Ontology-Driven Rules for Efficient OCEL Timeline Synthesis

Rikayan Chaki[0000−0002−7963−6647] and
Diego Calvanese[0000−0001−5174−9693]

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

Abstract. Unlike traditional process logs, which are flat, Object-Centric Event Logs store information about objects of different types, and their association to events. Timelines are artifacts that record how these objects, along with their relations and attributes, evolve over time. In the literature, rule-based approaches to construct timelines from event logs have been proposed. In this paper, we extend such approaches by allowing the user to reuse a restricted portion of knowledge gained during the evaluation of other rules. We also utilize the rules' structure to determine events whose effects are independent. Such a partitioning of events in the input event log enable a parallelization of the timeline generation algorithm. We also present OWL 2 models for both the rule schema and the timeline model, a scheme to visualize a summary of the generated timeline, and a tool that can *(i)* validate a given rule Knowledge Base (KB) against the presented rule schema, *(ii)* construct a timeline from a given OCEL v2.0-compliant input and a set of rules, and *(iii)* visualize the timeline using the aforementioned timeline summary visualizer model.

Keywords: Rule-based Knowledge Graph Creation, ASP, Timeline Generation, Object-Centric Event Logs

1 Introduction

Event logs have traditionally been flat representations of the actions occurring in a system from the view of a specific object type (identified as the "case" type). This *case-centric* view limits the log to events in which objects of the chosen type participate and hides details about interactions between objects of different types in the system. Further, these issues increase in prominence as the complexity of the data model of a system increases.

To capture interactions between control flow and data more faithfully, object-centric process logs have been proposed, where events can be related to multiple objects. These event to object relations are termed *correlations*. Various schemas support this modelling of an event log as a graph, such as OCEL v1.0 [14], OCEL v2.0 [3], and Event Knowledge Graphs (EKGs) [15]. However, they support only a limited vocabulary for representing dynamic information about system data. Recently, tEKG, an extension to the EKG model, has been proposed [16]. This can record snapshots of objects and qualified relations between objects at different times. However, they use this model to encode an OCEL v2.0 [3] log, relying on the implicit assumption that the rich spatio-temporal information in processes is present in the source log itself, and does not need to be extracted.

In practice, temporal knowledge is often implicitly reconstructed by domain experts from event logs, especially in legacy systems where such information may not be readily apparent. In [9], we addressed the extraction of spatio-temporal knowledge from event logs using domain knowledge. The framework takes as input an object-centric event log, condition-effect rules, user-specified objects, and a data schema. It constructs a view of the log containing only events associated with the selected objects, and applies the rules to record creations and deletions of objects, their attributes, and object-to-object relations. The tool implemented in [9] can export the resulting timelines to Neo4j [19] for querying. However, this timeline generation framework did not support attribute value updates. We extended it in [8] by introducing a mechanism for recording attribute value updates, together with a formal execution semantics and a corresponding timeline generation algorithm. Timeline construction proceeds by sequentially processing the events while maintaining the current data state to evaluate rules and incrementally building the timeline. The object filtering of [9] is regarded as a preprocessing step and is therefore omitted in both [8] and the present work.

While the approach of [8] is theoretically sound, it lacks a practical implementation integrated with existing knowledge-base technologies, despite the availability of tools such as OnProm [5–7]. Furthermore, the task of specifying large rule sets does not scale well without a structured schema supporting validation and interoperability.

Although the execution semantics of [8] adds support for recording attribute updates, including for objects beyond those directly linked to an event, specifying such rules efficiently remains challenging in practice. Rule conditions frequently share common sub-formulae, leading to repeated evaluation of the same conditions across rules. For example, in an order-management scenario, suppose that a SUBMITORDER event involves only the order and its items. Due to inflation, the prices of the products corresponding to the items must be updated by traversing the relations between items and products. If prices may increase at most once per year, the timestamp of the last price increase must also be recorded to enable this condition verification. Consequently, the timestamp update occurs exactly when the price update does. Rather than evaluating the same condition twice, a rule reuse mechanism allows the date update rule to reuse the price update rule's result. Existing approaches do not support such rule reuse, which would both simplify rule specification and avoid duplicate evaluations during timeline generation.

To this end, we present a hybrid rule-evaluation architecture in which graph query (fragments), realized practically via Cypher [12], derive facts over the current data state, which are subsequently processed by a non-recursive Answer Set Program (ASP) to compute rule dependencies and targets. The resulting objects and object pairs determine the state updates applied to the data model. From an implementation perspective, this allows us to leverage ASP solvers such as Clingo [13]. By restricting the logic program to a non-recursive fragment, the program can be stratified, avoiding non-deterministic branching during the solving process [10].

To operationalize our approach, we provide UML Class Diagrams to model respectively the input rule schema and the generated output timeline. This unified representation provides several core benefits: *(i)* We can exploit the correspondence between UML Class Diagrams and ontology languages [2], specifically OWL 2, to obtain a representation both of rule schemas and of output timelines in terms of (RDF) knowledge graphs. *(ii)* Such

knowledge graphs can be serialized into standard formats (e.g., Turtle, JSON-LD), facilitating the management of input rules and output timelines via standard Semantic Web technologies. *(iii)* The standardized vocabulary allows automated integrity validation of user-supplied rule KBs via tools like SHACL. *(iv)* By relying on OWL 2 to define the structural aspects of the rule and timeline schemas, we reduce custom syntactic complexity, allowing us to focus on novel concepts and easing future extensions.

In this work, we present a parallelized algorithm that realizes the proposed timeline generation framework, and a tool that implements this algorithm, supports various visualization/benchmarking capabilities, and is also able to import/export the input rules and the output timeline as knowledge bases.

2 Preliminaries

We define mutually disjoint, and countably infinite sets used throughout the paper: activities $\mathbb{A}$, timestamps $\mathbb{T}$, truth values $\top$ and $\bot$, non-negative integers $\mathbb{N}$, object types $\mathbb{E}$ (with static types $\mathbb{E}_S \subseteq \mathbb{E}$), relation types $\mathbb{R}$, events $\mathbb{U}_E$, objects $\mathbb{U}_O$, attribute types $\mathbb{U}_A$, and values $\mathbb{V} \supseteq \mathbb{T}$. We also define a mapping $\mathsf{R}_X : \mathbb{R} \to \mathbb{E} \times \mathbb{E}$ that associates to each relation type the types of objects participating in a relation of that type. When convenient, we refer to functions using relational notation, i.e., we view a function as a set of tuples.

Event Log Schema. While OCEL v2.0 [3] supports various advanced features, we restrict our event log definition to the elements used for timeline construction, in particular typed object-object relations. We thus define an event log as a tuple $\langle \mathcal{E}, \mathcal{O}, \mathsf{act}, \mathsf{ts}, \mathsf{type}, \mathsf{aVal}, \mathit{CORR}, \mathsf{O2O} \rangle$ where $\mathcal{E} \subseteq \mathbb{U}_E$ is a set of events, $\mathcal{O} \subseteq \mathbb{U}_O$ is a set of objects, $\mathsf{act} : \mathcal{E} \to \mathbb{A}$ and $\mathsf{ts} : \mathcal{E} \to \mathbb{T}$ map events to their activities and timestamps, respectively, $\mathsf{type} : \mathcal{O} \to \mathbb{E}$ maps objects to their types, $\mathsf{aVal} : \mathcal{O} \times \mathbb{K} \nrightarrow \mathbb{V}$ is a finite partial function associating object-attribute type pairs with their values, $\mathit{CORR} \subseteq \mathcal{E} \times \mathcal{O}$ associates events with participating objects, and $\mathsf{O2O} \subseteq \mathcal{O} \times \mathbb{R} \times \mathcal{O}$ denotes a finite set of relations between objects typed by values in $\mathbb{R}$. We define $\mathcal{A}$ as the range of act.

Answer Set Programming (ASP) [4]. A *logic program* is a collection of logical *clauses* used to perform inference over a given domain. Let $\mathcal{LC}$, $\mathcal{LX}$, $\mathcal{LE}$, and $\mathcal{LP}$ respectively be infinite mutually disjoint sets of constants, variables, extensional, and intensional predicate names. We define an atom as $p(t_1, \ldots, t_n)$, $n \geq 0$, where $p \in \mathcal{LP} \cup \mathcal{LE}$ and each term $t_i \in \mathcal{LC} \cup \mathcal{LX}$. A *normal clause* ρ of a program has the form

$$\alpha \leftarrow \beta_1, \ldots, \beta_n, \neg\beta_{n+1}, \ldots, \neg\beta_m$$

where α is an intensional atom called the *head* of ρ, $m \geq 0$, and $\beta_1, \ldots, \beta_m$ are (intensional or extensional) atoms that form the *body* of ρ. A collection of such clauses is called a *normal logic program*. Observe that we have excluded function symbols from this definition, as we not require them in our framework. Let $\mathrm{Vars}(\rho)$ be the variables in ρ, and $\mathrm{Vars}^+_b(\rho)$ the variables in the positive atoms in its body. A clause ρ is *safe* if all of its variables appear at least once in a positive atom, that is, $\mathrm{Vars}(\rho) \subseteq \mathrm{Vars}^+_b(\rho)$. In the following, we assume that all clauses in our logic programs are *safe* and *normal*.

Additionally, we assume that the programs are *non-recursive*, i.e., the dependency graph of clauses is acyclic. This allows us to evaluate the program using the standard stratified bottom-up evaluation approach based on a topological sort of the clause dependencies [1]. Extensional predicate groundings are restricted by the safety of the clauses. Thus, we can compute extensions via standard join operations over finite interpretations rather than the full domain. In the following, we denote by $\text{Pred}_{ext}(P)$ the set of all extensional predicate symbols appearing in the bodies of the rules in an ASP P. *Answer Set Programs* (ASPs) are logic programs interpreted under the stable model semantics. In our setting, an interpretations I is a finite set of ground atoms, where atoms in I are true and all others are false. I is said to be a model of P (denoted $I \models P$) if and only if every clause in the grounding of P is satisfied by I. Given a model I, its Gelfond-Lifschitz reduct P^I is formed by grounding P, removing clauses with a negated atom $\neg\beta$ where $\beta \in I$, and dropping all negated atoms from the remaining clauses. Since P is safe and function-free, grounding is finite. The reduct P^I is strictly positive (that is, its clauses contain no negated atoms) with a unique least model $\text{LM}(P^I)$. An interpretation I is an answer set of P iff $I = \text{LM}(P^I)$. Following standard logic programming semantics, an extensional predicate is identified with its extension, namely the set of its ground facts.

Data States. Our timeline construction must capture the evolution of objects, inter-object relations, and object attributes. Following the data model from [8], we utilize a set $\mathbb{AP}$ of atomic predicates over the sorts $\mathbb{U}_O$ (object identifiers) and $\mathbb{V}$ (attribute values). Specifically, $\mathbb{AP}$ represents an object $x \in \mathbb{U}_O$ as $\texttt{Obj}(x)$, its type $X \in \mathbb{E}$ as $\texttt{type}(x, X)$, a relation of type $R \in \mathbb{R}$ from x to y as $R(x, y)$, and an attribute of type $K \in \mathbb{K}$ mapping x to $v \in \mathbb{V}$ as $K(x, v)$. We define $\mathcal{S} \subseteq \mathbb{AP}$ to be a *data state* if: *(i)* for all $x, y \in \mathbb{U}_O$ and $R \in \mathbb{R}$, if $R(x, y) \in \mathcal{S}$ then $\{\texttt{Obj}(x), \texttt{Obj}(y)\} \subseteq \mathcal{S}$, and if $\mathsf{R}_X(R) = (X, Y)$ then $\{\texttt{type}(x, X), \texttt{type}(y, Y)\} \subseteq \mathcal{S}$; and *(ii)* for all $x \in \mathbb{U}_O$ and $K \in \mathbb{K}$, there is at most one $v \in \mathbb{V}$ such that $K(x, v) \in \mathcal{S}$.

Two-Sided Single-Source Regular Path Queries (2SSRPQ). Since we aim at natively evaluating path queries without translating them into intermediate logic formats, we define a *Two-Sided Single-Source Regular Path Query* (2SSRPQ) over a data state $\mathcal{S}$ as a pair $\langle a, \mathcal{P}\rangle$, where $a \in \mathbb{U}_O$ is an origin object and $\mathcal{P}$ is a regular language over the alphabet $\mathbb{R} \cup \{R^- \mid R \in \mathbb{R}\}$. An object $b \in \mathbb{U}_O$ satisfies $\langle a, \mathcal{P}\rangle$ over $\mathcal{S}$ if there exists a sequence $c_0, Q_1, c_1, \ldots, c_{k-1}, Q_k, c_k$ with $a = c_0$ and $b = c_k$ such that the relation sequence $Q_1 \cdots Q_k \in \mathcal{P}$, and for $0 < i \leq k$, either: *(i)* $Q_i \in \mathbb{R}$ and $Q_i(c_{i-1}, c_i) \in \mathcal{S}$, or *(ii)* Q_i is of the form R^- for some $R \in \mathbb{R}$, and $R(c_i, c_{i-1}) \in \mathcal{S}$. We denote the set of such path languages $\mathcal{P}$ by $\mathbb{RP}$.

3 The Rule Model

We introduce now our rule model and its execution semantics.

3.1 Data Representation

Event logs map straightforwardly into graph structures. We realize both event logs and abstract data states as concrete property graphs in Neo4j.

APs in a data state $\mathcal{S}$ map to elements in its graph representation $G_{\mathcal{S}}$ as follows: $\texttt{Obj}(x)$ maps to a node, $\texttt{type}(x, X)$ to node labels, $R(x, y)$ to typed directed edges, and $K(x, v)$ to node properties. Because $\mathcal{S}$ only permits at most one relation of a given type between two objects, $G_{\mathcal{S}}$ is a simple graph. Enforcing the *Unique Name Assumption* (UNA), each object node is uniquely identified by an invariant `id` property across all representations. Let $\mathcal{ID}$ be the set of all identifiers in $\mathbb{U}$.

3.2 Querying the Graph State

In this section, we present how user-provided logic is automatically translated or *compiled* into Cypher queries evaluated over the graph encoding $G_{\mathcal{S}}$ of a data state $\mathcal{S}$, given the correlation set $V \subseteq \mathcal{ID}$ of the processed event (see Section 3.5).

Query fragments are used both in preconditions that determine if activity occurrence has any effect on the data state and in rules. We group rules by the type of data they affect into three classes: object, relation, and (object) attribute rules. Queries of preconditions, as well as object and attribute rules, are evaluated against a single target object (n_x), while those of relation rules evaluate against a target object pair (n_x, n_y). For object creation rules, n_x denotes an identifier instead of referring to the object being created.

User-specified 2SSRPQs are used to determine, starting from objects in V, the target objects for attribute rules. They are automatically translated into Cypher queries that accept the set V by a reserved parameter `$corr`, exploiting Cypher's native support for transitive closure in path expressions. In the generated query, we apply `DISTINCT` to the `RETURN` statement so that the returned target object identifiers form a set.

Preconditions and rules additionally employ query fragments that are automatically compiled into queries evaluated over $G_{\mathcal{S}}$, given V, which is passed using the reserved parameter `$corr`. Rules additionally utilize the timestamp of the processed event, passed via another reserved parameter `$time`. For attribute rules, an additional parameter `$target` stores the objects reached by the rule's path query, leaving `$corr` available for reasoning over the original event correlation set. Given a valid Cypher query Q, its evaluation over $G_{\mathcal{S}}$ is denoted by $[\![Q]\!]_{G_{\mathcal{S}},\theta}$, where θ is a parameter binding environment. We denote by $\theta[p \mapsto v]$ the environment θ updated with parameter p bound to value v.

User-supplied logic is embedded into system-generated Cypher query templates determined by the surrounding activity or rule. These automatically bind the target variables to the appropriate objects (or object pairs) using the supplied parameters, while enforcing the required structural constraints. Object creation rules instead use a fresh variable bound to a dummy node, since the object being created is not yet available. Within this template architecture, users specify rule logic via two Cypher fragments:

- $\mathbb{GQ}^C$ denotes the collection of *condition fragments* that realize activity preconditions and rule conditions. A fragment in $\mathbb{GQ}^C$ is a boolean predicate evaluated relative to target object bindings.
- $\mathbb{GQ}^A$ denotes the collection of *attribute-value query fragments* used by attribute rules to collect context for external functions. These extend $\mathbb{GQ}^C$ by permitting `MATCH` clauses to navigate to reachable objects and return property values.

Rule conditions are evaluated simultaneously over the entire correlation set V using the `$corr` or `$target` parameter, replacing object-by-object validation with set-based

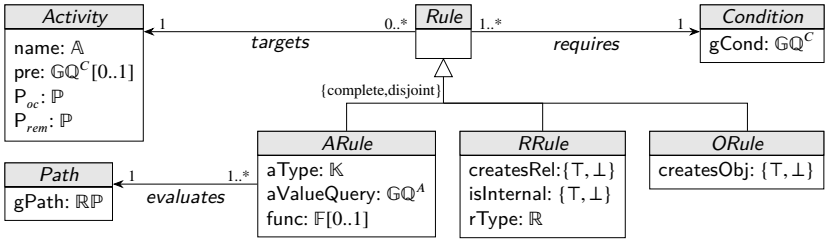

Fig. 1. UML rule model $\mathbb{O}_R$

filtration. This allows Neo4j to exploit query-planner optimizations and compute the full extension of the target objects in a single query.

A $\mathbb{GQ}^C$ fragment functions as a `WHERE`-clause boolean predicate, containing, e.g.:

- label expression predicates over node identifiers to enforce object typing;
- pattern existence predicates (`EXISTS {...}`), possibly including variable-length path quantifiers (i.e., `*` or `+`);
- additional operations of Cypher, such as standard logical connectives (`AND`, `OR`, `NOT`), scalar comparisons (`=`, `!=`, `IS NULL`, `IS NOT NULL`) over node properties, aggregation, and string and list operations.

Let $\mathcal{Q}(F)$ denote the complete Cypher query compiled from a user-supplied fragment $F \in \mathbb{GQ}^C \cup \mathbb{GQ}^A$ by injecting it into the system template. For a condition fragment $C \in \mathbb{GQ}^C$, if C is used as a precondition, the compilation $\mathcal{Q}(C)$ builds a query verifying that C holds for *at least one* supplied target object by appending a `RETURN` clause that evaluates `COUNT(*) > 0`. For the remaining uses of these clauses, crucially, Cypher's native multiset (bag) semantics must be reconciled with the set semantics of data states. While encapsulating conditions in `EXISTS` subqueries inherently forces boolean evaluation, multiset overlaps can still occur. If the C above is used instead in a rule condition, we ensure that set semantics is preserved by injecting `WITH DISTINCT` in $\mathcal{Q}(C)$ for relation rules. The compilation $\mathcal{Q}(C)$ also automatically projects the necessary bindings by appending `RETURN` n_x`.id` (and n_y`.id`, for relation rules). The evaluation $[\![\mathcal{Q}(C)]\!]_{G_S,\theta}$ yields a boolean value when C is a precondition, and a set of filtered objects (or object pairs) otherwise. For an attribute query fragment $A \in \mathbb{GQ}^A$, the compilation prepends the target object identifier n_x`.id` to the `RETURN` clause and adds the `DISTINCT` qualifier. The evaluation $[\![\mathcal{Q}(A)]\!]_{G_S,\theta}$ returns a set of tuples, where the first element is the target object identifier and the remainder consists of the retrieved attribute values.

Finally, we observe that by relying on Cypher as a query language, we allow for expressive power that goes beyond FO.

3.3 Rule Model

We introduce now our rule model, which we present in the form of the UML Class Diagram $\mathbb{O}_R$, shown in Figure 1, together with some additional conditions, specified below. An *Activity* instance binds to rules (*ARule*, *RRule*, or *ORule*) and optionally defines a precondition (pre) and two logic programs ($\mathsf{P}_{oc}, \mathsf{P}_{rem} \in \mathbb{P}$, defined below).

These logic programs enable rule reuse by allowing complex conditions to be evaluated once and shared across multiple rules within the same activity. Rules dictate state changes conditionally based on an associated condition, expressed as a $\mathbb{GQ}^C$-formula (gCond). For *ARule* instances, an path query (gPath) identifies the target objects, while aValueQuery and an optional external function compute new attribute values. We call a set of rules compliant to the rule model in Figure 1 a *rule-base*. For such a rule-base, we require some conditions to be specified, which we detail in the following.

Intensional and Extensional Predicates. To support ASP inference over the data state, we introduce for each rule r (which is either an *ORule*, an *RRule*, or an *ARule*), the following predicates (with arguments ranging over $\mathbb{U}_O$):

1. Intensional target predicates: $\mathsf{eTarUnary}_r(x)$ for object and attribute rules, and $\mathsf{eTarBinary}_r(x, y)$ for relation rules.
2. Extensional satisfaction predicates: $\mathsf{cSatUnary}_r(x)$ for object and attribute rules, and $\mathsf{cSatBinary}_r(x, y)$ for relation rules.
3. Extensional attribute predicates: $\mathsf{pSatUnary}_r(x)$ and $\mathsf{candChanges}_r(x)$ for attribute rules.

Let $\mathbb{P}$ be the collection of all ASPs over these predicates and the set $\mathbb{U}_O$ of constants.

Structural Conditions for a Rule-base. We require a rule-base to satisfy the following structural conditions: *(i)* for each activity $a \in \mathbb{A}$ the following holds: for object creation rules r targeting a, $\mathsf{P}_{oc}(a)$ may only deduce $\mathsf{eTarUnary}_r$, while for a's remaining rules, $\mathsf{P}_{rem}(a)$ is restricted to target predicates; *(ii)* if an attribute rule's aValueQuery returns $k \geq 0$ attribute values (excluding the object identifier), its external function f must have the signature $\mathbb{V}^k \times \mathbb{T} \to \mathbb{V}$, and f may only be absent if $k = 1$. We denote these structural conditions by $\mathbb{I}_R$, and call a rule-base satisfying them *valid*.

Example 1 (Sample rule-base). Consider a game where a player starts researching the *Technology* Blast_Furnace at a Blacksmith, connected via the *TECH_AT_Blacksmith* relation, using the COMMAND_RESEARCH_BLAST_FURNACE activity. We model this with a single activity $a1$ and attribute rule $r1$. Specifically, we assert $\mathsf{name}(a1, \text{COMMAND\_RESEARCH\_BLAST\_FURNACE})$, $\mathsf{targets}(r1, a1)$, $\mathsf{requires}(r1, c)$, $\mathsf{gCond}(c, n_x : \textit{Technology})$, and $\mathsf{aType}(r1, \textit{research\_starts\_at})$. The current event timestamp is available through `Time`, so we define the research start time as the current event's time using $\mathsf{aValueQuery}(r1, \texttt{\$time})$. ◁

3.4 Extensional Predicate Evaluations

Goal evaluation of a rule r is performed over G_S with respect to a correlation set $V \subseteq \mathcal{ID}$ of object identifiers and an event timestamp $\tau \in \mathbb{T}$. This context establishes the initial parameter binding environment $\theta = [\texttt{\$corr} \mapsto V, \texttt{\$time} \mapsto \tau]$. Since compiled queries already enforce structural constraints and type compatibility, evaluation using $[\![\cdot]\!]_{G_S,\theta}$ directly yields the predicate extensions (the set of satisfying facts):

- **Object rules:** For an object rule with condition $C \in \mathbb{GQ}^C$, the extension is exactly the returned set: $\mathsf{cSatUnary}_r = [\![Q(C)]\!]_{G_S,\theta}$.

- **Relation rules:** Target object pairs (n_x, n_y) are drawn from $V \times V$ if the rule is internal ($\mathsf{isInternal} = \top$), or from either $V \times (\mathcal{ID} \setminus V)$ or $(\mathcal{ID} \setminus V) \times V$ if the rule is external ($\mathsf{isInternal} = \bot$). Assuming that the rule targets $R \in \mathbb{R}$, the template validates node types against $\mathsf{R}_X(R)$ and pre-filters structural conflicts (e.g., ensuring $(n_x) \xrightarrow{R} (n_y)$ is not in G_S for creation rules). Evaluating $C \in \mathbb{GQ}^C$ under θ yields $\mathsf{cSatBinary}_r = [\![\mathcal{Q}(C)]\!]_{G_S,\theta}$.
- **Attribute rules:** Attribute rules first evaluate the 2SSRPQ gPath starting from the correlation set V, producing the set of target objects $\mathsf{pSatUnary}_r$. The condition $C \in \mathbb{GQ}^C$ is then evaluated in the updated environment $\theta' = \theta[\texttt{\$target} \mapsto \mathsf{pSatUnary}_r]$, that is, $\mathsf{cSatUnary}_r = [\![\mathcal{Q}(C)]\!]_{G_S,\theta'}$. Finally, we identify the target objects whose attribute values will change. Let $A \in \mathbb{GQ}^A$ be the rule's attribute query targeting property K. Evaluating A under $\theta'' = \theta'[\texttt{\$target} \mapsto \mathsf{cSatUnary}_r]$ yields a tuple multiset $R_A = [\![\mathcal{Q}(A)]\!]_{G_S,\theta''}$. For each object a, let $R_A|_a = \{(u_1, \ldots, u_k) \mid (a, u_1, \ldots, u_k) \in R_A\}$ be the restriction of R_A to a. Given an external function f:

$$\mathsf{candChanges}_r = \left\{ a \in \mathsf{cSatUnary}_r \;\middle|\; \begin{array}{l} \text{Node } a \text{ lacks property } K \text{ in } G_S \vee \\ \text{its current value } v \neq f(R_A|_a, \tau) \end{array} \right\}$$

  If the attribute rule lacks an external function, the query A must yield a unique attribute value for every object in $\mathsf{cSatUnary}_r$. This uniqueness is structurally ensured by requiring the returned value to be either fixed to a constant in $\mathbb{V}$ or bound to the event timestamp via the `$time` parameter.

3.5 Event Evaluation

In the following, we assume all rule assertions are present in a given rule-base $\mathbb{C}$. Consider an event $e \in \mathcal{E}$ occurring in state G_S. Let $\tau = \mathsf{ts}(e)$ be its timestamp, $\textsc{Act} = \mathsf{act}(e)$ its activity, and define $corr_e = \{\mathrm{id}(o) \in \mathcal{ID} \mid (e, o) \in CORR\}$ as the set of identifiers for objects correlated to e. Additionally, let a be the activity object in $\mathbb{C}$ describing Act.

Event processing invokes the rule evaluations from Section 3.4 using $V = corr_e$ and timestamp τ. First, the precondition T of a (i.e., such that $\mathsf{pre}(a, T)$ is in $\mathbb{C}$) is tested by evaluating $[\![\mathcal{Q}(T)]\!]_{G_S,[\texttt{\$corr}\mapsto corr_e]}$. If this returns false, no correlated object satisfies the precondition and the event is assumed to have no effect. Otherwise, let R be the rules targeting a in $\mathbb{C}$. Execution proceeds in two stages, allowing newly created objects to participate in subsequent relation and attribute updates within the same event:

1. **Object Creation:** Processing *ObjectRule* instances with $\mathsf{createsObj} = \top$, governed by program $\mathsf{P}_{oc}(a)$.
2. **Remaining Effects:** Processing deletion, relation, and attribute rules over the resulting state, governed by program $\mathsf{P}_{rem}(a)$.

For the active logic program $P \in \{\mathsf{P}_{oc}(a), \mathsf{P}_{rem}(a)\}$, we lazily materialize the extensional fact base $\mathcal{F}_{ext}$ exclusively for predicates in $\mathrm{Pred}_{ext}(P)$, bounded by the correlation set $corr_e$ and its query-reachable neighborhood. Let M be the unique answer set of $P \cup \mathcal{F}_{ext}$. The *ASP-refined* goal set of an object or attribute rule r is:

$$goals_r = \begin{cases} \mathsf{cSatUnary}_r, \text{ if } \mathsf{eTarUnary}_r \notin \mathrm{Pred}_h(P), \\ \left\{ a \in \mathsf{cSatUnary}_r \mid \mathsf{eTarUnary}_r(a) \in M \right\}, \text{ otherwise.} \end{cases}$$

where $\mathrm{Pred}_h(P)$ is the set of head predicates of rules in P. For relation rules, goals are defined analogously over binary predicates, yielding a set $goals_r \subseteq \mathcal{ID} \times \mathcal{ID}$ of object pairs.

Finally, the effects for all rules in the current stage are computed and concurrently applied. The effects of a rule r, denoted η_r, are formalized as a set of logical updates over the atomic predicates $\mathbb{AP}$ of the data state $\mathcal{S}$.

For object creation rules, $\eta_r = \{\mathtt{Obj}(a) \mid a \in goals_r\}$. For deletion rules, $\eta_r = \{\neg\mathtt{Obj}(a) \mid a \in goals_r\}$. Relation rules are defined analogously over pairs in $goals_r$. For an attribute rule targeting property K with external function f, $\eta_r = \{K(a, f(R_A|_a, \tau)) \mid a \in goals_r\}$, where $R_A = [\![Q(A)]\!]_{G_\mathcal{S},\theta''}$ and $R_A|_a$ is its restriction to object a.

These logical updates are realized as Cypher `CREATE`, `DELETE`, and `SET` statements, ensuring that the graph interpretation and logical data state remain synchronized before the start of each execution stage and at the end of event processing.

Example 2 (Rule Application). Continuing Example 1, suppose event e_1 at time 1 observes the activity COMMAND_RESEARCH_BLAST_FURNACE, connected to a technology `a` and a blacksmith `b`. Lacking a path query, the rule's condition applies directly to the correlated objects $\{\mathtt{a}, \mathtt{b}\}$. Since `a` is of type *Technology* in $\mathcal{S}$, condition filtration yields the singleton set $\{\mathtt{a}\}$. The aValueQuery is then evaluated for `a`. As e_1 occurs at time 1, the effect *research_starts_at*(a, 1) is added to $\mathcal{S}$. Let event e_2 observe activity $a2$ (COMPLETE_RESEARCH_BLAST_FURNACE), connected to a technology `a` and a blacksmith `b`. Consider two attribute rules: $r2$ tracks the number of researched blast furnaces, and $r3$ records the current research rate. Both require the technology to be related to the blacksmith whose attributes are being set. In $\mathbb{GQ}^C$, this is specified as the condition `EXISTS` $((n_x) \xrightarrow{\textit{TECH\_AT\_Blacksmith}} (:\textit{Technology}))$. Assuming this holds for `b` at execution time $t = 2$, the fact $\mathsf{cSatUnary}_{r2}(\mathtt{b})$ is derived. To avoid redundantly evaluating $r3$'s condition, $\mathsf{P}_{rem}(a2)$ contains the single clause $\mathsf{eTarUnary}_{r3}(x) \leftarrow \mathsf{cSatUnary}_{r2}(x)$. Evaluating this yields the answer set $\{\mathsf{cSatUnary}_{r2}(\mathtt{b}), \mathsf{eTarUnary}_{r3}(\mathtt{b})\}$. Without a derived $\mathsf{eTarUnary}_{r2}$, $r2$ targets `b` via standard condition evaluation, and $\mathsf{func}(r2, \mathsf{increment})$ increments its count. For $r3$, $\mathsf{eTarUnary}_{r3}(\mathtt{b})$ restricts the target to `b`. Its value query (in $\mathbb{GQ}^A$) retrieves the start time of a (1) and current researched count (0), passes them to the external aggregation function $\mathsf{updateRate}(\{\langle 1, 0\rangle\}, 2)$, and yields the new attribute assignment *Blast_Furnace_research_rate*(b, 0.5). ◁

3.6 Output Timeline Model

The generated timeline records lifecycle events that create or delete objects, as well as relations between objects and from objects to attributes. Unlike conventional Labelled Property Graph (LPG) approaches, we output a set of facts that conforms to the UML Class Diagram $\mathbb{O}_T$ shown in Figure 2, which enables via the correspondence to OWL 2 a standardized serialization as RDF Knowledge Graph (similarly to what done for the input schema $\mathbb{O}_R$).

Structure of a Timeline. In $\mathbb{O}_T$, any time-varying entity is modeled as an instance of *TemporalThing*, which is partitioned into *Object* and *Relation*, whose instances

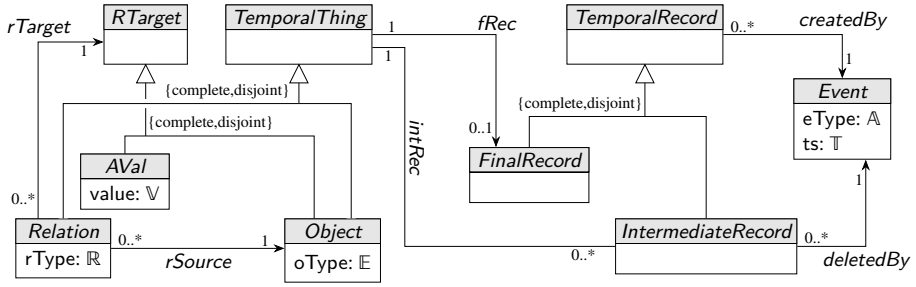

Fig. 2. UML timeline generation model $\mathbb{O}_T$

respectively associate two objects, and an object with an attribute value *AVal*. State transitions are tracked via *TemporalRecord* instances:

- *Lifecycle records* connect a *TemporalThing* to its generating event via createdBy. Records closed by a deletion event via deletedBy are *IntermediateRecord* instances, whereas open records active at timeline termination are *FinalRecord* instances.
- *Attribute updates* for an attribute property K of object x from value v to value v' are modeled as two concurrent relation lifecycle changes: one deleting $x \xrightarrow{K} v$ and one creating $x \xrightarrow{K} v'$.

By embedding generating events, $\mathbb{O}_T$ seamlessly integrates with object-centric event log standards supporting dynamic attributes, such as OCEL 2.0.

Projection to Temporal Property Graphs and Dual Querying. While $\mathbb{O}_T$ explicitly captures causal event links, standard Temporal Property Graphs (TPGs) [11] track history strictly through validity time intervals $[\tau_{\text{start}}, \tau_{\text{end}})$, inherently losing event-level provenance under concurrent updates. Because $\mathbb{O}_T$ is strictly more expressive, the generated timeline KB D seamlessly projects into a TPG: *(i)* an *IntermediateRecord* bounded by e_1 and e_2 projects to $[\mathsf{ts}(e_1), \mathsf{ts}(e_2))$; *(ii)* a *FinalRecord* created by e_1 projects to $[\mathsf{ts}(e_1), \infty)$; and *(iii)* static entities without a createdBy event project to $[\tau_0, \infty)$, where τ_0 is the earliest timestamp of an event in the TPG.

This projection bridges two distinct query paradigms: D can be queried directly by relying on SPARQL and Semantic Web technologies, using the vocabulary defined by $\mathbb{O}_T$, while its projected interval structure allows the use of graph querying mechanisms such as Cypher and T-GQL, to execute complex queries [11]. We discuss the physical realization of this hybrid schema in Section 5.

4 Timeline Generation Algorithm

We present a timeline generation algorithm (Algorithm 1) that takes as inputs an event log G and a valid rule-base $\mathbb{C}$ (hence satisfying schema $\mathbb{O}_R$ and the structural conditions $\mathbb{I}_R$), and produces a timeline D consistent with $\mathbb{O}_T$. We begin by constructing event groups that can be processed in parallel. To do so, we construct a rule dependency graph

Algorithm 1 Global Timeline Generation

procedure MAIN(G, $\mathbb{C}$)
  $rGraph \leftarrow$ BUILDEXTENDEDRULEDEPENDENCYGRAPH($\mathbb{C}$)
  $aGraph \leftarrow$ COLLAPSEINTOTARGETACTS($rGraph$)
  $sccGraph \leftarrow$ COLLAPSEINTOSCCGRAPH($aGraph$)
  $indEventGroups \leftarrow$ EXPANDINTOEVENTCOMPONENTS($sccGraph, G$)
  $orderedEvents \leftarrow$ ORDEREVENTS($ind\,EventGroups$)
  $G_S \leftarrow$ POPULATEINITIALGRAPH(G)
  $D \leftarrow$ INITIALIZETIMELINE(G_S)
  for $eGroup \in orderedEvents$ **in parallel do**
    for $event \in eGroup$ **do**
      if SATISFIESPRECONDITIONS($event, \mathbb{C}$) **then**
        Let a be such that $\mathsf{name}(a, \mathsf{act}(event)) \in \mathbb{C}$
        Let P_1, P_2 be such that $\mathsf{P}_{oc}(a, P_1)$ and $\mathsf{P}_{rem}(a, P_2)$ are in $\mathbb{C}$
        $rules \leftarrow \{r \mid \mathsf{targets}(a, r) \in \mathbb{C}\}$
        $results_1 \leftarrow$ EVALUATEEVENT($event, P_1, \{r \mid r \in rules \wedge \mathit{ORule}(r) \in \mathbb{C}\}, G_S$)
        $G_S \leftarrow$ RECORDEFFECTS($G_S, results_1, D, event$)
        $results_2 \leftarrow$ EVALUATEEVENT($event, P_2, \{r \mid r \in rules \wedge \mathit{ORule}(r) \notin \mathbb{C}\}, G_S$)
        $G_S \leftarrow$ RECORDEFFECTS($G_S, results_2, D, event$)

where a directed edge $r_1 \rightarrow r_2$ indicates that r_1 produces an effect (an object, relation, or attribute type) that is consumed by r_2. We collapse this into an activity-level dependency graph and partition its strongly connected components into disconnected subgraphs. Because effects cannot propagate across disconnected components, their corresponding event groups are causally independent. Expanding these components over the input log G yields independent event streams that can be evaluated in parallel.

Following this, the initial graph interpretation G_S of the data state is obtained as a view over G by removing event nodes, non-static typed objects, and their incident edges, leaving only static objects, their relations, and initial attribute values. We then initialize the timeline D by instantiating *TemporalThing* for each node and edge in G_S. Because static entities exist prior to the log's execution, their corresponding *TemporalThing* instances are created without an associated createdBy event, representing entities active from the start of the system execution encoded in G. Following this setup, the timeline construction begins. As noted above, event groups are processed in parallel. Within each event group, events are processed strictly in timestamp order to maintain causality, ensuring correspondence with the fully serial timeline construction approach [8]. For each event, the active ASP programs are retrieved and evaluated according to Section 3.5, updating G_S. The two-stage evaluation yields two sets of state updates (creations, deletions, and attribute modifications), which are recorded as temporal records in D.

5 Implementation and Experimental Evaluation

In this section, we describe results from applying timeline generation to a real-world dataset. For this process, we built a tool[1] that allows us to view and manipulate timelines.

[1] See https://gitlab.inf.unibz.it/Rikayan.Chaki/timelinejava.

In our implementation, the abstract input graph G is realized as an Event Knowledge Graph stored in Neo4j. Rather than materializing each evolving data state as a separate graph, G_S is represented as a view over this graph, whose relations and properties evolve accordingly. All object nodes in the graph are labelled with the reserved label *Object*. The reserved label *Alive* on object nodes is used to indicate the objects present in the current data state. Object creation and deletion therefore correspond to updating labels rather than creating or deleting nodes, allowing deleted objects to participate in later events if re-created. This representation avoids copying graph structures while supporting efficient state updates. To ensure that user-supplied query fragments reason only over the current data state, we require the compiled queries to bind variables only to object nodes carrying the reserved label *Alive* and to relations between such objects. Parallel event groups are executed as separate Neo4j transactions, relying on its ACID (Atomicity, Consistency, Isolation, Durability) guarantees for isolation and consistency.

Importantly, the output timeline D is physically realized within Neo4j as a labelled property graph. To preserve the strict causal semantics of $\mathbb{O}_{\mathbb{T}}$, the graph is not stored as a TPG. Instead, nodes and edges maintain arrays of creating and deleting event identifiers alongside their respective timestamps. This enriched schema natively acts as a superset of the TPG model [11]. Standard TPG temporal queries execute seamlessly over the timestamp arrays, while the retained event identifiers allow for lossless extraction back into a rich timeline compliant with $\mathbb{O}_{\mathbb{T}}$.

5.1 AoE2 Dataset

The AoE2 (Age of Empires 2) dataset [17] consists of event logs of $100,000$ matches of the online game AoE2. The log has been normalized to the OCEL v2.0 schema that is expected by our tool. It has:

1. Object types: *Match*, *Session*, *Player*, *Villager*, *Town_Center*. The remaining object types are various building types present in a match. Their roles in events do not depend on the exact type, but on this overall grouping. We will therefore refer to a type in this category by the *generic Building* type.
2. $2,372,505$ events, each having an identifier, an activity and a timestamp value.
3. $613,953$ objects, each having a type and a unique *id* value.

Preprocessing. As the baseline dataset [17] lacks explicit game mechanics, we preprocessed the log to manifest these. Specifically, QUEUE and RESEARCH activities queue units and technologies at buildings, yet these entities do not appear as explicit objects. To enrich timeline generation, we reify them as *Unit* and *Technology* objects. *Unit*-typed objects are constructed as follows:

- For each event e observing COMMAND_QUEUE_*Unit*, if there is a *Building*-typed object x such that $(e, x) \in CORR$ (there is never more than one), let *xid* be its id. Create a fresh *Unit* object u, set $\mathsf{aVal}(u, visited) = false$, $\mathsf{aVal}(u, sub\_type) = Unit$, and $\mathsf{aVal}(u, of) = xid$, and add (e, u) to $CORR$.
- For each event e observing START_QUEUE_*Unit* or COMPLETE_QUEUE_*Unit*, extract the subtype *Unit* and, if present, the identifier *xid* of a correlated object other than *Match*, *Session*, *Player*, or *Unit* (there is never more than one such object).

– For each extracted pair $(xid, Unit)$, if there is already a *Unit* object u with $\mathsf{aVal}(u, visited) = false$, $\mathsf{aVal}(u, sub\_type) = Unit$, and $\mathsf{aVal}(u, of) = xid$, reuse it; otherwise create such a fresh *Unit* object u. In either case, mark u as visited and add (e, u) to *CORR*.

The *visited* attribute records which units have already been consumed by a start/complete queue event. This reification assumes that, at matching time, a *Unit* instance is uniquely identified by (sub_type, of). An analogous procedure is applied to RESEARCH activities to create *Technology* objects.

Activity Semantics. Expanding on the simplified RESEARCH rules from Example 1, we formalize the assumed effects for other key activities using public game documentation. For simplicity, we assume all types except the reified *Unit* and *Technology* are *static*. The full derived rule-base is available in our repository. Below, we outline some of the rules, using $REL(x, y)$ to denote a directed relation from x to y:

– START_QUEUE_*Villager* with a *Villager* v and a *Town_Center* t (optional): Delete a *QUEUE_TRAINING_AT* relation from v to t, if it is not currently being built. If successful, create a *TRAINING_AT* relation in its place. If no town center is specified, find one connected to v by a *QUEUE_TRAINING_AT* relation, and replace it with a *TRAINING_AT* relation.
– COMMAND_RESEARCH_*Technology* with a *Player* p, a *Technology* t (optional), and a *Building* b (optional): Create t, a *TRY_RESEARCH* relation from p to t; create a *TECH_AT_Building* relation from t to b; if unset, set t.*research_starts_at* to current time.
– START_RESEARCH_*Technology* with a *Technology* t and a *Building* b (optional): Find a player p connected to t; replace $TRY\_RESEARCH(p, t)$ with $DO\_RESEARCH(p, t)$.
– COMPLETE_RESEARCH_*Technology* with a *Technology* t and a *Building* b (optional): Find all players p connected to t; delete $DO\_RESEARCH(p, t)$; increment *Technology_type_researched* by 1 and update *Building_type_research_rate*.

We use the above activity ordering to disambiguate event timestamps in the source data that have the same value. Thus, for example, it is assumed that if two events in the source log have types COMPLETE_QUEUE_*Unit* and START_QUEUE_*Unit*, and have the same timestamp, the former happened before the latter.

5.2 Rule Reuse

To understand how rule dependencies can be utilized, we consider how the domain knowledge deduced above can be translated into concrete rules for the START_QUEUE_*Villager* activity. One of the tasks in this activity is the deletion of the *QUEUE_TRAINING_AT* relation by an *external* rule. Let this rule be $r1$. Translating the semantics of this activity and considering the fact that deletion rules require an existing relation of the target type between the object pair being considered, we obtain the following condition:

$$\texttt{EXISTS}((n_x) \xrightarrow{QUEUE\_TRAINING\_AT} (n_y))\ \texttt{AND NOT EXISTS}(() \xrightarrow{BUILDS\_Town\_Center} (n_y))$$

Deleting *TRAINING_AT* using an external relation requires checking this condition and ensuring the *BUILDS* relation does not hold. Clearly, this rule can be rewritten to use the

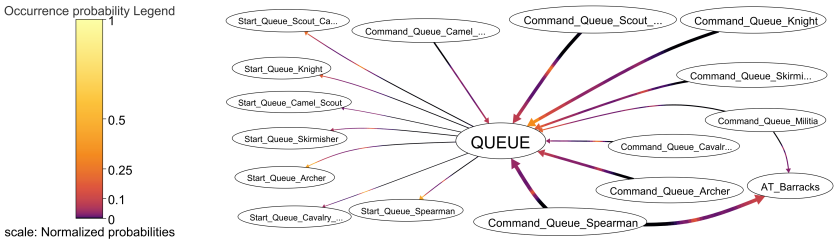

Fig. 3. Part of Visualized Timeline

object pairs filtered by $r1$. Let $r2$ be the rule that creates the *TRAINING_AT* relations. Consider the following logic program: $\mathsf{eTarBinary}_{r2}(x, y) \leftarrow \mathsf{cSatBinary}_{r1}(x, y)$. We observe that its evaluation produces an answer set that contains $\mathsf{eTarBinary}_{r2}$ which has object pairs that are filtered by rule $r1$'s condition. Thus, we can observe how, by utilizing these ASPs, we avoided additional computations where they are not necessary.

5.3 Timeline Results and Update Occurrence Probability Visualizer

The timeline is output to the Neo4j source database, enabling Cypher querying. It can also be imported or exported as JSON or as a knowledge graph under the $\mathbb{O}_T$ schema. An interactive diagram visualizes update probabilities across the event log using nodes for activities, object types, relation types, and attributes. Edges from an activity ACT to an object type X denote object creations, while edges in the opposite direction denote object deletions. The meaning of edges between relation/attribute types and activities is defined analogously. Edge thickness reflects the frequency of creations, while a color gradient maps the creation probability over time, estimated via Gaussian kernel smoothing of update timestamps for each pair of activity and object/relation/attribute type.

Figure 3 visualizes the AoE2 dataset timeline. We observe that *QUEUE* creations (in COMMAND_QUEUE_*Unit*) are more frequent and occur at about the same time as their deletions (in START_QUEUE_*Technology*). This indicates that these relations are created for very short periods of time, and not all create *QUEUE* activities are deleted by START_QUEUE activities. Additionally, edge coloring reveals a concentration of activity near the end of the event log. We confirmed this by querying the log across five equal time windows, verifying that event density is significantly higher in the later timeframes.

5.4 Benchmarking

We evaluated 10,000 arbitrarily chosen matches on an Intel Core i7-1165G7 (16GB RAM) running Windows 11 Pro, Java OpenJDK 22.0.1, and Neo4j 2025.10.1 Enterprise. We benchmarked two rule versions for the AoE dataset, denoted 'simple' and 'complex', with the former excluding logic programs/preconditions, and the latter including them while maintaining the former's semantics. Figure 4 illustrates the average execution times for evaluating events associated with specific activity types, such as COMMAND_QUEUE or START_RESEARCH. Observe that activities like START_RESEARCH show significant

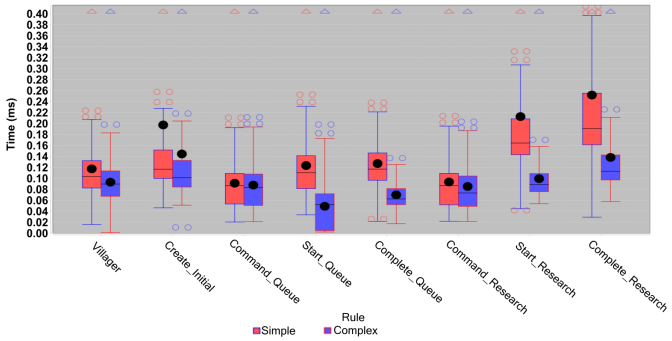

Fig. 4. Average Rule Execution times, grouped by activity kinds

performance gains due to rule reuse (Example 1), whereas activities without rule reuse (such as the COMMAND_RESEARCH activities) do not.

Following this, we also ran the series and parallel algorithms five times each for the complex version of the rules, resulting in median computation times of 2,381.87 and 1,138.13 seconds, respectively. The 95% confidence intervals for the two algorithms over these runs are [2,239.95, 2,509.42] and [1,000.82, 1,320.46], respectively. The observed performance improvement stems from the concurrent execution of the research and query rule classes. As these two classes contain the vast majority of log events, this substantial speedup aligns with expected parallelization gains.

6 Conclusions

This paper advanced domain knowledge-based timeline generation from object-centric event logs. We have adopted non-recursive logic programs to simplify rule specification and improve performance, and have developed a parallelized timeline construction algorithm that demonstrates significant performance gains. To support this methodology, we have implemented an interoperable, interactive visualization tool capable of exploring schema-level data update frequencies. Future work will focus on enriching our currently generic rule effects by integrating domain knowledge expressed as an OWL 2 QL ontology [18] and accessing event data following the virtual knowledge graph paradigm [20], in line with what done in the onprom approach [6, 7]. This will also allow us to move beyond the generic creation and deletion that is presented here.

Acknowledgments. This research has been partially supported by the HEU project CyclOps (GA n. 101135513), by the Province of Bolzano and FWF through project OnTeGra (DOI 10.55776/PIN8884924), by the Province of Bolzano and EU through projects ERDF-FESR 1078 CRIMA, and ERDF-FESR 1047 AI-Lab, and by MUR through the PRIN project 2022XERWK9 S-PIC4CHU.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley Publ. Co. (1995)
2. Berardi, D., Calvanese, D., De Giacomo, G.: Reasoning on UML class diagrams. Artificial Intelligence **168**(1–2), 70–118 (2005). https://doi.org/10.1016/j.artint.2005.05.003
3. Berti, A., Koren, I., Adams, J.N., Park, G., Knopp, B., Graves, N., Rafiei, M., Liß, L., Unterberg, L.T.G., Zhang, Y., et al.: OCEL (Object-Centric Event Log) 2.0 specification. CoRR Technical Report arXiv:2403.01975, arXiv.org e-Print archive (2024), https://arxiv.org/abs/2403.01975
4. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Communications of the ACM **54**(12), 92–103 (2011). https://doi.org/10.1145/2043174.2043195
5. Calvanese, D., Artale, A., Kalayci, T.E., Montali, M., Santoso, A.: onprom: An OBDA-based event log extraction tool suite (2020), https://github.com/onprom/onprom, [Accessed 15-12-2023]
6. Calvanese, D., Jans, M., Kalayci, T.E., Montali, M.: Extracting event data from document-driven enterprise systems. In: Proc. of the 35th Int. Conf. on Advanced Information Systems Engineering (CAiSE). Lecture Notes in Computer Science, vol. 13901, pp. 193–209. Springer (2023). https://doi.org/10.1007/978-3-031-34560-9_12
7. Calvanese, D., Kalayci, T.E., Montali, M., Santoso, A.: OBDA for log extraction in process mining. In: Reasoning Web: Semantic Interoperability on the Web – 13th Int. Summer School Tutorial Lectures (RW), Lecture Notes in Computer Science, vol. 10370, pp. 292–345. Springer (2017). https://doi.org/10.1007/978-3-319-61033-7_9
8. Chaki, R., Calvanese, D.: Timeline generation from event logs under evolving properties. In: Proc. of the 28th Int. Symp. on Methodologies for Intelligent Systems (ISMIS). Lecture Notes in Computer Science, vol. 16685, pp. 35–45. Springer (2026). https://doi.org/10.1007/978-3-032-32643-0_4
9. Chaki, R., Calvanese, D., Montali, M.: Generation of timelines from event knowledge graphs using domain knowledge. In: Proc. of the 24th Int. Conf. on Computer Information Systems and Industrial Management Applications (CISIM). Lecture Notes in Computer Science, vol. 15927, pp. 275–289. Springer (2025). https://doi.org/10.1007/978-3-032-02406-0_20
10. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Computing Surveys **33**(3), 374–425 (2001). https://doi.org/10.1145/502807.502810
11. Debrouvier, A., Parodi, E., Perazzo, M., Soliani, V., Vaisman, A.: A model and query language for temporal graph databases. The VLDB J. **30**(5), 825–858 (2021). https://doi.org/10.1007/S00778-021-00675-4
12. Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: 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). https://doi.org/10.1145/3183713.3190657
13. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. Theory and Practice of Logic Programming **19**(1), 27–82 (2019). https://doi.org/10.1017/S1471068418000054
14. Ghahfarokhi, A.F., Park, G., Berti, A., van der Aalst, W.M.P.: OCEL: A standard for object-centric event logs. In: New Trends in Database and Information Systems – ADBIS 2021 Short Papers, Doctoral Consortium and Workshops. Communications in Computer and Information Science, vol. 1450, pp. 169–175. Springer (2021). https://doi.org/10.1007/978-3-030-85082-1_16
15. Guan, S., Cheng, X., Bai, L., Zhang, F., Li, Z., Zeng, Y., Jin, X., Guo, J.: What is Event Knowledge Graph: A survey. IEEE Trans. on Knowledge and Data Engineering **35**(7), 7569–7589 (2023). https://doi.org/10.1109/TKDE.2022.3180362

16. Khayatbashi, S., Hartig, O., Jalali, A.: Transforming object-centric event logs to temporal event knowledge graphs. In: Revised Selected Papers of the BPM 2024 International Workshops. Lecture Notes in Business Information Processing, vol. 534, pp. 300–313. Springer (2024). https://doi.org/10.1007/978-3-031-78666-2_23
17. Liss, L., Elbert, N., Flath, C.M., van der Aalst, W.M.: Object-centric event log for Age of Empires Game interactions (Aug 2024). https://doi.org/10.5281/zenodo.13365584
18. Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: OWL 2 Web Ontology Language profiles (second edition). W3C Recommendation, World Wide Web Consortium (Dec 2012), https://www.w3.org/TR/owl2-profiles/
19. Neo4j, Inc.: Neo4j, version 2025.04. https://neo4j.com/ (2025)
20. Xiao, G., Calvanese, D., Kontchakov, R., Lembo, D., Poggi, A., Rosati, R., Zakharyaschev, M.: 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