015, pp. 64–78 (2015)
23. Vogelgesang, T., Appelrath, H.: Pmcube: a data-warehouse-based approach for multidimensional process mining. In: Business Process Management Workshops—BPM 2015, 13th International Workshops, Innsbruck, Austria, August 31–September 3, 2015, Revised Papers, pp. 167–178 (2015)
24. Vogelgesang, T., Kaes, G., Rinderle-Ma, S., Appelrath, H.: Multidimensional process mining: questions, requirements, and limitations. In: Proceedings of the CAiSE'16 Forum, at the 28th International Conference on Advanced Information Systems Engineering (CAiSE 2016), Ljubljana, Slovenia, June 13–17, 2016., pp. 169–176 (2016)
25. Zhang, M., Hadjieleftheriou, M., Ooi, B.C., Procopiuc, C.M., Srivastava, D.: On multi-column foreign key discovery. Proceedings of the VLDB Endowment **3**(1–2), 805–814 (2010)
26. Zur Muehlen, M.: Workflow-based process controlling-or: what you can measure you can control. Workflow handbook, pp. 61–77 (2001)
27. Zur Muhlen, M.: Evaluation of workflow management systems using meta models. In: Proceedings of the 32nd Annual Hawaii International Conference on Systems Sciences, HICSS-32 (1999)
28. Zur Muehlen, M., Rosemann, M.: Workflow-based process monitoring and controlling-technical and organizational issues. In: Proceedings of the 33rd Annual Hawaii International Conference on System Sciences, 2000, pp. 10–pp. IEEE (2000)
29. Zur Muehlen, M.: Process-driven management information systems combining data warehouses and workflow technology. In: Proceedings of the International Conference on Electronic Commerce Research (ICECR-4), pp. 550–566 (2001)

Eduardo González López de Murillas has been a Ph.D. student at the Eindhoven University of Technology, the Netherlands, since 2014. His research interests include process mining, data extraction and transformation, data querying, automated event log building, and business process management.

Hajo A. Reijers is a full professor of Business Informatics at Vrije Universiteit Amsterdam, the Netherlands. He also holds a position as part-time, full professor at Eindhoven University of Technology. Previously, he worked as a management consultant and R&D manager. His research interests relate to business process management, data analytics and conceptual modeling. On these and other topics, he published over 200 scientific papers, book chapters and professional publications.

Prof.dr.ir. Wil M. P. van der Aalst is a full professor at RWTH Aachen University. He is also a visiting researcher at Fondazione Bruno Kessler (FBK) in Trento and a member of the Board of Governors of Tilburg University. Until 2018 he was also the scientific director of the Data Science Center Eindhoven (DSC/e). His personal research interests include process mining, Petri nets, business process management, workflow management, process modeling and process analysis. Wil van der Aalst has published over 200 journal papers, 20 books (as author or editor), 450 refereed conference/workshop publications and 65 book chapters. Next to serving on the editorial boards of over 10 scientific journals he is also playing an advisory role for several companies, including Fluxicon, Celonis and ProcessGold. Van der Aalst received honorary degrees from the Moscow Higher School of Economics (Prof. h.c.), Tsinghua University, and Hasselt University (Dr. h.c.). He is also an elected member of the Royal Netherlands Academy of Arts and Sciences, the Royal Holland Society of Sciences and Humanities, and the Academy of Europe.