Generation of Timelines from Event Knowledge Graphs Using Domain Knowledge

Rikayan Chaki(✉), Diego Calvanese, and Marco Montali

Free University of Bozen-Bolzano, 39100 Bolzano, Italy
{rchaki,diego.calvanese,marco.montali}@unibz.it

Abstract. Analysts reconstruct the data state of processes from an event log that records the process execution trails. Efficient analysis of these logs requires the storage of event data throughout the lifetime of a process. Traditional trace-based logs store the traces generated by (isolated) process instances. These call for reconstructing case variables after each event in the trace. Object-centric process mining pushes this complexity one level up, as each event can create, delete, or update multiple artifacts like objects, relations, and attributes. Besides, object-centric event data alone is not good enough to reconstruct such updates. In this paper, we tackle this pressing problem by augmenting object-centric event data with a lightweight form of domain knowledge, consisting of condition-effect rules. These rules explicitly capture the semantics of events contained in the logs. We then propose a method that takes an object-centric event log and a set of condition-effect rules as input, and produces a timeline accounting for the step-by-step evolution of the artifacts. Cypher queries can then be used to inspect the timeline and retrieve which facts hold at a given time point, or the time intervals of existence for the artifacts.

Keywords: Object-centric process mining · Domain knowledge · Condition-effect rules · Temporal Property Graph

1 Introduction

Process mining is concerned with the problem of extracting useful insights and knowledge from the data about (business) processes, which are typically generated by information systems that record information about the processes running inside an organization [1,3]. Traditional process mining techniques rely on the implicit or explicit presence of a *case notion*. At the model level, this notion is used to single out process instances, with the strong assumption that different instances do not interact with each other. At the event data level, this notion is essential to group recorded events into execution traces. Extensive literature indicates that such a case notion is often too restrictive, since business and work processes not only focus on the co-evolution of multiple objects simultaneously,

K. Saeed et al. (Eds.): CISIM 2025, LNCS 15927, pp. 275–289, 2026.
https://doi.org/10.1007/978-3-032-02406-0_20

but also depend on one-to-many and many-to-many relationships connecting them [6]. In such cases, *flattening* process models and event data through the lens of a single case notion can lead to severely underspecified models and misleading process mining results [2,6].

An alternative approach is that of *object-centric process mining*, in which the intrinsic relational nature of events and objects is preserved. This calls for lifting execution traces to relational representations, such as event tables in OCEL 1.0 [16] and 2.0 [20], or event knowledge graphs (EKGs) [13]. However, these approaches cannot conclusively infer which objects/relationships/attributes hold at a given time, since the connections between events and entities recorded in such tables/graphs do not explicitly indicate *how* such objects participate in the different events, nor which relationships are *created/removed by specific events*.

To obtain a *global timeline* from the log, it is necessary to make use of *domain knowledge* regarding the initial data state and, even more importantly, the effects induced by the different (types of) events in the log. In [24], an EKG is enriched with a *semantic layer*, which contains lightweight domain knowledge, formulated using schema-level qualifiers on the event's operations. Such qualifiers, paired with assumptions about the EKG itself, indirectly enable the reconstruction of a global timeline. Techniques from action learning in planning [21] could also be used in this regard, but to the best of our knowledge there has been no research so far in this direction. Domain knowledge has been used to derive semantics from event logs in various process and data handling tasks, such as for workflow management [26], as a language for the Semantic Web [23], and even in user interface design [17]. Such formalisms are also customary in all modelling approaches dealing with processes and relational data [8].

In this paper, we tackle this issue from a different angle: we enrich an object-centric log with domain knowledge by indicating *condition-effect rules that explicitly capture the semantics of events present in the log*. Specifically, taking inspiration from an extensive body of work at the intersection of process and data management [8], we define, for each rule, a *relational condition* that, if true, triggers an *effect* adding or deleting a fact in the state (where a fact may relate to an object, an attribute, or a relationship).

Specifically, we provide the following contributions, described in Sect. 3.

1. We justify and describe two variants of timelines (namely, global and local) that are generated with our approach, as discussed in Sect. 3.1.
2. We provide methods that derive the step-by-step evolution of objects, relationships, and attributes using rules that consider the current state of these artifacts, as well as the object-centric event log, as discussed in Sect. 3.2.
3. We define a formalism to specify event condition-effect rules for global and local timeline reconstruction, as discussed in Sect. 3.3.
4. We define a set of queries that can inspect the timeline and retrieve which facts hold at a given time point, or the time intervals of existence for objects, attributes, and relationships, as discussed in Sect. 3.4.

2 Preliminaries

We now provide the technical background for our work, and we illustrate the notions we introduce on a running example.

2.1 Event Knowledge Graphs

An *Event Knowledge Graph* (EKG), storing the log of an event in the form of a graph, is used to represent complex relationships between *events* and *multiple entities* of different *types* [12]. A key advantage of this formalism over a traditional event log with a fixed case notion is that it does not require an entity type to be marked as a *case* type before the log is created. This prevents problems such as *convergence* and *divergence* following data flattening, which often occur when a fixed case notion is used [2].

An EKG is a form of *Labeled Property Graph* (LPG), defined over a set *Label* of labels and a set V of values, and represented by a tuple $G = (N, R, label, prop)$, where N is a set of nodes, $R \subseteq N \times N$ a set of relationships, $label : (N \cup R) \rightarrow 2^{Label}$ the labeling function assigning to each graph element (i.e., node or relationship) a set of labels in *Label*, and $prop : (N \cup R) \times K \rightarrow V$ maps, for each graph element, attribute keys in a set K to a value in V. In an EKG, nodes are partitioned into event and entity nodes, respectively labeled with *Event* and *Entity*, and corresponding to events and entities in the log. Events are assumed to be atomic. Given an event node n, we assume that the time at which it occurs is stored in a $\mathsf{timestamp}$ attribute, thus, $prop(n, \mathsf{timestamp}) = t$ where $t \in V_{time}$, and $V_{time} \subseteq V$ is a totally-ordered set of time values.

An event node is correlated to one or more entity nodes through the *CORR* relationship. This relationship allows us to define the *directly-follows* relationship *DF*, relating two event nodes e_1 and e_2 for an entity a if and only if e_1 is directly followed by e_2 from the *perspective* of events observed by a. This implies that both e_1 and e_2 are correlated to a, e_2 occurs at a later time than e_1, and there is no other event correlated to a whose timestamp is between the ones of e_1 and e_2.

We assume that relationships between entity nodes are *not* stored in the EKG itself. This is because such relationships are dynamic — their existence changes over time. We also assume that initially, before, any event occurs, there does not exist any relationship between entities.

Notice how inherently complex the semantics of events can be, and that it is in general impossible to infer such knowledge by merely looking at the EKG.

2.2 Temporal Property Graphs and Condition-Effect Rules

A *Temporal Property Graph* (TPG) allows one to represent information that changes over time in terms of an LPG, and such LPG can be stored in a graph database [9]. The main benefit of doing so is that information stored in such databases may be queried efficiently using graph query languages, such as GQL [19] and Cypher [15], which are equipped with navigational features. Also, graph databases have been shown to exhibit very good performance especially

in those settings where queries involve both spatial and temporal constraints, e.g., in the domain of Geographic Information System (GIS) data analysis [5]. In addition, given the flexibility of graph-based models, data in graph databases is easier to visualize and thus to use [22].

Syntactically, both TPGs and EKGs are forms of LPGs. However, semantically, each node and relationship in a TPG has a time interval that indicates when that element exists in the system [4]. This makes TPGs an ideal model to store the timeline so that it can be readily queried by the user relying on the expressive power and navigational features of graph query languages. We observe that this model is semantically equivalent to Duration-labeled Temporal Graphs defined in [9], where temporal information is associated to edges only, as intervals defined on vertices in TPGs can be represented in the duration-labeled model using self-loop edges on the vertices with the corresponding time intervals.

As mentioned, we use condition-effect rules to encode domain semantics, and in the condition part of such rules we must be able to refer to and query the data at hand, which in our case is available in graph form. Therefore we have chosen to base our query language on Cypher [15], which, in addition to being a declarative graph query language used by the widely adopted graph database system Neo4J, is being developed towards compliance[1] with the new GQL standard [10,14]. Specifically, a rule can use the relevant entity (in case of rules for entities), entity pairs (in case of rules for relationships), or entity-attribute pairs (in case of rules for attributes), which are provided as parameters in the rule condition.

2.3 Running Example

To demonstrate our proposal, we use a running example involving a system that models a simple company with several departments. An employee of a specific department can work on a project. Additionally, each project is managed by an employee responsible for its overall success.

A system monitoring the actions performed in such a company can have entities of one of the *entity types* *Employee*, *Department*, or *Project*. Between these entity types, we have the following relationship types:

1. *EMPLOYMENT*, relating *Department*s to the *Employee*s therein;
2. *WORKS_IN* relating *Employee*s to the *Project*s they work in;
3. *MANAGES* relating *Employee*s to the *Project*s they manage.

We assume that this kind of schema-level information on the relevant relationship and entity types of the system is provided by domain experts.

In Fig. 1, we show an EKG representing an event log generated by the company system. We represent the type of a node by its shape, where event nodes are square-shaped, while entity nodes are oval-shaped. The type of an entity node is indicated by its color and by the letter in its label: *Employee* nodes a_i in blue, *Department* nodes d_i in black, and *Project* nodes p_i in green. Dashed lines indicate *CORR* relationships between events and entities to which they are

[1] See https://neo4j.com/docs/cypher-manual/current/appendix/gql-conformance/.

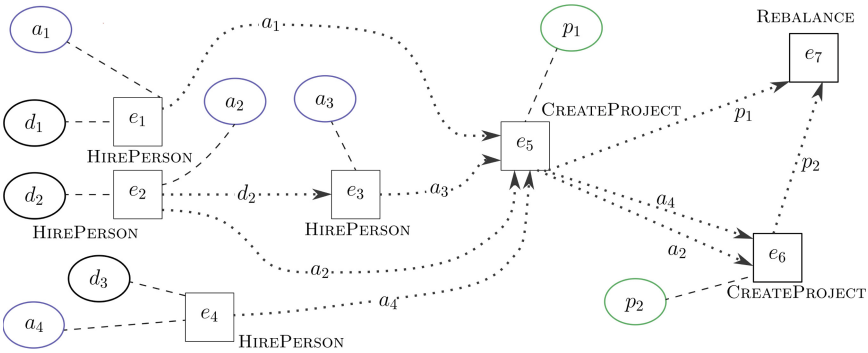

Fig. 1. Sample EKG of Company with Event IDs as timestamps

correlated, while instances of the *DF* relationship type, shown as arrows with a dotted line, have a qualifierattribute (indicated as arrow label) whose value is the identifier of the entity used to derive them. This allows us to de-clutter the diagram by omitting a *CORR* relationship from an event e to an entity a if e has an incident *DF* relationship labeled with a.

The three activities observed in the events of Fig. 1 are HirePerson, CreateProject, and Rebalance. When a project is created, the longest-employed employee becomes the supervisor. In case of a tie, the employee with the smallest employee ID in the tie is picked as supervisor. Activity Rebalance works as follows: Given a set P of projects that are correlated to it:

1. Identify the largest project $p_1 \in P$ and second-largest project $p_2 \in P$, based on the numbers n_1 and n_2 of employees respectively working in p_1 and p_2.
2. If $n_1 - n_2 \leq 1$, no changes are made (notice that $n_1 \geq n_2$).
3. Otherwise, pick the $\lfloor \frac{n_1 - n_2}{2} \rfloor$ employees with the smallest employee IDs in p_1 that are not in p_2, and move them to p_2.
4. Adjust supervisors based on the new employees working in p_1 and p_2.

3 Methodology

In this section, we present our approach for extracting timelines. We first focus on the overall structure of two algorithms employed to reconstruct respectively the global timeline of the entire process and the local timeline for one or more entities controlled by a human actor. We then discuss the condition-effect rules used to express the update semantics of events. Finally, we describe the timelines themselves in more detail.

We start by stating some assumptions for our timeline generation process:

A1: An EKG, a formal system description, a specification of static entity types, and a set of condition-effect rules are provided as input to our algorithm. Entities having static types are not affected by events in the log, and are assumed to be created at the same time as the first event in it.

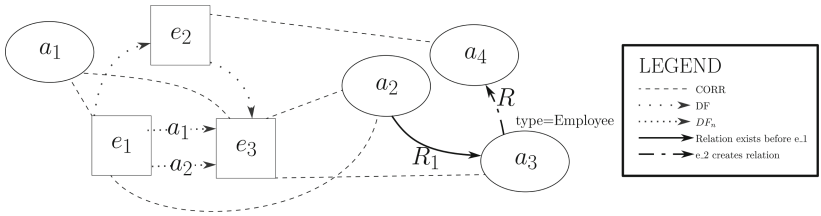

Fig. 2. Scenario of Example 1, on the difference between global and local timelines

A2: Effects of events are instantaneous, i.e., they happen without a delay.
A3: Events that occur at the same time have mutually-independent effects.
A4: The users of a system may not have access to all of its components. For example, in the context of Business Process Mining [11], where key stakeholders in a process lifecycle are its participants, a *process participant* is a *human* actor, responsible for the functioning of a part of a process. Such an actor can only access the events associated with the entities under their responsibility.
A5: The applicability of a condition-effect rule to an event is given by a condition that includes the name of the activity observed by it. Further, we also assume that a rule takes as input the current entity/relationship/attribute associated to the event whose creation/deletion is being considered.

3.1 Global and Local Timelines

We present two algorithms that produce different timelines from a given input, assuming different log views, given A1. Further, considering A2 and A3, the reconstruction of how objects (entities and relations) and their properties evolve can be done by incrementally considering events in ascending order of timestamps, and events that happen at the same time can be processed in an arbitrary order.

The first algorithm we propose considers events in the log chronologically and finds a single, so-called *global timeline* that includes information, instant-by-instant, about which entities, attributes, and relationships hold. Thus, we assume that the effects obtained by applying the condition-effect rules on this "sequence of all events" would yield timelines consistent with the true state of the system at all intermediate moments, starting from the first event. Obtaining such a timeline requires complete knowledge of all the events of a system. The second algorithm uses the notion of process participants and the entities they can access, as discussed in A4. It finds timelines relying on information that is locally available to an actor, and we call such timelines *local*. Notice that if A4 does not hold, the local timeline cannot be reconstructed by merely looking at directly connected events and may also be affected by indirect dependencies with other objects and events.

We illustrate the difference between the two approaches using an example.

Example 1. Consider the events e_1, e_2, and e_3 in Fig. 2, and assume that the relationship $R_1(a_2, a_3)$ already exists before any of the events occur. We note that, differently from the company EKG in Fig. 1, we have specified here the *CORR* relationships from an event e to an entity x even when e has a *DF* relationship labeled by x that is incident on it. Suppose that a set of condition-effect rules are formulated such that event e_1 always creates the entity a_1, and e_2 always creates the relationship $R(a_3, a_4)$. Additionally, the rule for event e_3 is such that it destroys any entity correlated to it if there exists a path from some entity correlated to e_3 through an R_1-relationship to some entity m, and then from m through an R-relationship to some other entity.

When generating the global timeline, e_1 causes a_1 to be created, and then e_2 creates relationship $R(a_3, a_4)$. Thus, the rule for event e_3 deletes all of a_1, a_2, and a_3. Conversely, when generating the local timeline for an actor with only access to a_1, the algorithm *skips over* e_2, which is not correlated to a_1. Therefore $R(a_3, a_4)$ is not created, so the deletion condition fails to hold when e_3 gets processed and specifically a_1 is not deleted. Thus, in this timeline, none of the entities are deleted. ◁

3.2 General Structure of the Algorithms

For the local algorithm involving entities in the set $\mathcal{E}$, we can find a timeline starting from the events E_{start} that do not have a *DF*-relation qualified by any $a \in \mathcal{E}$ ending on them. Then, following *DF* relationships qualified by such entities a using breadth-first search (BFS) starting from nodes in E_{start}, we find the relevant event ordering for processing. The global algorithm, conversely, simply orders *all* events by their timestamps, and then follows this chronological order during processing.

If an entity a having a non-static type is found to be correlated to an event e associated with it, but has not been created by a prior event, then a relation can be created with that entity.

1. When creating a global timeline, we assume that a exists. This is because when creating such a timeline, no event in the log is skipped. Thus, the correlation of a to e provides a strong positive basis for assuming that the event that created it was either not correctly recorded into the log or happened before the log started recording. The time at which a is created is assumed to be as old as the earliest event in the log.
2. When creating a local timeline for entity set $\mathcal{E}$, we assume that a is created only if $a \in \mathcal{E}$. This is because if this is not so, entity a could be created at some unknown time before e by an event that the actor responsible for entities $\mathcal{E}$ does not have acces to. Thus, to maintain consistency, we assume that neither a, nor the relation involving a, is created. If $a \in \mathcal{E}$, we assume that a is created at the same time as the earliest event(s) in E_{start}.

When generating the timeline, we store the interim data states using a simple structure that creates/deletes entities/relations as they are created/deleted by the effects of rules, separately from the final timeline. When considering the effect of an individual event e, both algorithms proceed as follows:

1. We get the list of rules for the activity observed by e.
2. For each rule involving entities, we iterate over the entities correlated to e. If a rule holds, its effects are applied to the current system state.
3. Rules involving relationship creation may either only be applied between the entities correlated to e, or between the entities correlated to e and entities that are *not* correlated to it. Pairs of entities that may be affected by the rule are filtered so that their types match the expected relationship type of the rule. Subsequently, these filtered entity pairs are applied as parameters to evaluate if the effect holds in each case. When the rule creates a relationship, the relationship automatically obtains an identifier whose value orders the relationships in the same way as they were created.
4. For rules involving relationship deletion, the identifiers of existing relationships are used as parameters.
5. When a creation or deletion effect applies to an entity, we add a corresponding temporal attribute, respectively t_{create} or t_{delete}, to the copy of the node corresponding to that entity.
6. Similarly, when a creation effect applies to a relationship of type R between two entities, we create a relationship of type R between their copies, and add to this relationship the attribute t_{create} with the current timestamp. For relationship deletion, we add the attribute t_{delete} to the relationship whose identifier matches the given value.

3.3 Condition-Effect Rules

We use condition-effect rules to determine when an entity, relationship, or entity attribute is created or deleted by an event observing some activity, and a given *data state* of the system. This state is stored as a graph that must conform to the relationship and entity types prescribed in the input. These rules are used to store domain knowledge about the semantics of relevant activities implicitly.

We define a set $\mathcal{A}$ of relationship types used to model attributes of entities. Relationships in $\mathcal{A}$ are directed from an entity to an *attribute-type* node, which is assumed to always exist as an endpoint for the attribute relationship. Given an entity a of type A and an attribute *attr*, such a node has the unique identifier $a\_\_attr$, which is used when constructing a relationship to model the creation of the attribute in question. The relationship connecting a to $a\_\_attr$ has type $A\_\_attr$. For example, if an activity creates an attribute called *name* for entities *john* and *mary* of type *Employee*, we create relationships of type *Employee*$\_\_$*name* from *john* to *john*$\_\_$*name* and from *mary* to *mary*$\_\_$*name*. Rules for these relationships can thus be specified using the same condition-effect rule formalism as rules involving relationships between entities.

Rules have the form

$$\textsc{Act}^{\ell}, \varphi(\vec{x}) \;\rightsquigarrow\; E(\vec{x}),$$

where:

1. Act is the name of the activity that must be observed by any event e for which the rule is applied.

2. $\ell \in \{\texttt{E}\} \cup \mathcal{R} \cup \mathcal{A}$, where $\mathcal{R}$ is the set of relationship types used for rules involving relationships, $\mathcal{A}$ is the set of attribute types, as discussed above, and `E` is used to indicate rules involving entities.
3. $\vec{x} \in N \cup N \times N \cup N \times N_A \cup N_R \cup N_A$, where N, N_R, and N_A are respectively the sets of entity, relationships, and attribute identifiers.
4. $\varphi(\vec{x})$ is a Cypher query with parameters $\vec{x}$, of the following specific form:

   `MATCH (...) WHERE` *condition* `RETURN COUNT(*) > 0`

   These queries always yield boolean values as their result.
   - The *condition* is a Cypher `WHERE` clause specified by the user. The `MATCH` clause selects the entities/relationships that are used in *condition*, and it is subsequently added as a prefix of the query. The `RETURN` clause, added as a suffix of the query, ensures that the query returns *true* if and only if the entities/relationships selected by the `MATCH` clause satisfy *condition*. This is necessary since Cypher does not allow for a formula with only a condition in its syntax.
   - $\varphi(\vec{x})$ may contain special terms $\mathsf{entity}(x)$ and $\mathsf{relation}(x)$, where x in $\vec{x}$, which are substituted with the entities/relationships obtained from the added `MATCH` clause.
   - $\varphi(\vec{x})$ may also contain the special term *CORR*, which is substituted by the list of entities that are correlated to event e.
5. $E(\vec{x})$ represents the *effect* of the rule, and it can be one of CEnt, DEnt, $\mathsf{CIntRel}$, $\mathsf{CExtRel}$, $\mathsf{DIntRel}$, $\mathsf{DExtRel}$, CAtt, DAtt, representing respectively the creation/deletion of entities/internal relationships/external relationships/attributes. Internal relationships are between entities in *CORR*, while external relationships are between entities in *CORR* and outside *CORR*.
   Notice that, when a rule is applied to an entity $a \in N$, the effect is the creation or deletion of a; when it is applied to a pair $(a_1, a_2) \in N \times N$, the effect is the creation of a relationship from a_1 to a_2; when it is applied to a pair $(a_1, a_2) \in N \times N_A$, the effect is the creation of a node $a-$ and of an attribute relationship from a_1 to a_2; when it is applied to a relationship $r \in N_R$, the effect is the deletion of r. and when it is applied to an attribute $attr \in N_A$, the effect is the deletion of $attr$.

Example 2 (Example in Sect. 2.3 cont.'d). To create a *MANAGES* relationship, relating an employee with identifier z to a project with identifier v, we can use the following condition-effect rule for the CreateProject activity:

$$\textsc{CreateProject}^{MANAGES}, \varphi(z, v) \ \rightsquigarrow \ \mathsf{CIntRel}(z, v)$$

where the *condition* part in $\varphi(z, v)$ is:

```
NOT EXISTS (
    MATCH (y:Employee) ← [e1:EMPLOYMENT] − (),
          (entity(z)) ← [e2:EMPLOYMENT] − ()
    WHERE y ∈ CORR
          AND (e1.start_emp < e2.start_emp
               OR (e1.start_emp = e2.start_emp
                   AND y.employee_nr < entity(z).employee_nr)))
```

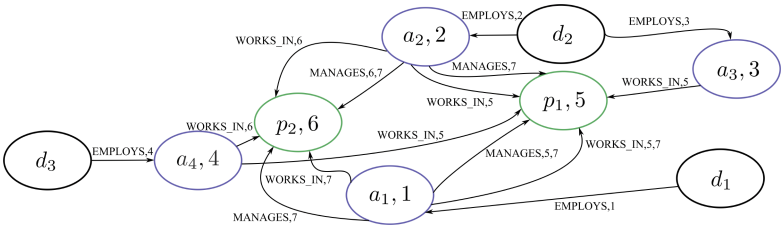

Fig. 3. Global timeline of all entities

The `MATCH` clause in this rule searches for two *paths* that both begin with an *Employee*-type entity node, followed by an incoming relationship of type *EMPLOYMENT*. The first of these *Employee* nodes, matched by identifier y, must be a node in $CORR_{\text{CREATEPROJECT}}$, while the second is $\mathsf{entity}(z)$, the entity associated with identifier z, i.e., the employee being considered for the manager role. This can be inferred from the *labels*, and the directions of the arrows between the nodes and the edges in the path specification. In Cypher, a label is specified using the *variable*:*label* notation. Thus, e.g., the *EMPLOYMENT* relationships for the employees y and $\mathsf{entity}(z)$ are respectively assigned to e_1 and e_2.

This initial `MATCH` and `WHERE` condition are followed by a disjunction of the following clauses:

1. $e_1.start\_emp < e_2.start\_emp$ checks if $e_1.start\_emp$ is an earlier date than $e_2.start\_emp$. Thus, this condition holds if and only if y joined the company earlier than $\mathsf{entity}(z)$.
2. The second condition is a conjunction that first restricts the focus to cases where y joined the company on the same date as $\mathsf{entity}(z)$, via $e_1.start\_emp = e_2.start\_emp$, and then checks if the employee number of y is smaller than that of $\mathsf{entity}(z)$.

Since the whole clause is inside a `NOT EXISTS` (...) clause, this second clause holds for $\mathsf{entity}(z)$ if and only if *(i)* no employee joined the company earlier than $\mathsf{entity}(z)$, and *(ii)* for all employees that joined the company at the same time as $\mathsf{entity}(z)$, their employee numbers are larger than or equal to that of $\mathsf{entity}(z)$.

These conditions, together with the condition that $\mathsf{entity}(z)$ is employed are therefore used to determine if $\mathsf{entity}(z)$ is to manage a newly-created project. ◁

3.4 Timeline Extraction and Querying

We illustrate how the global and a local timelines can be extracted from an EKG by applying rules like the ones mentioned above for the running example introduced in Sect. 2.3, with the entity type *Department* being specified as a static entity.

Example 3 (Example 2 cont.'d — Global Timeline Extraction). Applying the global timeline algorithm to the EKG in Fig. 1 yields the timeline represented by the TPG in Fig. 3. The node colors and identifiers that indicate the

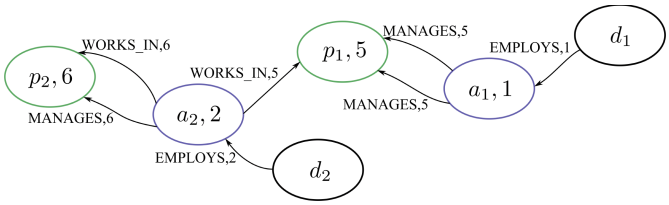

Fig. 4. Local Timeline for entities a_1, a_2

entity type are the same as those in Fig. 1. For visualization purposes, nodes are labeled with the node identifier, followed by its t_{create} value, and if set, its t_{delete} value. Relationships are labeled by their type, followed by similar time values. We assume that the timestamp of event e_i is i.

Thus, for example, we can observe that employee a_2 was created at time 2. This makes sense, since from Fig. 1, we note that e_2, the event that occurs at time 2, is a HIREPERSON event involving a_2.

We describe how the *MANAGES* relationship from employee a_1 to project p_1 evolves. Event e_5 (at time 5) created project p_1. Since it contained employee a_1, who is strictly the oldest employee in this sample, it is assigned as the manager of p_1. Then, in the REBALANCE activity of event e_7, since p_1 had a size of 4, and p_2 of 2, one employee is moved from the former to the latter. This chosen employee, that is, a_1, has the lowest employee number in p_1 that was not already in p_2. Since a_1 no longer belonged to p_1, the *MANAGES* relationship between them was destroyed at time 7. We can observe this in Fig. 3 considering the label *MANAGES*, 5, 7 between the nodes for p_1 and a_1. ◁

Example 4 (Example 2 cont.'d — Local Timeline Extraction). Figure 4 is the local timeline extracted from the EKG of Fig. 1 when considering an actor that has access to entities $\mathcal{E} = \{a_1, a_2\}$, following the algorithm described in Sect. 3.2. In comparison to the global timeline, we note that despite all four employees being correlated to the CREATEPROJECT event e_5, only a_1 and a_2 have *WORKS_IN* relations created for them at time 5. This is because entities a_3 and a_4 are of a non-static type *Employee*, they were not created at any time before e_5 in the context of this local timeline, and they are also not a part of the set $\mathcal{E}$. The *WORKS_IN* relationship from a_4 to p_2 is also not created for a similar reason. Finally, there is no effect of REBALANCE in this timeline, since at time 7, p_1 contains only one employee more than p_2. ◁

Once a timeline is generated and stored as a TPG, it can be queried using a graph query language such as Cypher. Below, we list some example queries:

(i) Is an entity with ID n present at time t?

```
MATCH (z)
WHERE z.id = n
      AND (z.t_create <= t AND (z.t_delete IS NULL OR t < z.t_delete))
```

(ii) Is a relationship present with type R from an entity with ID n_1 to one with ID n_2 at time t?

$$\begin{aligned}&\texttt{MATCH } (z_1) - [r{:}R] \rightarrow (z_2)\\ &\texttt{WHERE } z_1.id = n_1 \texttt{ AND } z_2.id = n_2 \texttt{ AND } r.t_{create} <= t\\ &\qquad \texttt{AND } (r.t_{delete} \texttt{ IS NULL OR } t < r.t_{delete})\end{aligned}$$

(iii) Is an attribute *attr* present for an entity type with ID n of type A at time t?

$$\begin{aligned}&\texttt{MATCH } (z{:}A) - [r] \rightarrow (a)\\ &\texttt{WHERE } z.id = n \texttt{ AND } a.id = n\_\_attr\\ &\qquad \texttt{AND } r.t_{create} <= t \texttt{ AND } (r.t_{delete} \texttt{ IS NULL OR } t < r.t_{delete})\end{aligned}$$

Alternative formulations of these queries can also be constructed to find the identifiers of data elements present at a certain timestamp in the absence of the non-temporal conditions. An example of the former kind of query is that of finding all relations that are present at time t.

Example 5 (Example 3cont.'d). Consider the global timeline generated in Fig. 3 in the company scenario. One can perform the following query:

At times 6 and 7, what are the *WORKS_IN* relationships connected to project p_2?

This query yields the result that at time 6, there are *WORKS_IN* relationships from a_2 and a_4 to p_2, while at time 7, the *WORKS_IN* relationships incident on p_2 are from a_1 and a_4. ◁

4 Experimental Evaluation

We have implemented[2] our algorithm. In this section, we describe how we obtained timelines from the BPI-2017 event log. All tests were performed on a platform with the following software and hardware specifications:

1. Software: Python 3.12 interpreter running in PyCharm version 2024.1.2 on Windows 11 Pro.
2. Hardware: Intel i7-1165G7 Processor, 16 GB DDR4-SDRAM, 512 GB SSD.

We have used Neo4J [18] to store both the Event Knowledge Graph and the generated Temporal Property Graph in the same database. To reproduce the results obtained, please follow the steps below

1. Install Neo4J on your system, noting down the username and password.
2. Run the `initialize.py` file to configure the implementation with the authentication details for your Neo4J installation.
3. Run `main.py` and follow the instructions. Enter the *absolute* path to the BPI17 dataset when this is requested.

[2] See https://gitlab.inf.unibz.it/Rikayan.Chaki/timeline-generation.

Table 1. Temporal statistics for queries (in milliseconds)

	Entity	Relationship	Attribute	All Entities	All Relationship	All Attribute
mean	14.55	14.13	13.28	1550.54	614.48	300.89
std	10.00	5.87	6.13	95.60	95.55	49.67
99%	49.47	49.24	49.49	1818.07	806.20	403.81

4.1 Input

Below, we list some basic features of BPI-2017, and some basic assumptions we use when attempting to model its underlying domain:

1. Number of entities and events: 74504 and 1202267 respectively.
2. Entity types: *Application*, *Offer*. We consider the *resourceId* attribute of the *Offer* type in our rules as well. We also assume that there is a relationship type *Against* from entities of type *Offer* to those of type *Application*.
3. Events have a *lifecycle* attribute, in addition to the standard *activity* one, that helps to determine the tasks being performed during them. Thus, we assume that activities for events of this log can be defined by a combination of the *activity* and *lifecycle* attributes, of the form *activity_lifecycle*.

4.2 Timeline

We perform 1000 trials for each of the query types discussed in Sect. 3.4 on the global timeline generated from the BPI17 log. For queries requiring entity IDs and relationship types, we chose them randomly from those present in the event log.

Table 1 contains the mean, standard deviation, and 99th percentile times for executing these queries. The first three columns are queries that request to see if a particular entity, a relationship (specified by a pair of entities and relationship type), or an attribute, given by its parent entity and attribute key, is present at a specific timestamp. The remaining three columns query to find *all* entities/relationships/attributes that are present at a given timestamp. We observe that the 99th percentile times are below 2 seconds on a graph containing over 100000 entities for all the queries. This shows that for these queries, our approach works efficiently for graphs of this size. We also note that the use of range indexes on entity IDs causes such to be about 100 times faster than the queries that do not have such restrictions.

5 Conclusions

We have developed a framework to derive timelines from object-centric event logs that pinpoint, moment by moment, which objects, attributes, and relationships are present. To facilitate this, we enhanced the object-centric event data with

domain knowledge, using condition-action rules that reflect the update semantics of events. We are considering different directions along which to extend our work: *(i)* Enriching the formalism used to represent domain knowledge by adopting domain ontologies. These allow one to specify condition-action rules at a higher level of abstraction, and could can rely on query-reformulation techniques [7,25] to compile those high-level rule specifications into concrete executable rules. *(ii)* Considering not only atomic events but also non-atomic events that may overlap in time and have timestamps that are time intervals. This requires a treatment of possible interactions between different events.

References

1. van der Aalst, W.M.P.: Process Mining – Data Science in Action. Springer, 2nd edn. (2016). https://doi.org/10.1007/978-3-662-49851-4
2. van der Aalst, W.M.P.: Object-centric process mining: dealing with divergence and convergence in event data. In: Proc. of the 17th Int. Conf. on Software Engineering and Formal Methods (SEFM). LNCS, vol. 11724, pp. 3–25. Springer (2019). https://doi.org/10.1007/978-3-030-30446-1_1
3. van der Aalst, W.M.P., et al.: Process Mining Manifesto. In: Proc. of the Business Process Management Workshops (BPM-WS). LNBIP, vol. 99, pp. 169–194. Springer (2011). https://doi.org/10.1007/978-3-642-28108-2_19
4. Arenas, M., Bahamondes, P., Aghasadeghi, A., Stoyanovich, J.: Temporal regular path queries. In: Proc. of the 38th IEEE Int. Conf. on Data Engineering (ICDE), pp. 2412–2425. IEEE Computer Society (2022). https://doi.org/10.1109/ICDE53745.2022.00226
5. Baker Effendi, S., van der Merwe, B., Balke, W.T.: Suitability of graph database technology for the analysis of spatio-temporal data. Future Internet **12**(5), 78 (2020)
6. Berti, A., Montali, M., van der Aalst, W.M.P.: Advancements and challenges in object-centric process mining: a systematic literature review. CoRR Technical Report arXiv:2311.08795, arXiv.org e-Print archive (2023)
7. Calvanese, D., et al.: Ontologies and databases: the DL-Lite approach. In: Reasoning Web: Semantic Technologies for Informations Systems – 5th Int. Summer School Tutorial Lectures (RW), LNCS, vol. 5689, pp. 255–356. Springer (2009). 10.1007/978-3-642-03754-2_7
8. Calvanese, D., De Giacomo, G., Montali, M.: Foundations of data-aware process analysis: a database theory perspective. In: Proc. of the 32nd ACM Symp. on Principles of Database Systems (PODS), pp. 1–12. ACM (2013). https://doi.org/10.1145/2463664.2467796
9. Debrouvier, A., Parodi, E., Perazzo, M., Soliani, V., Vaisman, A.: A model and query language for temporal graph databases. VLDB J. **30**(5), 825–858 (2021). https://doi.org/10.1007/s00778-021-00675-4
10. Deutsch, A., et al.: Graph pattern matching in GQL and SQL/PGQ. In: Proc. of the 43rd ACM Int. Conf. on Management of Data (SIGMOD), pp. 2246–2258. ACM (2022). https://doi.org/10.1145/3514221.3526057
11. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer, 2nd edn. (2018). https://doi.org/10.1007/978-3-662-56509-4

12. Esser, S., Fahland, D.: Multi-dimensional event data in graph databases. J. Data Semant. **10**(1–2), 109–141 (2021). https://doi.org/10.1007/S13740-021-00122-1
13. Fahland, D.: Process mining over multiple behavioral dimensions with Event Knowledge Graphs. In: Process Mining Handbook, LNBIP, vol. 448, pp. 274–319. Springer (2022). https://doi.org/10.1007/978-3-031-08848-3_9
14. Francis, N., et al.: A researcher's digest of GQL (Invited talk). In: Proc. of the 26th Int. Conf. on Database Theory (ICDT). Leibniz Int. Proc. in Informatics (LIPIcs), vol. 255, pp. 1:1–1:22. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.ICDT.2023.1
15. Francis, N., et al.: Cypher: an evolving query language for property graphs. In: Proc. of the 39th ACM Int. Conf. on Management of Data (SIGMOD), pp. 1433–1445. ACM (2018)
16. Ghahfarokhi, A.F., Park, G., Berti, A., van der Aalst, W.: OCEL standard. Techreport 1, Process and Data Science Group, RWTH Aachen University (2020)
17. Ghiani, G., Manca, M., Paternò, F., Santoro, C.: Personalization of context-dependent applications through trigger-action rules. ACM Trans. Comput. Human Interac. **24**(2), 14:1–14:33 (2017). https://doi.org/10.1145/3057861
18. Guia, J., Soares, V.G., Bernardino, J.: Graph databases: Neo4j analysis. In: Proc. of the 19th Int. Conf. on Enterprise Information Systems (ICEIS), pp. 351–356. SciTePress (2017). https://doi.org/10.5220/0006356003510356
19. Information technology – Database languages – GQL. Standard, International Organization for Standardization, Geneva, CH (2024)
20. Koren, I., Adams, J.N., Berti, A., van der Aalst, W.M.P.: OCEL 2.0 resources – www.ocel-standard.org. In: Doctoral Consortium and Demo Track at the Int. Conf. on Process Mining (ICPM). CEUR Workshop Proceedings, vol. 3648. CEUR-WS.org (2023)
21. Lamanna, L., Serafini, L.: Action model learning from noisy traces: a probabilistic approach. In: Proc. of the 34th Int. Conf. on Automated Planning and Scheduling (ICAPS), pp. 342–350. AAAI Press (2024). https://doi.org/10.1609/ICAPS.V34I1.31493
22. Orlando, D., Ormachea, J., Soliani, V., Vaisman, A.A.: TGV: a visualization tool for temporal property graph databases. Inf. Syst. Front. **26**(4), 1543–1564 (2024)
23. Poulovassilis, A., Papamarkos, G., Wood, P.T.: Event-condition-action rule languages for the semantic web. In: Revised Selected Papers of the EDBT 2006 Workshops. LNCS, vol. 4254, pp. 855–864. Springer (2006). https://doi.org/10.1007/11896548_64
24. Swevels, A., Fahland, D., Montali, M.: Implementing object-centric event data models in Event Knowledge Graphs. In: Revised Selected Papers of the Int. Conf. on Process Mining Workshops (ICPM). LNBIP, vol. 503, pp. 431–443. Springer (2023). https://doi.org/10.1007/978-3-031-56107-8_33
25. Xiao, G., et al.: Ontology-based data access: A survey. In: Proc. of the 27th Int. Joint Conf. on Artificial Intelligence (IJCAI), pp. 5511–5519. IJCAI Org. (2018). https://doi.org/10.24963/ijcai.2018/777
26. Ye, Y., Jiang, Z., Diao, X., Du, G.: Extended event-condition-action rules and fuzzy Petri nets based exception handling for workflow management. Expert Syst. Appl. **38**(9), 10847–10861 (2011). https://doi.org/10.1016/J.ESWA.2011.02.097