Process Mining over Multiple Behavioral Dimensions with Event Knowledge Graphs

Dirk Fahland([✉])

Eindhoven University of Technology, Eindhoven, The Netherlands
d.fahland@tue.nl

Abstract. Classical process mining relies on the notion of a unique case identifier, which is used to partition event data into independent sequences of events. In this chapter, we study the shortcomings of this approach for event data over *multiple entities*. We introduce *event knowledge graphs* as data structure that allows to naturally model behavior over multiple entities as a network of events. We explore how to construct, query, and aggregate event knowledge graphs to get insights into complex behaviors. We will ultimately show that event knowledge graphs are a very versatile tool that opens the door to process mining analyses in multiple behavioral dimensions at once.

Keywords: Event knowledge graph · Process mining

1 Introduction—A Second Look at Processes

Process mining aims at analyzing processes from recorded event data. Thereby, the actual processes are rather complex and emerge from the interplay of multiple inter-related *entities*: the various *objects* handled by the process as well as the *organizational entities* that execute the process. We best explain this kind of interplay by an example.

1. Consider a retailer who took two *Orders* for multiple *Items* from the same customer: the customer first places Order $O1$ for 2 items X and 1 item Y, and shortly afterwards Order $O2$ for 1 item X and 1 item Y. The retailer promises to ship every order within 6 days.

The retailer handles both orders as explained next and illustrated in Fig. 1.

2. Items X are provided by supplier A while items Y are provided by supplier B. To save costs, *workers* of the retailer bundle the orders for the items and place two *Supplier Orders*, one at A for 3 items X and one at B for 1 item Y. Suppliers ensure to deliver their products within 3 days of placing the order.
3. Invoice $I2$ for Order $O2$ is created right after placing the supplier order at B.
4. When the retailer receives the Supplier Order from A, workers unpack three Items X one by one and store them in an *automated warehouse* until needed for shipment. At this point, workers also create *Invoice* $I1$ for $O1$.

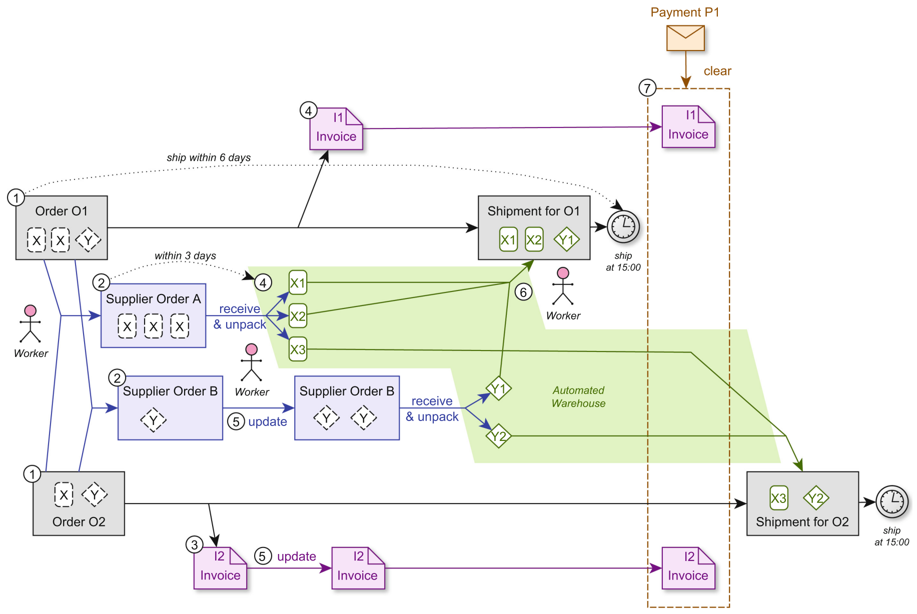

Fig. 1. Illustration of a multi-entity process: a retailer handles two orders for multiple items by placing and receiving supplier orders for specific items.

5. Around the time of receiving the supplier order from A, a *worker* notices they made a mistake: they ordered only one item Y from B while $O1$ and $O2$ both require one item Y *each*. The worker updates the Supplier Order $O2$ and invoice $I2$ accordingly.
6. When finally the supplier order from B is received, the items Y are unpacked. One item Y is stored in the warehouse while the other item Y is packed together with two items X taken from the *warehouse* into the shipment for $O1$. Packed shipments are picked up for delivery every day at 15:00.
7. The retailer has the policy that they only ship to a customer if there is at most one unpaid invoice. Thus, packing and shipment of $O2$ (another item X and the second Y) are delayed until *Payment* $P1$ is received which covers the amount for both invoices $I1$ and $I2$.

This process relies on 7 different types of entities. *Actors* (human workers) and *machines* (an automated warehouse) together handle 5 types of objects: *Orders*, *Supplier Orders*, *Items*, *Invoices*, *Payments*.

Challenges Due to Event Data over Multiple Entities. A process mining analysis of the above process execution relies on recorded event data. Each event has to record in its attributes at least (1) which *action* (or activity) has been executed (2) at which *time*. To construct an event log, classical process mining also expects each event to record (3) in which process execution, typically called

Table 1. Event table of events underlying the event log of Table 2.

EventID	Activity	Time	Actor	Order	Supplier Order	Order Details	Item	Invoice	Payment
e1	Create Order	01-05 09:05	R1	O1		2·X, 1·Y			
e2	Create Order	01-05 09:30	R1	O2		1·X, 1·Y			
e3	Place SO	01-05 11:25	R1		A	3·X			
e4	Place SO	02-05 11:55	R3		B	1·Y			
e5	Create Invoice	03-05 16:15	R3	O2				I2	
e6	Receive SO	00-01 10:00	R2		A		X1,X2,X3		
e7	Update SO	04-05 10:25	R1	O2	B	2·Y			
e8	Unpack	00-01 10:30	R2		A		X3		
e9	Update Invoice	04-05 10:50	R2					I2	
e10	Unpack	04-05 11:00	R2		A		X1		
e11	Unpack	04-05 11:15	R2		A		X2		
e18	Create Invoice	06-05 14:35	R3	O1				I1	
e19	Receive SO	07-05 10:10	R2		B		Y1,Y2		
e20	Unpack	07-05 10:45	R2		B		Y1		
e21	Unpack	07-05 11:00	R2		B		Y2		
e27	Pack Shipment	07-05 17:00	R4	O1			X1,X2,Y1		
e28	Ship	08-05 15:00	R4	O1					
e29	Receive Payment	09-05 08:30	R5						P1
e30	Clear Invoice	09-05 08:45	R5					I1,I2	P1
e33	Pack Shipment	09-05 11:45	R4	O2			X3,Y2		
e34	Ship	09-05 15:00	R4	O2					

case, the event occurred (see [13], Sect. 2). Table 1 shows the events related to the above example.

1. create *Orders* in e_1, e_2;
2. create *Supplier Orders* in e_3, e_4;
3. create *Invoice* $I2$ in e_5;
4. receive *Supplier Order* from A in e_6 and unpack *Items* X in e_8, e_{10}, e_{11}, and create *Invoice* $I1$ in e_{18};
5. update *Supplier Order* for B in e_7 and *Invoice* $I2$ in e_9;
6. receive *Supplier Order* for B in e_{19} and unpack *Items* Y in e_{20}, e_{21}, and pack and ship *Order* $O1$ in e_{27}, e_{28};
7. receive *Payment* $P1$ and clear *Invoices* $I1, I2$ in e_{29}, e_{30} and finally pack and ship *Order* $O2$ in e_{33}, e_{34}.

In contrast to classical event logs, Table 1 contains no typical case identifier attribute by which each event is related to one specific process execution. Instead, we see multiple sparsely filled attributes identifying *multiple entities* of various types: *Order* ($O1, O2$), *Supplier Order* (A, B), *Item* ($X1, X2, X3, Y1, Y2$), *Invoice* ($I1, I2$), and *Payment* ($P1$).

This makes it difficult to construct an *event log* which is the basis for process mining analysis. Recall that to obtain a classical event log we select one *case identifier* attribute. Then all events referring to the same case id and ordered by time form the *trace* of this case, that is, one process execution. In this way, classical event logs partition the recorded behavior into multiple process executions. Process mining techniques then identify frequent patterns shared by all process executions, or identify outliers and deviations of specific process executions.

However, what exactly *is* a process execution in our example? It is not just all events related the one particular entity. For instance, if we chose *Order* as case identifier, we would obtain traces $\langle e_1, e_{18}, e_{27}, e_{29}\rangle$ for $O1$ and $\langle e_2, e_5, e_7, e_{33}, e_{34}\rangle$ for $O2$. These traces do reveal that both orders were not shipped within 6 days as intended by the supplier. However, they do not allow us to understand the cause for this as they clearly do not describe the entire behavior shown in Fig. 1. We could try to group all events into traces using *multiple related case identifiers*. However, we will see in Sect. 3 that doing so introduces false behavioral information called *convergence* and *divergence* [41,45] in the resulting event log leading to false analysis results (see [1], Sect. 3)

False behavioral information arises when flatting Table 1 into sequential traces because we *cannot* partition the entities $O1, O2, A, B, X, Y, I1, I2, P1$ into disjoint sets, each belonging to one process execution that is independent of all others. Rather, the behavior itself is a larger "fabric" of multiple entities that are inter-related and inter-twined over time as shown in Fig. 1. This "fabric" is even more complex as individual *Actors* ($R1, \ldots, R5$) are specialized in specific activities across multiple different entities, e.g., $R2$ specializes receiving, updating, and unpacking *Supplier Orders* and handling *Items*. In the following, we explain how to analyze this very "fabric" of multiple inter-related entities as a whole from a simple event table over multiple entity identifiers such as Table 1.

A Graph-Based Approach. Our trick will be to slightly adapt the existing definitions for obtaining an event log from an event table: instead of constructing entire traces related to a single case identifier, we discuss in Sect. 3 a *local directly-follows* relation for each *individual* entity in the data. Each event can be part of multiple such directly-follows relations, depending on to how many entities it is correlated. We then use the model of *labeled property graphs* in Sect. 4 to create an *event knowledge graph* having events as nodes and the local directly-follows relations as edges between events. We obtain a graph similar to what is shown in Fig. 1, but with precise semantics for events and behavioral information.

A path of directly-follows edges over events related to the same entity is similar to a classical trace. However in an event knowledge graph, such paths meet whenever an event is related to more than one entity, where in an event log each trace is disjoint from all others. We explain in Sect. 5 how to interpret and analyze behavioral information in event knowledge graphs. We show how basic *querying* on event knowledge graphs gives insights into complex behavioral properties.

We show how *aggregation* on event knowledge graphs allows to construct multi-entity process models that better describe such processes.

We finally explore the versatility of event knowledge graphs beyond the control-flow perspective in Sect. 6. We show how event knowledge graphs naturally integrate the control-flow perspective and the *actor perspective*. Querying for specific structures in the event knowledge graph reveals complex patterns of *task instances* not visible in either perspective alone. Further, we show how event knowledge graphs allow us to take a *system-perspective* (or queueing perspective) to analyze emergent behavior and performance problems across multiple entities. We conclude in Sect. 7 with an outlook on the various applications areas of event knowledge graphs in process mining, and on open research challenges.

All concepts for constructing and analyzing event knowledge graphs presented in this chapter are implemented as Cypher queries on the graph database system Neo4j[1] at https://github.com/multi-dimensional-process-mining/event graph_tutorial [28].

2 Multi-entity Event Data

Before we discuss problems and solutions for analyzing event data over multiple entities, we first define what "event data over multiple entities" actually is.

2.1 Events

We assume all data to be given in a single event table. Data is recorded from a universe of values Val; timestamps $Val_{time} \subseteq Val$ are totally ordered by $\leq$.

Definition 1 (Event Table). *An* event table $T = (E, Attr, \#)$ *is a set* E *of events, a set* $Attr$ *of* attribute names *with* $act, time \in Attr$. *Partial function* $\# : E \times Attr \nrightarrow Val$ *assigns an event* $e \in E$ *and an attribute name* $a \in Attr$ *to a value* $\#_a(e) = v$; $\#_a(e) = \perp$ *if* a *is undefined for* e.

Each event $e \in E$ *records an activity and a timestamp, i.e.,* $\#_{act}(e) \neq \perp$ *and* $\#_{time}(e) \in Val_{time}$.

We write $e.a = v$ for $\#_a(e) = v$ as a shorthand. An event table specifically allows multi-valued attributes, e.g., sets of values $\#_a(e) = \{v_1, v_2, v_3\}$ or a list of values $\#_a(e) = \langle v_1, v_2, v_3, v_1 \rangle$.[2] Simplifying notation, we also may write $v \in e.a$ if $e.a = v$ or if $e.a = \langle \ldots, v, \ldots \rangle$.

An event table only defines $e.activity$ and $e.time$ attributes for each event. The special characteristic of event data over multiple entities is that it does not record a unique case identifier attribute, but identifiers of *multiple entity types*.

Definition 2 (Event table with entity types). *An* event table with entities types $T = (E, Attr, \#, ENT)$ *additionally designates one or more attributes* $\emptyset \neq ENT \subseteq Attr$ *as names of* entity types.

[1] neo4j.com.

[2] We assume the values in an event table to be consistent with some data model that is specified elsewhere. Our subsequent discussion does not rely on it.

A classical event log corresponds to an event table with a single entity type $\mathit{ENT} = \{\mathit{case}\}$. We can consider Table 1 is an event table with entity types $\mathit{ENT} = \{\mathit{Resource}, \mathit{Order}, \mathit{Supplier\ Order}, \mathit{Item}, \mathit{Invoice}, \mathit{Payment}\}$.

Event tables (Definition 1) are also called raw event logs and are – besides relational data – the most common form of input to process mining. The entity types of Definition 2 can be retrieved from an event table through schema recovery techniques [46]. Note that Definition 2 formalizes the object-centric event logs (OCEL) described in Sect. 3.4 of [1]; we here use the more general term "entity" instead of "object" as we will later study behavior over entities which are not tangible objects.

Event tables do not model the ordering of events with respect to a case or an entity which is needed for process mining. Before we study the ordering of events, we explain how events relate to entities.

2.2 Entities and Correlated Events

Each entity type $\mathit{ent} \in \mathit{ENT}$ is a column in the event table T. Each value in that column refers to a specific entity.

Definition 3 (Entities). *Let $T = (E, \mathit{Attr}, \#, \mathit{ENT})$ be an event table with entities. Let $\mathit{ent} \in \mathit{ENT}$ be an entity type. The* set of entities *in T of type ent is $\mathit{Entities}(\mathit{ent}, T) = \{n \mid \exists e \in E : n \in e.\mathit{ent}\}$.*

From Table 1 we identify 6 entity types with corresponding entities: (1) Order: $\{O1, O2\} = \mathit{Entities}(\mathit{Order}, T)$, (2) Supplier Order: A, B, (3) Item: $X1, X2, X3$ and $Y1, Y2$, (4) Invoice: $I1, I2$, (5) Payment: $P1$, (6) Resource: $R1 - R5$ (see Definition 3).

An event $e \in E$ which has a value $n = e.\mathit{ent}$ or $n \in e.\mathit{ent}$ is *correlated* to entity n.

Definition 4 (Correlation). *Let $T = (E, \mathit{Attr}, \#, \mathit{ENT})$ be an event table with entities. Let $n \in \mathit{Entities}(\mathit{ent}, T)$ be an entity of type $\mathit{ent} \in \mathit{ENT}$.*

Event e is correlated to *entity n, written $(e, n) \in \mathit{corr}_{\mathit{ent},T}$ iff $n = e.\mathit{ent} \vee n \in e.\mathit{ent}$. We write $\mathit{corr}(n, \mathit{ent}, T) = \{e \in E \mid (e, n) \in \mathit{corr}_{\mathit{ent},T}\}$ for the set of events correlated to entity $n \in \mathit{Entities}(\mathit{ent}, T)$.*

For example, for Table 1, event e_{30} is correlated to $I1$, $I2$, and $P1$, i.e., $(e_{30}, I1), (e_{30}, I2) \in \mathit{corr}_{\mathit{Invoice},T}$ and $(e_{30}, P1) \in \mathit{corr}_{\mathit{Payment},T}$. The events correlated to $I2$ are $\mathit{corr}(I2, \mathit{Invoice}, T) = \{e_5, e_9, e_{30}\}$. In case the entity identifiers used by different entity types are disjoint, e.g., there are not an Order $O3$ and an Item $O3$, we can omit entity types and just write $(e_{30}, I1), (e_{30}, I2), (e_{30}, P1) \in \mathit{corr}_T$ and $\mathit{corr}(I2, T) = \{e_5, e_9, e_{30}\}$.

Correlation lifts to a *set* N of entities by union: $\mathit{corr}(N, T) = \bigcup_{n \in N} \mathit{corr}(n, T)$. We will later use this to collect events of (transitively) related entities, which we discuss next.

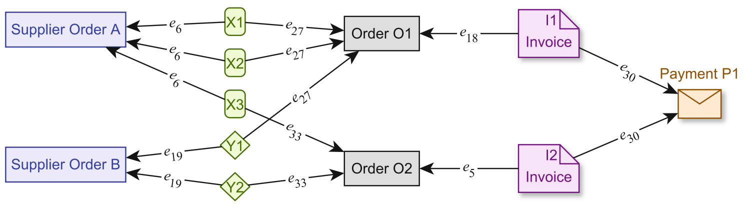

Fig. 2. Relations between entities derived from Table 1

2.3 Relations Between Entities

We now make a first important observation. Although our data only defines entity types explicitly, it *implicity* defines *relations between entity types*. A record e in Table 1 containing two identifiers n_1, n_2 of two different types implicitly relates n_1 and n_2. For example, event e_5 defines that $e_5.\mathit{Order} = O2$ is related to $e_5.\mathit{Invoice} = I2$ and event e_{18} defines that $e_{18}.\mathit{Order} = O1$ is related to $e_{18}.\mathit{Invoice} = I1$. We can write this as a relation $R_{(\mathit{Invoice},\mathit{Order})} = \{(O1, I1), (O2, I2)\}$.

Definition 5 (Relation). *Let* $T = (E, \mathit{Attr}, \#, \mathit{ENT})$ *be an event table with entities. Let* $\mathit{ent}_1, \mathit{ent}_2 \in \mathit{ENT}$ *be two entity types. The* relation between ent_1 and ent_2 in T *is* $R_{(\mathit{ent}_1,\mathit{ent}_2)} = \{(e.\mathit{ent}_1, \mathit{ent}_2) \mid e.\mathit{ent}_1 \neq \perp, e.\mathit{ent}_2 \neq \perp\}$.

Note that Definition 5 does not impose the direction of a relation. Figure 2 visualizes the relations we can derive from Table 1.

Recall that in relational data modeling, each relation $R_{(\mathit{ent}_1,\mathit{ent}_2)}$ has a *cardinality* describing how many entities of type ent_1 are related to each entity of type ent_2, and vice versa. We can infer this cardinality from the tuples in $R_{(\mathit{ent}_1,\mathit{ent}_2)}$ if we assume that the data in the input event table is sufficiently complete. For example, for the relations in Fig. 2,

- $R_{(\mathit{Invoice},\mathit{Order})}$ is a 1-to-1 relation as each invoice is related to one order, and vice versa;
- $R_{(\mathit{Invoice},\mathit{Payment})}$ is an n-to-1 relation as both $I1$ and $I2$ are related to $P1$;
- $R_{(\mathit{Item},\mathit{Order})}$ is an n-to-1 relation as each order has multiple items but each item relates to exactly one order;
- $R_{(\mathit{Item},\mathit{Supplier\ Order})}$ is an n-to-1 relation.

Entities are also transitively related by concatenating or joining the relations on a shared entity typed (and then omitting this shared entity type). For example, $R_{(\mathit{Order},\mathit{Payment})} = R_{(\mathit{Invoice},\mathit{Order})} \bowtie R_{(\mathit{Invoice},\mathit{Payment})} = \{(O1, P1), (O2, P1)\}$ is an n-to-1 relation, and $R_{(\mathit{Order},\mathit{Supplier\ Order})} = R_{(\mathit{Item},\mathit{Order})} \bowtie R_{(\mathit{Item},\mathit{Supplier\ Order})} = \{(O1, A), (O1, B), (O2, A), (O2, B)\}$ is an n-to-m relation.

Entities, relations, and correlation of events can be automatically retrieved from event tables [46] and relational databases [41,43] through schema recovery techniques. However, we have to be aware that relations and their cardinalities recovered according to Definition 5 are a *static* view of the relations obtained by aggregating all observations over time while *a process updates relations dynamically*. For instance, *Order* $O1$ was not related to any *Item* until event e_{27}. Modeling such dynamics requires additional concepts as defined in XOC event logs [39,40]. We have to ignore this aspect in the remainder.

3 Shortcomings of Event Logs over Multi-entity Event Data

Having defined event data over multiple entities, we can now discuss ways of ordering events correlated to a case or an entity, which is the basis for process mining analysis. We first explain how transforming multi-entity data into a classical event log with a single case identifier (Sect. 3.1) introduces false behavioral information leading to false analysis results (Sect. 3.2). We then propose a different approach to ordering events with respect to individual entities (Sect. 3.3).

3.1 Classical Event Log Extraction

We cannot directly turn the event data in Table 1 into a classical event log, because we lack a clear case identifier column that is defined for all events. While *Actor* is an entity identifier defined for all events, it does not group events into the process executions described in Sect. 1. The standard procedure to extract a classical event log from such data is the following (see also Def. 5 of [1] and [13]).

Step 1. Determine relevant entities in the data. An event table with entity identifiers already defines the set of entities in the process (see Definition 3). For extracting an event log for a process execution, we only consider entities that are also handled "along" or "within" a process execution. Thus, we now focus on *Order*, *Supplier Order*, *Item*, *Invoice*, and *Payment* and exclude *Actor*.[3]

Step 2. Pick one entity as case identifier. As the process goal is to complete an order, entity *Order* is our best candidate for a case identifier. This identifier defines two cases: $O1$ and $O2$. However, as most events in Table 1 are not directly correlated to an *Order*, we cannot simply group events by attribute *Order*.

Step 3. Define the set of all entities related to a case. The classical idea is to "enlarge" the scope of the case. We include all entities which are (transitively) related to the case entities $O1$ and $O2$ via the relations we can identify in the event table (see Definition 5 and Fig. 2).

– Order $O1$ is related to Invoice $I1$, Payment $P1$, Items $X1, X2, Y1$, and Supplier Orders A, B, i.e., $\mathit{caseEntities}(O1) = \{O1, I1, P1, X1, X2, Y1, A, B\}$.
– Order $O2$ is related to Invoice $I2$, Payment $P1$, Items $X3, Y2$, Supplier Orders A, B, i.e., $\mathit{caseEntities}(O2) = \{O2, I2, P1, X3, Y2, A, B\}$.

[3] In later sections we will not have to make such a distinction and can consider behavior along any kind of entity.

Step 4. Construct a trace from events of all entities in a case. Each event e correlated to an entity $n \in \mathit{caseEntities}(O1)$ is now also considered as correlated to case $O1$: $\mathit{corr}^*(O1,T) = \mathit{corr}(\mathit{caseEntities}(O1),T)$. For example, for $O1$ we extract from Table 1:

- $\mathit{corr}(O1,T) = \{e_1, e_2, e_{18}\}$
- $\mathit{corr}(I1,T) = \{e_{18}, e_{30}\}$
- $\mathit{corr}(P1,T) = \{e_{29}, e_{30}\}$
- $\mathit{corr}(X1,T) = \{e_6, e_{10}, e_{27}\}$
- $\mathit{corr}(X2,T) = \{e_6, e_{11}, e_{27}\}$
- $\mathit{corr}(Y1,T) = \{e_{19}, e_{20}, e_{27}\}$
- $\mathit{corr}(A,T) = \{e_3, e_6, e_8, e_{10}, e_{11}\}$
- $\mathit{corr}(B,T) = \{e_4, e_7, e_{19}, e_{20}, e_{21}\}$

Taking their union yields $\mathit{corr}^*(O1,T) = \{e_1, e_3, e_4, e_6, e_{10}, e_{11}, e_{18}, e_{19}, e_{20}, e_{27}, e_{28}, e_{29}, e_{30}\}$. We store all events extracted for $O1$ in a new event table where we explicitly set the attribute *Case* to $O1$. In this way, we materialize that each $e_i \in \mathit{corr}^*(O1,T)$ is correlated to $O1$. We repeat this procedure for each case. Table 2 shows the extracted events for $O1$ and $O2$.

Note that this extraction approach can extract the same event *multiple times* for different cases but with a different value for the newly set *Case* attribute. For instance, e_3 and e_{30} are extracted both for $O1$ and for $O2$. This is due to the n-to-m relation between *Order* and *Supplier Order* and the n-to-1 relation between *Payment* and *Order*.

Ordering the extracted events by time in each case results in the *traces* from the viewpoint of $O1$ and from the viewpoint of $O2$ respectively as shown in Tab 2.

Event logs can be automatically extracted in this way from event tables with multiple entity identifiers [46]. Extraction from relational databases succeeds through SQL queries that extract and group events from different tables into traces [35]. These queries can be generated automatically using a variety of techniques [6,7,12,29,35,41]; see [2,13] for a detailed discussion.

3.2 False Behavioral Information in Classical Event Logs

Note that the event log in Table 2 contains numerous *false* behavioral information. Some events were duplicated and occur in both traces, e.g., $e_3, e_4, e_6, e_{19}, e_{29}$, suggesting that in total four *Supplier Orders* were placed and received (while there were only two) and that two *Payments* were received (while there was only one). This is also known as *divergence* [41,41,45,52].

Further, the order of events in both traces gives false behavior information. For instance, in the trace for O2, *Update SO* (e_7) occurs after *Receive SO* (e_6) suggesting a supplier order was updated after it had been received (while this never happened for any Supplier Order). This is also known as *convergence* [41, 45,52].

Where divergence falsifies frequencies of events, convergence falsifies the behavioral information in the directly-follows relation, which is the basis for

Table 2. Classical event log of order process with events extracted for case identifier *Order*.

EventID	Case	Activity	Time	Actor	Order	Supplier Order	Order Details	Item	Invoice	Payment
e1	O1	Create Order	01-05 09:05	R1	O1		2·X, 1·Y			
e3	O1	Place SO	01-05 11:25	R1		A	3·X			
e4	O1	Place SO	02-05 11:55	R1		B	1·Y			
e6	O1	Receive SO	04-05 10:00	R2		A		X1,X2,X3		
e10	O1	Unpack	04-05 11:00	R2		A		X1		
e11	O1	Unpack	04-05 11:15	R2		A		X2		
e18	O1	Create Invoice	06-05 14:35	R3	O1				I1	
e19	O1	Receive SO	07-05 10:10	R2		B		Y1,Y2		
e20	O1	Unpack	07-05 10:45	R2		B		Y1		
e27	O1	Pack Shipment	07-05 17:00	R4	O1			X1,X2,Y1		
e28	O1	Ship	08-05 15:00	R4	O1					
e29	O1	Receive Payment	09-05 08:30	R5						P1
e30	O1	Clear Invoice	09-05 08:45	R5					I1,I2	P1
e2	O2	Create Order	01-05 09:30	R1	O2		1·X, 1·Y			
e3	O2	Place SO	01-05 11:25	R1		A	3·X			
e4	O2	Place SO	02-05 11:55	R3		B	1·Y			
e5	O2	Create Invoice	03-05 16:15	R3	O2				I2	
e6	O2	Receive SO	04-05 10:00	R2		A		X1,X2,X3		
e7	O2	Update SO	04-05 10:25	R1	O2	B	2·Y			
e8	O2	Unpack	04-05 10:30	R2		A		X3		
e9	O2	Update Invoice	04-05 10:50	R2					I2	
e19	O2	Receive SO	07-05 10:10	R2		B		Y1,Y2		
e21	O2	Unpack	07-05 11:00	R2		B		Y2		
e29	O2	Receive Payment	09-05 08:30	R5						P1
e30	O2	Clear Invoice	09-05 08:45	R5					I1,I2	P1
e33	O2	Pack Shipment	09-05 11:45	R4	O2			X3,Y2		
e34	O2	Ship	09-05 15:00	R4	O2					

most process discovery techniques. As a result, also discovered process models are wrong. Figure 3 (left) shows the directly-follows graph (DFG) of the log in Table 2 and the corresponding process model discovered with the Inductive Miner (IM) annotated with the mean waiting times. Both models show false information suggesting that

– a *Supplier Order* was *Updated* after it was *Received* while this never happened;
– *rework* happened around receiving a *Supplier Order* and unpacking *Items* while each Supplier Order and each Item was touched only once;
– an *Invoice* can be *created* and *updated* in an arbitrary order while only one order was observed;

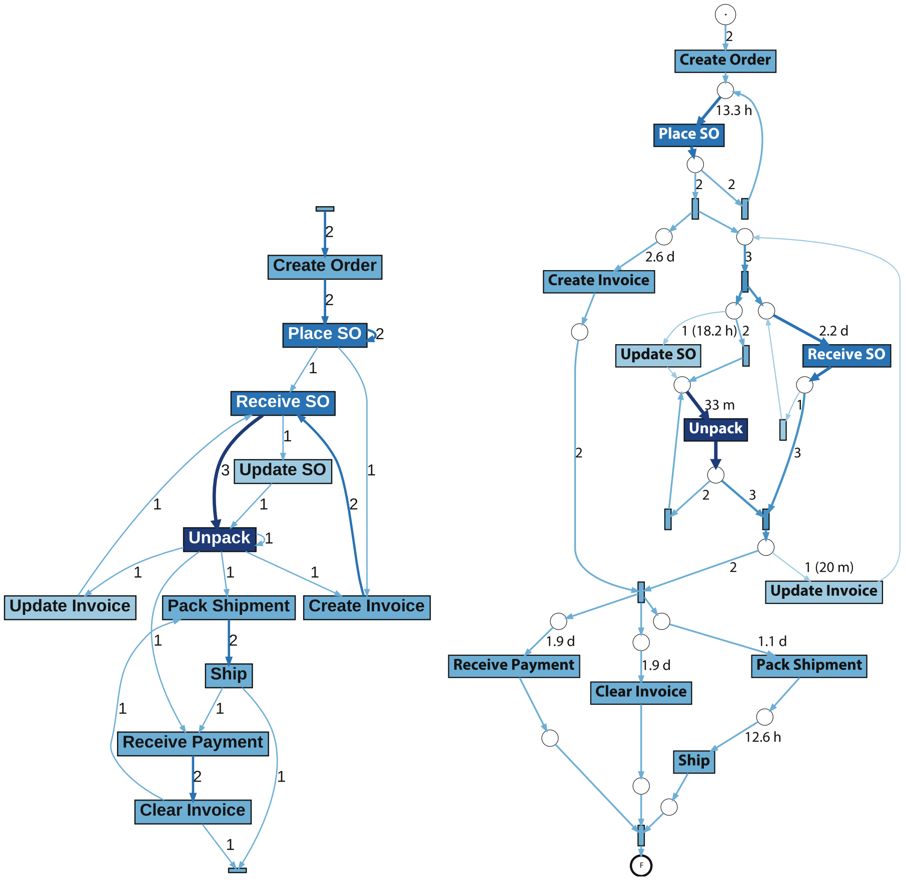

Fig. 3. Directly-follows graph of event log of Table 2 (left) and Inductive Miner model (right) show false dependencies.

The performance information in the IM model suggests that

– the mean time for receiving a *Supplier Order* after placement is 2.2d while A was received within 3d after placement (e_3-e_6) and B was received within 5d after placement (e_4-e_{19}) and within 3d after the last update (e_7-e_{19}).

This false behavioral information makes it impossible to properly locate deviating behaviors and causes for delays, e.g., the reasons why both orders were not shipped within 6 days.

3.3 Correct Behavioral Information: Local Directly-Follows

The reason why the event log in Table 2 contains false behavioral information is the following:

- Events that are (transitively) correlated to the global case identifier *Order* via a 1-to-m relationship are visible to multiple cases, and thus extracted multiple times, e.g. e_3.
- Extracting events from multiple different entities and ordering them by time from the perspective of the global case identifier *Order* constructs a temporal order between events that are actually unrelated, e.g., e_6 and e_7.

We can avoid both problems by simply *not* extracting all events towards a single case identifier, but keeping all events local to the entities they are *directly* correlated to. To analyze behavior, we only construct a temporal order between events that are related, e.g., correlated to the same entity.

In other words, instead of defining one global directly-follows relation for all events based on a global case identifier, we define a local directly-follows relation *per* entity [30, Def. 4.6].

Definition 6 (Directly-Follows (per Entity)). *Let* $T = (E, Attr, \#, ENT)$ *be an event table with entities. Let* $n \in Entities(ent, T)$ *be an entity of type* $ent \in ENT$.

Let $e_1, e_2 \in E$ *be two events;* e_2 directly follows e_1 from the perspective of n, *written* $e_1 \lessdot_{n,T} e_2$ *iff*

1. $(e_1, n), (e_2, n) \in corr_{ent,T}$ *(both are correlated to* n*),*
2. $e_1.time < e_2.time$ *(*e_1 *occurred before* e_2*),*
3. *and there is no other event* $(e', n) \in corr_{ent,T}$ *with* $e_1.time < e'.time < e_2.time$

For example, while e_7 directly follows e_6 globally for $O2$, they do not follow each other locally from the perspective of $O2$. Instead, from the perspective of $O2$, e_7 directly follows e_4, i.e., $e_4 \lessdot_{B,T} e_7$. Interestingly, also $e_2 \lessdot_{O2,T} e_7$ and $e_6 \lessdot_{R2,T} e_7$ hold. That means e_7 directly follows *three* different events as seen from three different perspectives: the Supplier Order B, the Order $O2$ and resource $R2$.

We cannot represent this information in a single table or a sequential event log. Extracting a *collection* of related sequential event logs from event tables [46] and relational databases [41] results in collection of directly-follows relations per entity-type. However, the behavioral information remains separated per entity type, hindering reasoning about the process as a whole [25]. We therefore turn to a graph-based data model.

4 Event Knowledge Graphs

Our primary aim is to model multiple local directly-follows relations (see Definition 6) over events correlated to multiple entities. To construct these relations, we also have to model entities, relations between entities, and correlations of entities to events (see Sect. 2). A *typed* graph data model such as *labeled property graphs* [48] allows to distinguish different types of nodes (events, entities) and relationships (directly-follows, correlated-to). We adopt labeled property graphs to construct a *knowledge graph* [33] of a process from event data, to augment this graph with further knowledge, and to even perform process mining analysis within a graph. Section 4.1 defines the generic data model of labeled property graphs which we use in Sect. 4.2 to define *event knowledge graphs* and "directly-follows" paths in an event knowledge graph. In Sect. 4.3 we discuss how to algorithmically construct an event knowledge graph from an event table.

4.1 Labeled Property Graphs

A labeled property graph is a graph where each node and each directed edge (called relationship) has a type, called *label*. Further, each node and each relationship can carry attribute-value pairs as properties. For the remainder, we fix a set λ_N of node labels, a set λ_R of relationship labels, and a set *Attr* of property names over a value domain *Val*.

Definition 7 (Labeled Property Graph). *A* labeled property graph *(LPG)* $G = (N, R, \lambda, \#)$ *is a graph with* nodes N, *and* relationships R *with the following properties:*

1. *Each node* $n \in N$ *carries a* label $\lambda(n) \in \Lambda_N$.
2. *Each relationship* $r \in R$ *carries a label* $\lambda(r) \in \Lambda_R$ *and defines a directed* edge $\overrightarrow{r} = (n_{source}, n_{target}) \in N \times N$ *between two nodes.*
3. *Any node* n *and relationship* r *can carry* properties *as attribute-value pairs via function* $\# : (N \cup R) \times Attr \nrightarrow Val$

We write $x.a = v$ for $\#(x, a) = v$ and $x.a = \perp$ if a is undefined for x. We write $N^\ell = \{n \in N \mid \lambda(n) = \ell\}$ and $R^\ell = \{r \in R \mid \lambda(r) = \ell\}$ for the nodes and relationships with label ℓ, respectively. We also write $(n_1, n_2) \in R^\ell$ if there exists $r \in R^\ell$ with $\overrightarrow{r} = (n_1, n_2)$.

Figure 4 shows an example of a labeled property graph, defining 5 nodes with label *Event*, 3 nodes with label *Entity*, 7 relationships with label *corr*, and 4 relationships with label *df*.

We here also provide some notation for standard operations on LPGs. Let $G_1 = (N_1, R_1, \lambda_1, \#^1)$ and $G_2 = (N_2, R_2, \lambda_2, \#^2)$ be two LPGs.

G_2 is a *sub-graph* of G_1, written $G_2 \subseteq G_1$, iff $N_2 \subseteq N_1, R_2 \subseteq R_1, \lambda_2 = \lambda_1|_{N_2 \cup R_2}, \#_2 = \#_1|_{N_2 \cup R_2}$. The *union* of G_1 and G_2 is $G_1 \cup G_2 = (N_1 \cup N_2, R_1 \cup R_2, \lambda_1 \cup \lambda_2, \#^1 \cup \#^2)$ under the assumption that $\lambda_1(x) = \lambda_2(x)$ and $\#^1_a(x) = \#^2_a(x)$ for all $a \in Attr$ for any $x \in (N_1 \cup R_1) \cap (N_2 \cup R_2)$. For a set $\mathbf{G} = \{G_1, \ldots, G_n\}$ of graphs, we write $\bigcup_{G \in \mathbf{G}} G = G_1 \cup \ldots \cup G_n$.

Labeled property graphs are a native data structure for knowledge graphs [33] and for a variety of *graph database systems* [48] that provide data management and query languages for reading and manipulating graphs [5].

4.2 Formal Definition of an Event Knowledge Graph

To precisely model event data in an LPG, we have to restrict ourselves to specific node labels for events and entities, and to specific relationship labels for correlation and directly-follows. Thereby, directly-follows relationships can only be

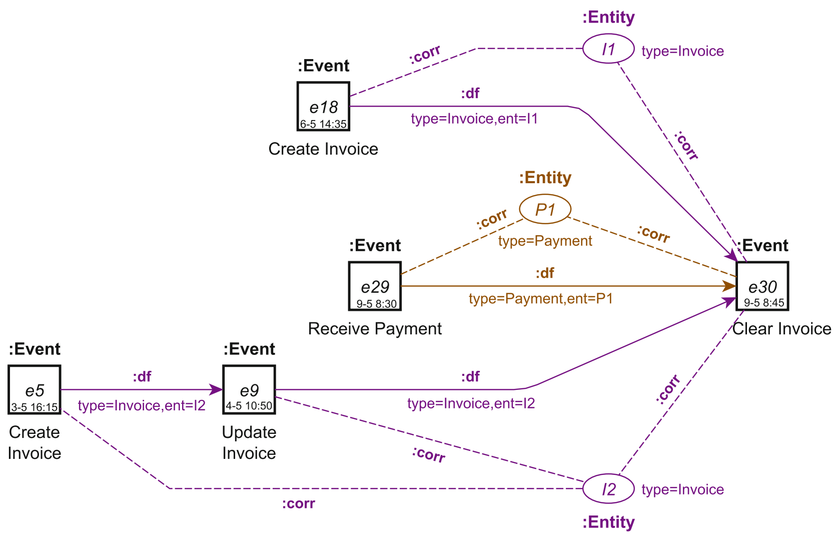

Fig. 4. Event knowledge graph of events $e_5, e_9, e_{18}, e_{29}, e_{30}$ of Table 2.

defined between events that are correlated to the same entity and directly follow each other from the viewpoint of that entity (Definition 6). This is formalized in the model proposed by Esser [25] which we here call *event knowledge graph*[4]

Definition 8 (Event Knowledge Graph). *An* event knowledge graph *(or just* graph*) is an LPG $G = (N, R, \lambda, \#)$ with node labels $\{Event, Entity\} \subseteq \Lambda_N$ and relationship labels $\{df, corr\} \subseteq \Lambda_R$ indicating "directly-follows" and "correlation" with the following properties.*

1. *Every event node $e \in N^{Event}$ records an activity name $e.act \neq \perp$ and a timestamp $e.time \neq \perp$.*
2. *Every entity node $n \in N^{Entity}$ has an entity type $n.type \neq \perp$.*
3. *Every correlation relationship $r \in R^{corr}$, $\overrightarrow{r} = (e, n)$ is defined from an event node to an entity node , $e \in N^{Event}, n \in N^{Entity}$; we write $n \in corr(e)$ and $e \in corr(n)$ as shorthand.*
4. *Any directly-follows relationship $df \in R^{df}$, $\overrightarrow{df} = (e_1, e_2)$ is defined between event nodes $e_1, e_2 \in N^{Event}$ and refers to a specific entity $df.ent = n \in N^{Entity}$ such that*
   (a) e_1 and e_2 are correlated to entity n: $(e_1, n), (e_2, n) \in R^{corr}$;
   (b) e_1 occurs before e_2: $e_1.time < e_2.time$; and
   (c) there is no other event $e' \in N^{Event}$ correlated to n, $(e', n) \in R^{corr}$ that occurs in between $e_1.time < e'.time < e_2.time$

[4] The initially chosen term "event graph" [25,38] which seems natural and shorter has previously been coined for a model for discrete event simulation [49]. At the same time, we will see that the proposed event *knowledge* graph model allows to capture more than just events.

We write $df.type = df.ent.type$ and $(e1, e2) \in R^{df}_n$.

Figure 4 shows an event knowledge graph for entities $I1, I2, P1$ of Table 2 and their correlated events. Each *df* relationship is defined between any two subsequent events correlated to the same entity. In the following, we omit the labels and use dashed edges for *corr* relationships, square nodes for *Event* nodes, and ellipses for *Entity* nodes.

A path along *df*-relationships corresponds to a trace in a classical event log. A *path* in a graph G is a sequence $\mathbf{r} = \langle r_1, \ldots, r_k \rangle \in R^*$ of consecutive relationships, i.e., the target node of $\overrightarrow{r_i} = (n_{i-1}, n_i)$ is the start node of $\overrightarrow{r_{i+1}} = (n_i, n_{i+1})$, $1 \leq i < k$.

Definition 9 (df-path). *Let $G = (N, R, \lambda, \#)$ be an graph.*

A path $\mathbf{r} = \langle r_1, \ldots, r_k \rangle \in (R^{df})^*$ of df-relationships is a directly-follows path (df-path) *iff all relationships are defined for the same entity, i.e., for all $1 \leq i < k$, $r_i.ent = r_{i+1}.ent = n$; we also say $\mathbf{r}$ is a df-path for entity n.*

$\mathbf{r}$ *is* maximal *iff there is no other df-relationship $r \in R^{df}$ so that $\langle r, r_1, \ldots, r_k \rangle$ or $\langle r_1, \ldots, r_k, r \rangle$ is also a df-path.*

For a path $\mathbf{r} = \langle r_1, \ldots, r_k \rangle \in (R^{df})^*, \overrightarrow{r_i} = (e_{i-1}, e_i)$ we write just the sequence of its nodes $\langle e_0, \ldots, e_k \rangle$ in case the correlated entity is clear. The graph in Fig. 4 defines three DF-paths: for $I1$: $\langle e_{18}, e_{30} \rangle$, for $I2$: $\langle e_5, e_9, e_{30} \rangle$, and for $P1$: $\langle e_{29}, e_{30} \rangle$.

Event knowledge graphs can be efficiently stored and queried using graph database systems [25]. This enables retrieving df-paths from graph databases using query languages, such as Cypher [25,33]. While the nodes and relationships of Definition 8 can also be encoded in RDF [11], the df-paths rely on attributes of relationships (Definition 9) which are not supported by RDF but by LPGs.

Alternative formalizations of Definition 8 define just a partial order over events [4,30,55,56] describing the local directly-follows relation wrt. various entities 6. Such a partial order view is equivalent to a family of df-paths [30, Cor. 4.9]. This equivalence allows to switch perspectives depending on the analysis task at hand.

4.3 Obtaining an Event Knowledge Graph from an Event Table

Event data is (currently) not recorded in the form of a graph, but for example in the form of an event table T with multiple entities (Definition 2). We obtain an event knowledge graph from an event table T in three steps.

1. Create an event node $e \in N^{Event}$ for each event record in the event table T.
2. *Infer entities and correlation relationships* from the event attributes: For each unique entity identifier found at some event e, create an entity node n and a *corr* relationship from e to n.
3. *Infer directly-follows relationships* between all events $e_1, \ldots, e_k$ with a *corr* relationship to the same entity node n.

We now explain and define each step along the running example of Table 1. We assume as input an event table $T = (E, Attr, \#^T, ENT)$ with multiple entities as stated in Definition 2. The central requirement is that each unique entity type $ent \in ENT \subseteq Attr$ is explicitly recorded as a dedicated attribute (column) of T, and that each value in column ent is an entity identifier.

Step 1: Create Event Nodes. We start by translating each event record in event table T into an event node in graph G.

Definition 10 (Event nodes from an event table). *Let $T = (E, Attr, \#^T, ENT)$ be an event table with entities. The* event nodes of T *are the graph $G_T^{Event} = (N^{Event}, \emptyset, \lambda, \#^G)$ with*

1. *$N^{Event} = E$, i.e., each event of T becomes an event node, and*
2. *$\#_a^G(e) = \#_a^T(e)$ for all $a \in Attr$, i.e., each event keeps all attributes from T as properties in G.*

The resulting graph G is a set of disconnected *Event* nodes only.

Step 2: Create Entity Nodes and Correlation Relationships. Each attribute of an event e in T that refers to an entity, e.g., $e.ent = \{n\}$, is now a property of the event node e in G. The basic idea is to "push out" this property: we make each unique value n an *Entity* node n and link e to n by a *corr* relationship. The following definition constructs a small graph $G^{corr}(n)$ that does exactly this. We then use graph union $G \cup \bigcup_n G^{corr}(n)$ to add them to G. The reason for doing so is that we can later calculate with various subgraphs.

Definition 11 (Entity and correlation inference). *Let $G = (N, R, \lambda, \#^G)$ be a graph and ENT be known entity types.*

Given a property name $ent \in ENT$, each property value $e.ent$ we find on an event node $e \in N^{Event}$ is an entity identifier of ent in G*: $Entities(ent, G) = \{n \mid \exists e \in N^{Event} : n \in e.ent\}$, see Definition 3.*

Let $n \in Entities(ent, G)$ be an identifier of type $ent \in ENT$. The entity and correlation inferred for n in G *is the graph $G^{corr}(n) = (N', R', \lambda'\#')$ with:*

1. *entity node $N'^{Entity} = \{n\}$ with $\#'_{type}(n) = ent$;*
2. *event nodes $N'^{Event} = \{e \in N^{Event} \mid n \in e.ent\}$ with $\#'(e) = \#(e)$ for each $e \in N'^{Event}$, i.e., each e is correlated to n, see Definition 4; and*
3. *correlation relationships $r_{e,n} \in R'^{corr}$, $\overrightarrow{r}_{e,n} = (e, n)$ iff $n \in e.ent$.*

We can infer entities and correlation on *any* event knowledge graph, not just the graph produced by Definition 10. This allows us to apply Definition 11 multiple times in any order. We can infer entities and correlation for an entity type ent by $G^{corr}(ent) = \bigcup_{n \in Entities(ent,G)} G^{corr}(n)$. We can add the inferred entities and correlation to graph G for all entity types ENT by graph union $G \cup \bigcup_{ent \in ENT} G^{corr}(ent)$. In the result, each value $n \in Entities(ent, T)$ becomes a new node n with $n.type = ent$. Correspondingly, each pair $(e, n) \in corr_{ent,T}$ becomes a new relationship of type *corr* from e to n.

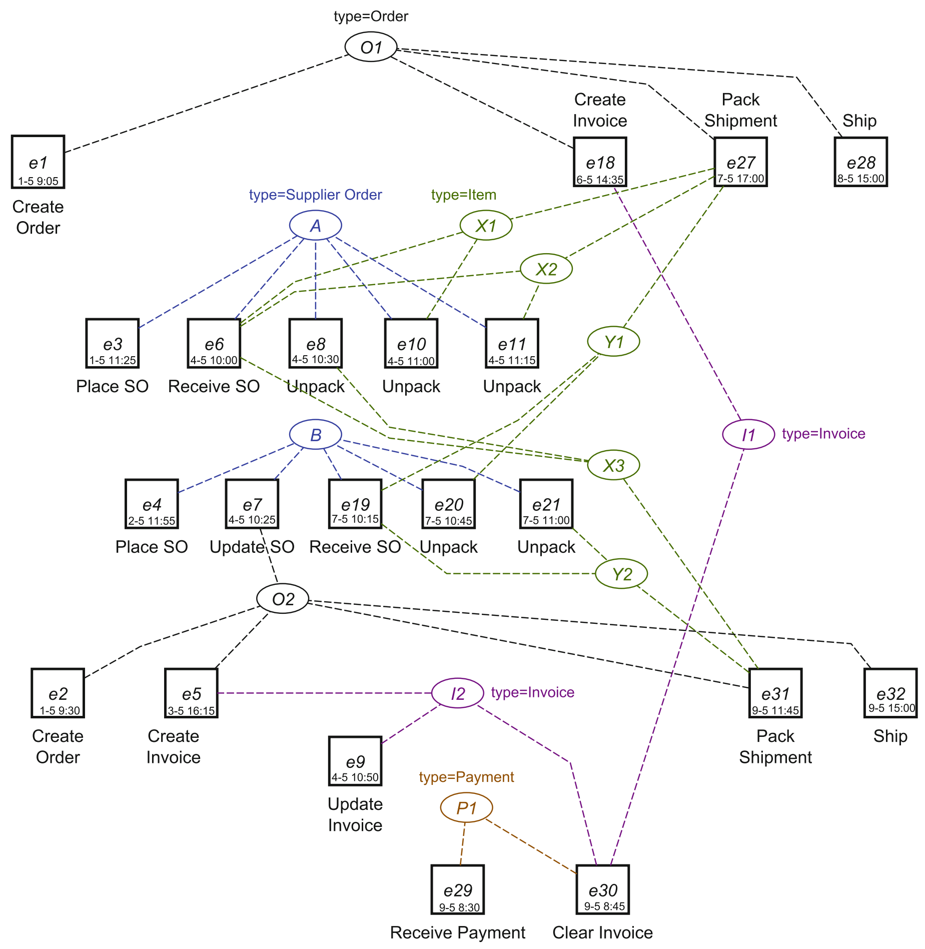

Fig. 5. Event graph of events of Table 2 without directly-follows relationships.

For example, applying Definition 10 on the event table of Table 2 results in the event nodes $e_1, \ldots, e_{11}, e_{18}, \ldots, e_{21}, e_{27}, \ldots, e_{32}$ shown in Fig. 5. Inferring entities and correlation for entity types *Order*, *Supplier Order*, *Item*, *Invoice*, and *Payment* adds the entity nodes and correlation edges shown in Fig. 5. In this graph we see that events $e_1, e_{18}, e_{27}, e_{28}$ are the events correlated to entity $O1$ of type *Order*. Moreover, event e_{18} is correlated to two entities *Order* $O1$ and *Invoice* $I1$; event e_{27} is correlated to four entities *Order* $O1$, *Item* $X1$, *Item* $X2$, and *Item* $Y1$.

Step 3: Infer Local Directly-Follows Relations. We now can infer the local directly-follows relation (Definition 6) and materialize it as *df*-relationships between event nodes. Again, the basic idea is simple: for each entity node n we retrieve all events $e_1, \ldots, e_n$ with a *corr*-relationship from e_i to n. We order

$e_1, \ldots, e_n$ by time and define a new *df*-relationship r from e_i to e_{i+1}; to remember for which entity r holds, we set $r.ent = n$.

As before, we do not add the *df*-relationships directly to G but construct a separate graph $G^{df}(n)$. We then add to G by graph union $G \cup \bigcup_n G^{df}(n)$ which later allows us to calculate with graphs.

Definition 12 (df inference). *Let $G = (N, R, \lambda, \#)$ be a graph. Let $n \in N^{Entity}$. Let $\langle e_0, \ldots, e_k \rangle$ be the sequence of events $\{e_0, \ldots, e_k\} = corr(n)$ correlated to n and sorted by time: $e_{i-1}.time < e_i.time, 1 \leq i \leq k$.*

The df-relationships inferred for *n in G is the graph $G^{df}(n) = (N'^{Event}, R'^{df}, \lambda', \#')$ with*

1. *event nodes $N'^{Event} = \{e_0, \ldots, e_k\}$, and*
2. *for each $1 \leq i \leq k$ one* df-*relationship $r_i \in R'^{df}$ with $\overrightarrow{r_i} = (e_{i-1}, e_i), \#'_{ent}(r_i) = n, \#'_{type}(r_i) = \#_{type}(n)$.*

We can only infer a df-relationship for entity n if $|corr(n)| > 1$. Thus, for df-inference to have any effect, we have to have inferred the entity n and correlation using Definition 11 and there are at least two events correlated to n. As for entity and correlation inference, we can add the inferred df-relationships to G by graph union $G \cup \bigcup_{n \in N^{Entity}} G^{df}(n)$.

For example, if we infer the df-relationships for each entity in the graph of Fig. 5 and add them to that graph, we obtain the graph shown in Fig. 6. Note that we only show the *corr* relationships to the first event of each entity for readability. This graph explicitly models the events, entities, correlation, and local directly-follows relations of all events in Table 2.

Complete Procedure. The following definition summarizes how to apply the above three definitions to obtain an event knowledge graph of an event table T.

Definition 13 (Event knowledge graph of an event table). *Let $T = (E, Attr, \#^T, ENT)$ be an event table with entities. The event table T defines the* graph $G = (N, R, \lambda, \#^G)$ of T *as follows:*

1. *Obtain the graph of event nodes G^{Event} of T (Definition 10).*
2. *Infer the entities and correlation for each entity type $ent \in ENT$ from G^{Event} (Definition 11), i.e., $G^{corr} = \bigcup_{ent \in ENT} G^{corr}(ent)$ which results in the intermediate graph $G^{Event} \cup G^{corr} = (N^{Event} \cup N^{Entity}, R^{corr}, \lambda, \#^G)$.*
3. *Infer the df-relationships $G^{df} = \bigcup_{n \in N^{Entity}} G^{df}(n)$ from $G^{Event} \cup G^{corr}$ (Definition 12) and return $G = G^{Event} \cup G^{corr} \cup G^{df}$.*

From Definition 10–13 follows that the df-relationships in graph G materialize the local directly-follows relation of event table T (Definition 6).

Lemma 1. *Let $G = (N, R, \lambda, \#^G)$ be the event knowledge graph of event table $T = (E, Attr, \#^T, ENT)$ with entities. For any entity $n \in Entities(ent, T)$, $ent \in ENT$ holds $e_1 \lessdot_{n,T} e_2$ (e_2 directly follows e_1 from the perspective of n) iff $(e_1, e_2) \in R^{df}_n$.*

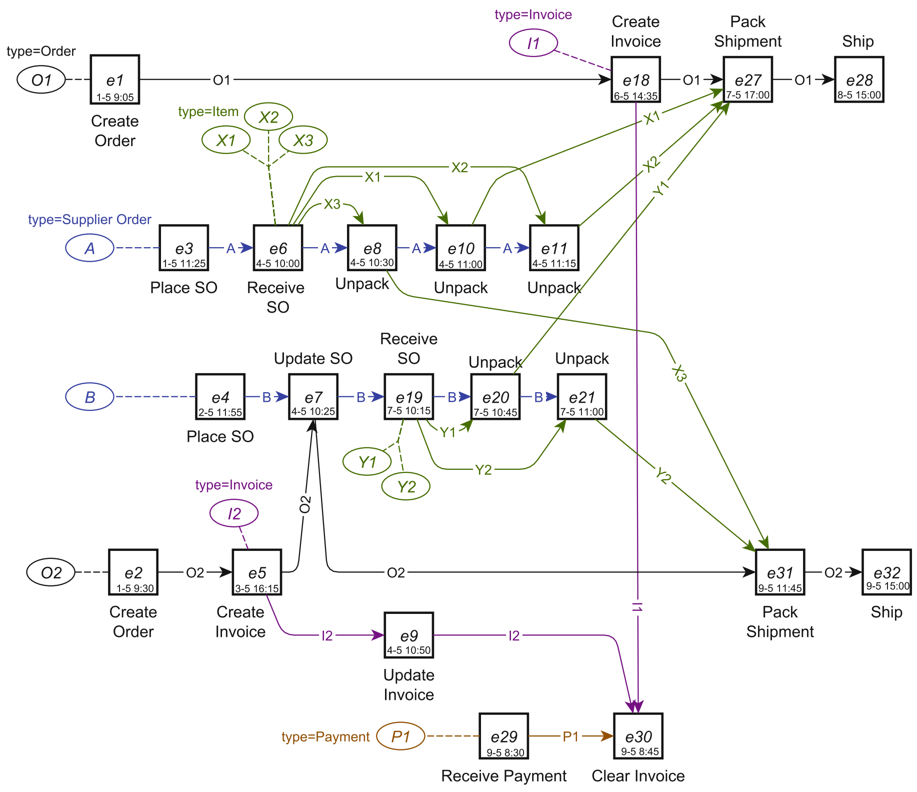

Fig. 6. Event graph of events of Table 2 after inferring directly-follows relationships.

4.4 Inferring Entity Interactions

The procedure of Definition 13 infers the local directly-follows relation for each entity in the graph. However, there are also important behavioral dependencies in the process *between* related entities, such as *Orders* and *Payments*, that are not visible in the graph of Fig. 6.

We know from Fig. 1 that shipping $O2$ has to wait until the invoice of $O1$ has been cleared by the related payment $P1$, but the graph of Fig. 6 suggests that e_{31} of $O2$ does not depend on e_{30} of $P1$ or any event of $O1$. This is because there is no entity correlated to both e_{31} and e_{30} or any event of $O1$.

Our analysis in Sect. 2.3 found that *Orders* are related to *Payments*. We can materialize this information in an event knowledge graph. We apply Definition 5 on all *Event* nodes to obtain relation $R_{(ent_1,ent_2)}$ between any two (interesting) entity types ent_1, ent_2. For each pair, $(n_1, n_2) \in R_{(ent_1,ent_2)}$ we add a new relationship with label *related* from entity node n_1 to entity node n_2. Figure 7 illustrates the result of this step for $(Order, Invoice)$ and $(Invoice, Payment)$. We can infer transitive relationships by materializing paths of *related*-relationships (ignoring their directions) as new *related*-relationships.

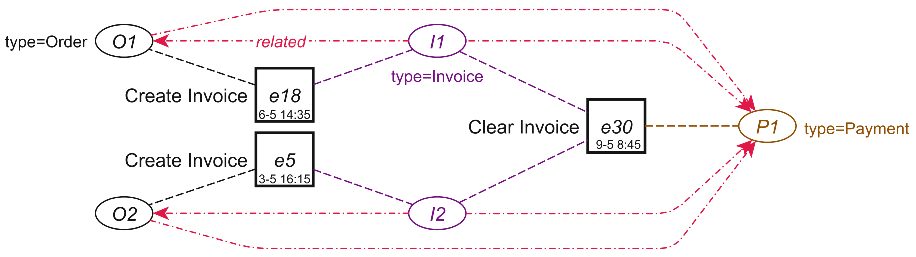

Fig. 7. Inferring relations between *Orders*, *Invoices*, and *Payments*.

For example, we materialize $\langle O1, I1, P1\rangle \in (R^{related})^*$ and $\langle O2, I2, P1\rangle \in (R^{related})^*$ as $(O1, P1), (O2, P1) \in R^{related}$ in Fig. 7. These steps obviously require domain knowledge to decide which potential relations to materialize, esp. when considering paths over n-to-1 and 1-to-n relationships [41].

We then can infer the behavior between two related entities by adapting entity and correlation inference (Definition 11) as follows [25]:

1. We *reify* the relation between two entity types ent_1 and ent_2 into a new *derived* entity type (ent_1, ent_2). That is, we make each pair $(n_1, n_2) \in R^{related}$ an entity node $(n_1, n_2) \in N^{Entity}$ with $(n_1, n_2).type = (ent_1, ent_2)$. For example, we create two entity nodes $(O1, P1), (O2, P1)$ of type *(Order,Payment)*. For traceability, we add a new relationship $d \in R^{derived}$ with label *derived* from entity (n_1, n_2) to n_1 and to n_2.
2. An event e is then correlated to a derived entity (n_1, n_2) iff e is correlated to n_1 or n_2 (or both). Formally, we add a new correlation relationship from e to (n_1, n_2) iff there is a correlation relationship $r \in R^{corr}$ from e to n_1 or n_2, i.e., $\overrightarrow{r} = (e, n_1)$ or $\overrightarrow{r} = (e, n_2)$.
3. Then we can treat any derived entity (n_1, n_2) just like any other entity and infer the df-relationships for (n_1, n_2), which results in a new path describing the interactions between n_1 and n_2.

Figure 8 shows the result of reifying the relation between *Order* and *Payment* entities of Fig. 7 into derived entities $(O1, P1)$ and $(O2, P1)$ of type $(Order, Payment)$ and inferring the df-relationships for this entity type. We now inferred df-paths from *Create Invoice* in $O1$ (e_{18}) via *Clear Invoice* in $P1$ (e_{30}) to *Pack Shipment* in $O2$ (e_{31}).[5]

Not all df-relationships for $(O1, P1)$ and for $(O2, P2)$ provide new information. For example in Fig. 8, $(e_2, e_5) \in R^{df}_{O2}$ and $(e_2, e_5) \in R^{df}_{(O2,P1)}$ run in parallel.

We say that a df-relationship $(e_1, e_2) \in R^{df}_{(n_1,n_2)}$ of a derived entity (n_1, n_2) *provides new information* if there is not already an existing df-relationship

[5] Our example here exploits that both orders of the same customer have invoices cleared by the same payment. For the more general case, we would have to include the customer in the data and infer the dependency via the customer entity.

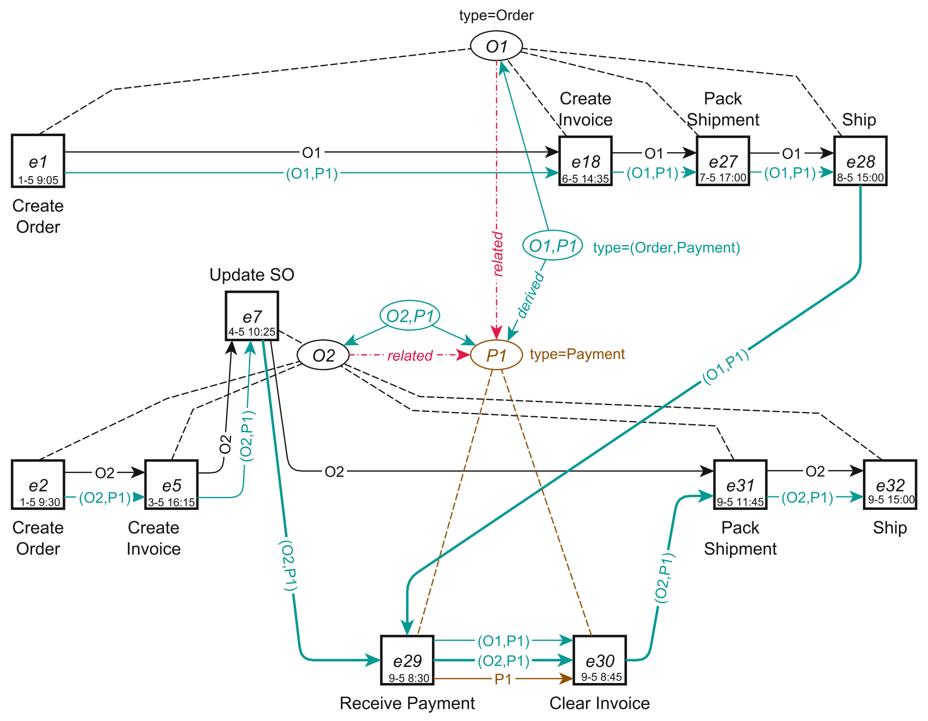

Fig. 8. Result of reifying the relation between *Order* and *Invoice* entities of Fig. 6 into a derived entity of type (*Order*, *Invoice*) and inferring the df-relationships for this entity type.

$(e_1, e_2) \in R^{df}_{n_1}$ or $(e_1, e_2) \in R^{df}_{n_2}$ for one of the original entities n_1 or n_2. Thus, a df-relationship (e_1, e_2) provides new information if it actually describes an interaction from n_1 to n_2 or vice versa. In Fig. 8, (e_7, e_{29}), (e_{28}, e_{29}), and (e_{30}, e_{31}) provide new information.

In principle we should keep only those *df*-relationships of a derived entity (n_1, n_2) that provide new information. However, we can best study the interaction between n_1 and n_2 when all *df*-relationships between n_1 and n_2 are part of a path related to (n_1, n_2). We therefore keep all *df*-relationships of (n_1, n_2) that either provide new information or are between two *df*-relationships of the *df*-path for (n_1, n_2) that do provide new information. In Fig. 8, for $(O2, P1)$, we keep (e_7, e_{29}) and (e_{30}, e_{31}) (provide new information) and also (e_{29}, e_{30}) (between df-relationships that provide new information); for $(O1, P1)$, we only keep (e_{28}, e_{29}).

The complete graph for Table 1 after inferring the *df*-relationships between *Order* and *Payment* entities is shown in Fig. 9.

4.5 Creating Event Knowledge Graphs from Real-Life Data

This method for constructing event knowledge graphs uses basic principles of information inference: (1) construct entities and correlation based on the presence of an entity identifier or a relation; and (2) derive a local directly-follows relation from the viewpoint of *each* entity. Our definitions assume the data to be accurate wrt. the real process, for instance, that entity identifiers and time stamps are recorded correctly and precise; otherwise further preprocessing is required [30,44,47].

All steps of the method can be implemented as a series of Cypher queries[6] to construct event knowledge graphs in a graph database for our running example [28] as well as for various real-life datasets comprising single and multiple event tables [24]; several event knowledge graphs of real-life processes are available [19–24]. A variant of event knowledge graphs, called *causal event graph* that only models events but not the entities, can be extracted automatically from relational databases [56].

In the following, we exploit the flexibility of LPGs that underly event knowledge graphs to infer and materialize further behavioral information, going beyond what event tables or event logs can describe.

5 Understanding Behavior over Multiple Entities

The event knowledge graph of Fig. 9 we obtain with the method of Sect. 4 explicitly models what we observed earlier in Sect. 1: the behavior of the different entities forms a complex *network* of synchronizing *df*-paths. This section first discusses how to interpret df-paths (Sect. 5.1) and how they synchronize (Sect. 5.2). We then discuss querying graphs through selection of entities and projection onto events in Sect. 5.3; we apply these operations to understand why the retailer of our example in Sect. 1 could not ship orders within the promised 6 days. We finally introduce aggregation in Sect. 5.4 which we use to discover basic process models directly within event knowledge graphs in Sect. 5.5.

5.1 How to Read Df-Paths in an Event Knowledge Graph

We discuss how to read *df*-paths over events based on running example of Fig. 6.

In a classical event log, each trace has a unique initial event and a unique final event indicating the start and completion of a process execution. A graph has multiple initial and final events – one per entity. Event e is *starting* or *ending* event if it has no incoming or outgoing *df*-relationship at all, e.g., $e_1, \dots, e_4$, and e_{32}. Event e is *starting* or *ending* event for entity n if it has no incoming or outgoing *df*-relationship for n. For example, e_{11} is the ending event of the *df*-path for A but it still has an outgoing *df*-relationship for $X2$. Some events are starting/ending events for *multiple* *df*-paths or entities. For example, e_6 is the

[6] https://github.com/multi-dimensional-process-mining/eventgraph_tutorial.

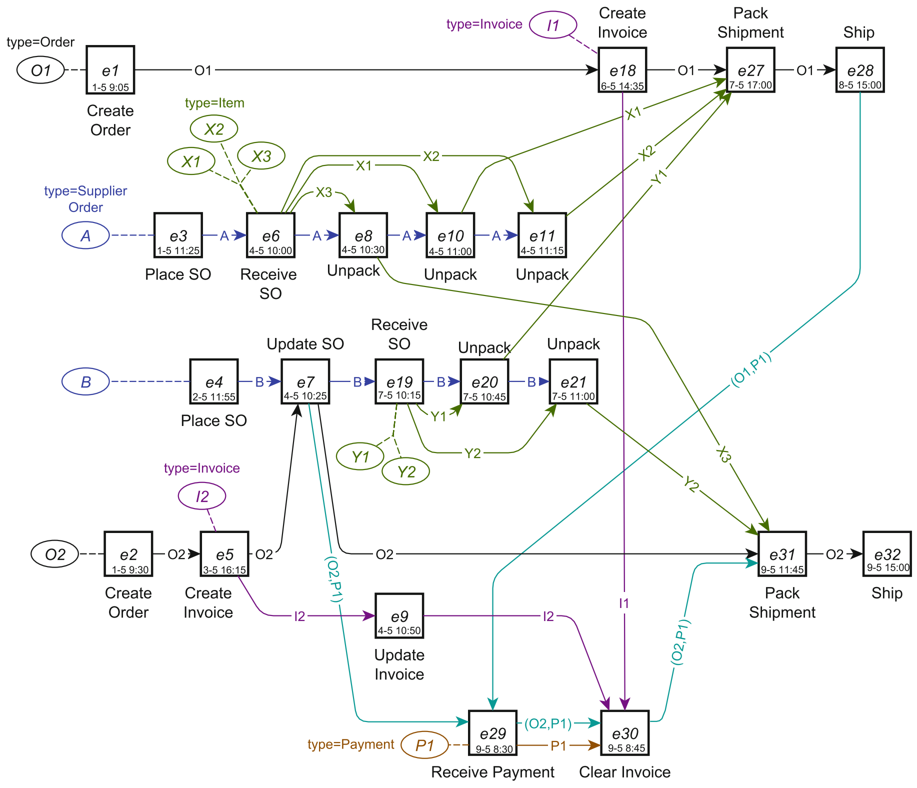

Fig. 9. Complete event knowledge graph of event table Table 1.

starting event for $X1, X2, X3$ and e_7 is the starting event for $Y1, Y2$ while e_{27} is the ending event for $X1, X2, Y1$ and e_{31} is the ending event for $X3, Y2$.

We call an event *intermediate* in a df-path of an entity n if it is not a starting or ending event in the df-path of n. For example, e_6 is an intermediate event of A.

In graph in Fig. 9 we see that the df-paths of entities of the same type are rather similar to each other.

- $O1$ and $O2$ both start with *Create Order* and end with *Ship* events with *Create Invoice* followed by *Pack Shipment* in between.
- A and B both start with *Place SO* (eventually) followed by *Receive SO*, ending with multiple *Unpack* events. Specifically
- Items $X1, \ldots, Y2$ start with *Receive SO* followed by *Unpack* and end with *Pack Shipment*
- $I1$ and $I2$ start with *Create Invoice* and end with *Clear Invoice*

Note that the graph no longer shows any directly-follows relation from *Receive SO* to *Update SO* that was falsely observed in Sect. 3. We can also analyze time differences between events on the df-path. For example, in Sect. 1 we stated that each Supplier Order is to be received within 3 days of placing the order.

- On the df-path of A, event e_6 (*Receive SO* for A, 4-5 10:15) is directly preceded by e_4 (*Place SO* for A, 1-5 11:25) which is within 3 days as required.
- On the df-path of B, event e_{19} (*Receive SO* for B, 7-5 10:15) is directly preceded by e_7 (*Update SO* for B, 4-5 10:25) which is within 3 days, but 6 days since e_4 (*Place SO* for B, 1-5 11:25). Thus, while the supplier delivered within the required 3 days since *Update SO*, the update itself introduced a 3-day delay.

Thus, the graph now shows temporal information and delays for individual entities correctly, in contrast to the classical event log of Sect. 3.

5.2 How to Read Synchronization in a Graph

Analyzing the df-paths for $O1$ and $O2$ also shows that none of the orders were shipped within 6 days: $e_{20}.time - e_1.time > 7 days$ and $e_{32}.time - e_2.time > 8 days$. As completing the orders depends on other entities, i.e., the items, we now analyze entity interactions through synchronization of df-paths.

A df-path $\mathbf{r} = \langle e_0, \ldots, e_k \rangle$ *goes through* an event e iff $e = e_i, 0 \leq i \leq k$. An event e is *local* to an entity n if there is only one df-path of entity n that goes through e, e.g., e_1, e_2, e_{32}. Two or more entities $n_1, \ldots, n_k$ *synchronize* in a *shared* event e if two or more df-paths of $n_1, \ldots, n_k$ go through e, e.g., e_7 synchronizes Supplier Order B and Order $O2$ whereas e_{19} synchronizes Supplier Order B and Items $Y1$ and $Y2$.

Reading Entity Creation and Updates. We now discuss different interpretations of entities $n_1, \ldots, n_k$ *synchronizing* in a shared event.

Event e *intermediately synchronizes* entities $n_1, \ldots, n_k$ when e is an intermediate event for $n_1, \ldots, n_k$. We can interpret an intermediate synchronization as an update or state change of one or more entities that requires the involvement of the other entities. For example, event e_7 intermediately synchronizes Order $O2$ and Supplier Order B to update B based on the information in $O2$; event e_8 updates both Supplier Order A and Item $X3$. Which entity changes state in e_8 is not visible in the graph of Fig. 9.

An event e that is intermediate for one entity n but a starting event for entities $n_1, \ldots, n_k$ can be interpreted as *entity n "created" or "initiated" entities $n_1, \ldots, n_k$*. For example, Supplier Order A created Items $X1, X2, X3$ in e_6, and Supplier Order B created $I2$ in e_5. Correspondingly, an event e that is intermediate for entity n and ending event for $n_1, \ldots, n_k$ is "closing" or "completing" entities $n_1, \ldots, n_k$. For example, Order $O1$ "completes" items $X1, X2, Y1$ in e_{27}.

An event e where multiple entities $n_1, \ldots, n_k$ of the same type synchronize is a *batching* event for $n_1, \ldots, n_k$ [36,42,55]. For example, e_{27} batches $X1, X2, Y1$, e_{30} batches $I1, I2$, and e_{31} batches $X3, Y2$.

However, we have to be careful with those interpretations as, both, the graph and the data from which it was created may be incomplete. Entities that are "created" or "closed" may continue to exist both prior and after the data recorded, e.g., all Items $X1, \ldots, Y2$ certainly exist prior to this process and after it, thus e_6 and e_{27} only show when these items entered the visibility or scope of our observations. Likewise, a starting event e for an entity n that is *not* an intermediate event for another entity n_2 does *not* describe how n was created. For example, $e_1, \ldots, e_4$ do not explain how $O1, O2, A, B$ were created. This is because our graph of Fig. 9 is incomplete as we did *not* (a) infer the *Resource* entity and the corresponding *df*-relationships from Table 1 and (b) we only recorded data in a limited time window. A helpful principle to check for incompleteness in distributed behavior is due to C.A. Petri [27]: most events happens due to a synchronous interaction of two or more entities, and most physical entities are never created from nothing and never disappear into nothing.

Reading Entity Interactions. Events and df-paths describe different modes of interaction. An event e where the df-paths of n_1 and n_2 synchronize is a *synchronous interaction.* A df-path for entity n describes an *asynchronous interaction* between n_1 and n_2 if n synchronizes both with n_1 and n_2 in different events. If the df-path for n has only 2 events $\langle e_1, e_2 \rangle$ then we can interpret entity n as *message* from n_1 to n_2. We can interpret an event e that is the ending event of entity n_1 and the starting event of entity n_2 as a *handover* from n_1 to n_2. In Fig. 9, e_7 is a synchronous interaction of $O2$ and B, the df-path of $Y1$ describes an asynchronous interaction from B to $O2$, and e_{28} is a handover from $O1$ to $(O1, P1)$.

If two entities n_1 and n_2 never synchronize in a shared event but there is at least one asynchronous interaction between n_1 and n_2, then n_1 and n_2 *interact asynchronously.* If all asynchronous interactions, i.e., df-paths, only go from n_1 to n_2, then the interaction is *one-directional*, and it is *bi-directional* otherwise. In Fig. 9, A and $O1$ interact asynchronously and one-directional (from A to $O1$ via $X1$), $O2$ and $P1$ interact asynchronously and bi-directional (via $(O2, P1)$).

n_1 and n_2 *interact indirectly* if for any two events e_1 of n_1 and e_2 of n_2 the shortest df-path from e_1 to e_2 involves df-relationships from multiple other entities. For example, $O1$ interacts indirectly with $O2$ via $(O1, P1)$ and $(O2, P2)$ (df-path $\langle e_{28}, e_{29}, e_{30}, e_{31} \rangle$).

Finally, n_1 and n_2 *do not interact* if there is no df-path from n_1 to n_2, or vice versa. For example, A and B do not interact. Note, however, that (indirect) interactions via other entities as well as non-interaction are subject to which entities have been included in the construction of the graph and which relations have been reified into derived entities.

Reading Event Dependencies and Delays. We observed in Sect. 5.1 that neither $O1$ nor $O2$ was shipped within 6 days as required in Sect. 1. We now want to analyze which entities, that synchronized with $O1$ and $O2$, delayed either order to be shipped on time.

Consider an event e that synchronizes the df-paths of multiple entities $n_1, \ldots, n_k$. Event e *directly depends on* any event e_i that directly precedes e via an incoming df-relationship $(e_i, e) \in R^{df}_{n_i}, 1 \leq i \leq k$ along entity n_i. We call $e.time - e_i.time$ the *delay* between e_i and e.

Suppose $e_1, \ldots, e_k$ are sorted on their delay to e. Event e_1 was the first event that directly preceded e, i.e., e could not have occurred earlier than e_1. The entity n_1, for which $(e_1, e) \in R^{df}_{n_1}$ was observed, was the first entity ready to synchronize in e. We can interpret that each later event $e_i, i > 1$ *delayed* the synchronization in e as entity n_i became ready to synchronize later than n_1 did, with e_k and n_k delaying e the most.

For example in Fig, 9, e_{31} (*Pack Shipment* for $O2$) depends on e_7, e_8, e_{21}, e_{30} along entities $O2$, $X3$, $Y2$, and $(O2, P1)$ with delays of 3 days, 3 days, 2 days, and 3 h, respectively. While $O2$ was first ready to synchronize in e_{31} after e_7 (*Update Order*); e_{31} was delayed most by e_{30} (*Clear Invoice* for $I1, I2$) along $(O2, P1)$.

For a given event e, we can build the set $delay^*(e)$ of transitive predecessors that delayed e the most, by first adding event e' that delayed e most, then adding event e'' that delayed e' most, etc. For example in Fig. 9, $delay^*(e_{32}) = \{e_{31}, e_{30}, e_{29}, e_{28}, e_{27}, e_{20}, e_{19}, e_7, e_5, e_2\}$.

Comprehending such subsets of events (and the dynamics they describe) is rather difficult. We use graph querying to reduce a graph to a subgraph of interesting events.

5.3 Basic Querying Operations

Similarly to classical event logs, we can also subset (or filter) event knowledge graphs for a more focused analysis. Recall that we have two basic operations to sub-setting classical event logs: selection (include only a subset of the cases with specific properties but keep all events in a case) and projection (keep all cases but keep only a subset of events with specific properties). The same operations can be applied on event knowledge graphs.

We *select* a subset of entities, but keep all event nodes correlated to the entities and all directly-follows relations between the events of these entities. Formally, given a graph G, we select entity nodes $N^{Entity}_{sel} \subseteq N^{Entity}$ from G by (1) removing all entity nodes $N^{Entity} \backslash N^{Entity}_{sel}$ and all adjacent *corr* relationships, then (2) removing all event nodes $e \in N^{Event}$ which no longer have any *corr* relationships (because none of their entities was selected) and the adjacent *df* relationships.

We *project* on a subset of events by keeping all entity nodes but only the selected event nodes; as this may interrupt df-paths (if an intermediate event gets removed) we have to recompute all df-relationships. Formally, given a graph G, we project onto event nodes $N^{Event}_{proj} \subseteq N^{Event}$ from G by (1) removing all *df*-relationships from G, (2) removing all event nodes $N^{Event} \setminus N^{Event}_{proj}$, and then (3) doing df-inference on the resulting graph (Definition 12).

The criteria by which we select events and entities can consider properties of events and entities but also relations to other event and entity nodes, and even

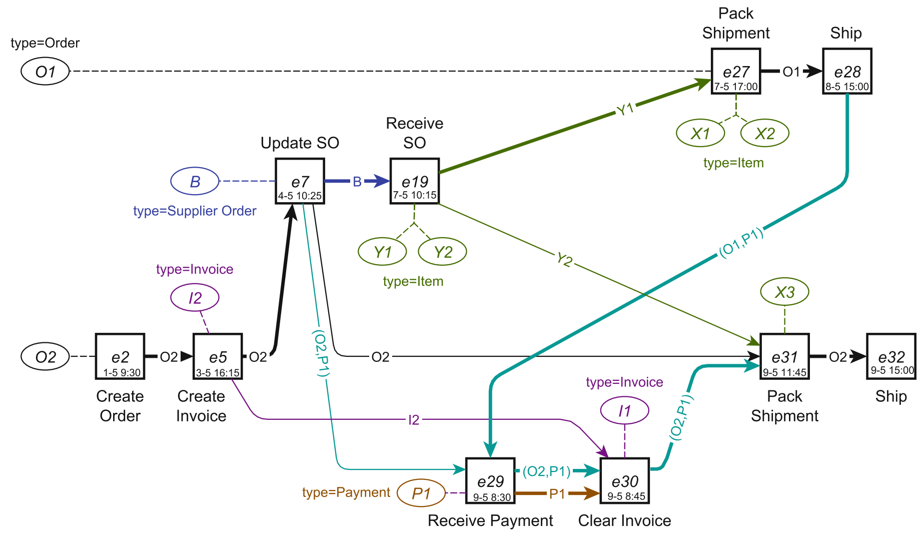

Fig. 10. Projection of Fig. 9 onto events that delayed most e_{28} and e_{32} and are not *Unpack* events. Bold df-relationships indicate which preceding event delayed an event the most.

more complex paths or sub-graphs. For example, to understand what caused delays in shipping order $O1$ (e_{28}) and $O2$ (e_{32}) while also removing unnecessary events, we can project the graph of Fig. 9 onto the events the (1) delayed either shipment the most (2) but without *Unpack* events. Formally, we project onto $(delay^*(e_{32}) \cup delay^*(e_{28})) \setminus \{e \in N^{Event} \mid e.act = Unpack\}$. Figure 10 shows the resulting graph. Note the new df-relationships $(e_5, e_{30}) \in R^{df}_{I2}$, $(e_{19}, e_{27}) \in R^{df}_{Y1}$, $(e_{19}, e_{31}) \in R^{df}_{Y2}$, obtained after doing df-inference over the remaining events.

In Fig. 10, we observe the following: *Pack Shipment* for $O1$ (e_{27}) was delayed by Item $Y1$ which was only ready for e_{27} after *Receive SO* (e_{19}). In turn, e_{19} was delayed by Supplier Order B with *Update SO* (e_7), which we already identified as cause for not receiving all items within 3 days in Sect. 5.1. *Pack Shipment* for $O2$ (e_{31}) was delayed by entity $(O2, P1)$, that means, by *Clear Invoice* (e_{30}) for the Payment $P1$ related to $O2$. *Receive Payment* for $P1$ (e_{29}) was delayed by $(O1, P1)$, that means, by *Ship* (e_{28}) for the related order $O1$.

Altogether, this allows us to pinpoint the bottlenecks in the process: *Update SO* delayed delivery of items $Y1, Y2$ needed for both $O1$ and $O2$, causing a delay in shipment for $O1$. The fact that the customer only paid and cleared both invoice $I1, I2$ after $O1$ was shipped delayed shipping $O2$ together with the retailer's policies.

5.4 Aggregating Events and Df-Relationships

Selection and projection allow to subset the data. Aggregation allows to materialize new nodes and relationships in the data. While the aggregation principle we explain here can be applied for many purposes, we specifically discuss it for

- aggregating sets of events into activities (or event classes), and
- aggregating df-relationships between events into corresponding relationships between activities.

The basic aggregation principle from sets of events to activities is formally identical to creating entity nodes from event properties as given in Definition 11.

- We select one event property that identifies a unique concept shared by many events, in this case the property *Activity*.
- For each value $c \in \{e.Activity \mid e \in N^{Event}\}$ of the *Activity* property that we find among the events in the graph, we create a new node c with label *Class* (representing the class of events with the same *Activity* property).
- We add an *observes* relationship from each event e to the *Class* node $c \in N^{Class}$ if $e.Activity = c$.
- We can also materialize how many events observe class $c \in N^{Class}$ in property $c.count$.

The yellow rounded rectangles in Fig. 11 represent the *Class* nodes of the events for Orders $O1$,$O2$ and Supplier Orders A,B. The dashed edges represent the *observes* relationship, e.g., e_2 and e_1 both observe *Create Order*.

We then can aggregate the df-relationships in a straight-forward way: for any two class nodes $c1$ and $c2$ we add a *df* relationship of type *ent* from c_1 to c_2 if there are corresponding events e_1 and e_2 that directly follow each other for *ent*, i.e., if $(e_1, c_1), (e_2, c_2) \in R^{observes}$ and $(e_1, e_2) \in R^{df}_n, n.type = ent$. We can also count how many df-relationships occur between events of c_1 and c_2 and add this as property to this relationship.

For example, in Fig. 11, we observe two df-relationships from *Create Order* to *Create Invoice* (e_1, e_{18}) and (e_2, e_5). Note, that this definition also creates self-loops around event classes, e.g., we observe three df-relationships from *Unpack* to *Unpack*. Also note that, as for events nodes, a class node can be part of *df*-relationships for multiple different entity types, e.g., *Update SO* is an activity that occurs for *Order* and *Supplier Order*.

5.5 Discovering Multi-entity Process Models

The aggregation operation of Sect. 5.4 essentially constructs a directly-follows graph. The key difference to the directly-follows graph of classical event logs is that each df-relationship between *Class* nodes is specific to one entity type. Thus, it respects the idea of the local directly-follows relation laid out in Definition 6. The resulting graph is a *multi-entity directly-follows graph*, also called *multi-viewpoint DFG* [4] or *artifact-centric model* [41].

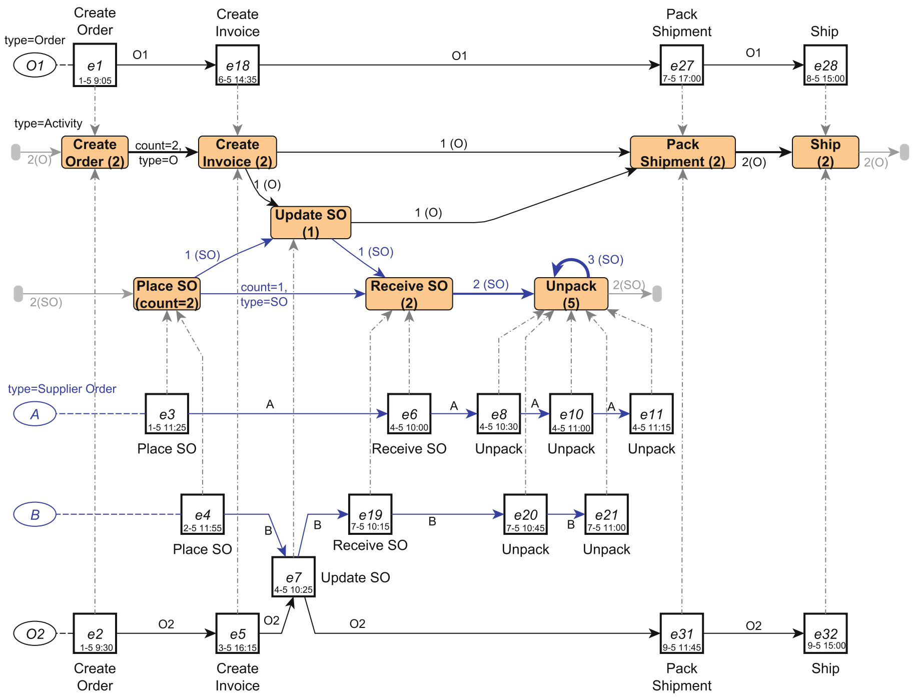

Fig. 11. Aggregating events to event classes and lifting the directly-follows relationships

Applying the event and df-aggregation of Sect. 5.4 to the graph of Fig. 9 results in the multi-entity DFG shown in Fig. 12. While the graph as a whole is rather complex, each edge is grounded in temporal relations of a specific entity type. Moreover, we can see that the behavior for each entity type is rather simple.

Event and df-aggregation can be implemented as simple, scalable queries[7] over standard graph databases, enabling efficient in-database process discovery [25,34]; the queries can be extended to filter based on frequencies or properties of the event knowledge graph [28].

An alternative representation of the multi-entity DFG is the proclet model [26] shown in Fig. 13. It is constructed by not creating a global *Class* node per unique *e.Activity* value in the data, but by creating a *Class* node per unique pair of activity name and entity type (*e.Activity*, *ent*). As a result, we see for example two *Create Invoice* nodes, one for *Order* and one for *Invoice*. Two class nodes of the same name are linked by a *cardinality* relationship that indicates how many entities are involved in an event of this class. For example, in every *Create Invoice* events, one *Order* and one *Invoice* is involved, while in every *Receive SO* event one *Supplier Order* and 2-3 *Items* are involved.

[7] https://github.com/multi-dimensional-process-mining/eventgraph_tutorial.

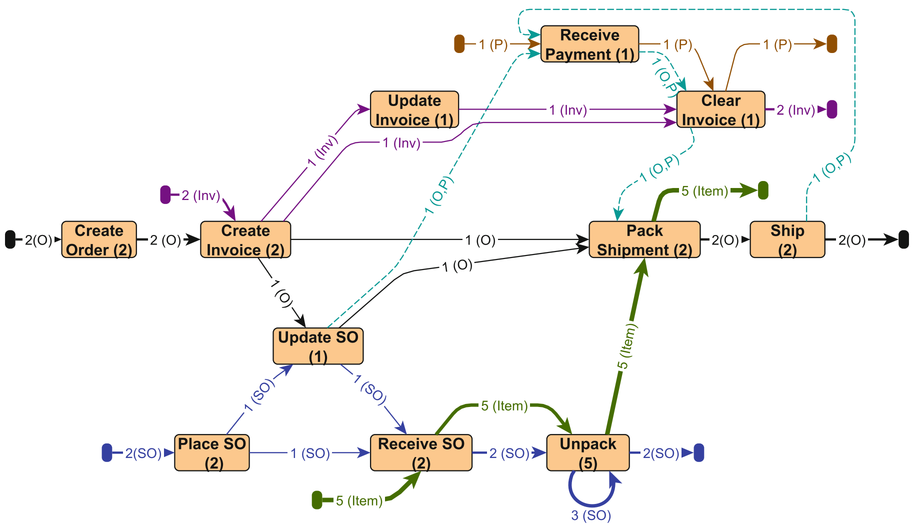

Fig. 12. Multi-Entity Directly-Follows-Graph of the running example obtained by aggregating the graph of Fig. 9

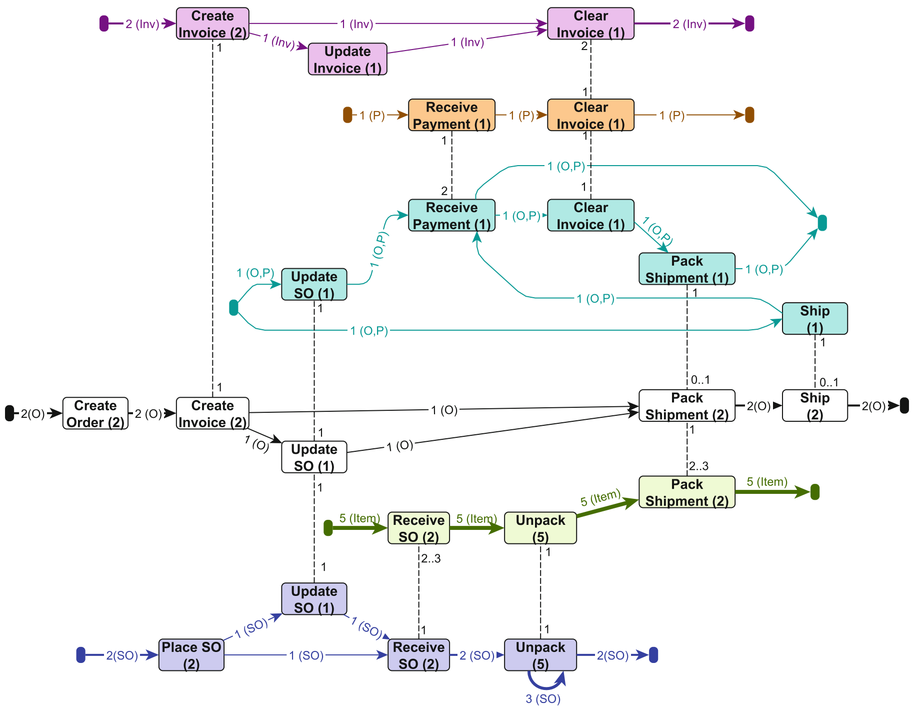

Fig. 13. Synchronous proclet model of the running example obtained by aggregating the graph of Fig. 9

6 Beyond Control-Flow: Multi-dimensional Process Analysis

So far, we analyzed the entities that are created and updated by the process based on the event data in Table 1. We now turn our attention to the *organizational entities* that actually make the process happen: the workers and supporting systems often called *resources*, and the work itself that is being carried out. Along the way, we showcase how flexible event knowledge graphs are. We integrate new events from a different data source in Sect. 6.1. We then enrich event knowledge graphs with df-paths over *activities* Sect. 6.2, which reveals *queues*. Enriching event knowledge graphs with df-paths over *workers* in Sect. 6.3 reveals patterns of how individual workers perform larger scale *tasks*. Finally, we show how to infer new information from (enriched) event knowledge graphs in Sect. 6.4.

6.1 Extending Event Knowledge Graphs with New Events

The process is supported by an automated warehouse (see Fig. 1). Figure 14 shows events of how the *Items* were handled by the warehouse. To analyze how the warehouse influenced the process, we have to combine these events with the events from Table 1. Luckily, we can avoid combining both tables into one joint event table and repeating the entire procedure of Sect. 4.3. We can simply *locally update* an existing graph with new events as follows. We choose to start from the graph of Fig. 6.

EventID	Activity	Time	Item
e12	Scan	04-05 13:00	X1
e13	Store	04-05 13:15	X1
e14	Scan	04-05 15:00	X2
e15	Store	04-05 15:15	X2
e16	Scan	04-05 17:00	X3
e17	Store	04-05 17:15	X3
e22	Retrieve	07-05 11:15	X1
e23	Retrieve	07-05 11:45	X2
e24	Scan	07-05 13:00	Y2
e25	Store	07-05 13:15	Y2
e26	Scan	07-05 15:00	Y1
e31	Retrieve	09-05 09:15	X3
e32	Retrieve	09-05 09:45	Y2

Fig. 14. Warehouse events

1. Import Fig. 14 into new event nodes (Definition 10). This results in new event nodes $e_{12}, \ldots, e_{17}, e_{22}, \ldots, e_{26}, e_{31}, e_{32}$.
2. Infer entities and correlation from the new event nodes (Definition 11). This results in the already existing entity nodes $X1, \ldots, Y2$.
3. For each entity node n inferred in step 2, remove every df-relationship $r \in R^{df}, r.ent = n$, and then infer the df-relationships for n (Definition 12) now including the new imported events.

The resulting graph is shown in Fig. 15. Note that we can obtained the original Fig. 6 again by selection of the original entities and projection onto the original events (see Sect. 5.3).

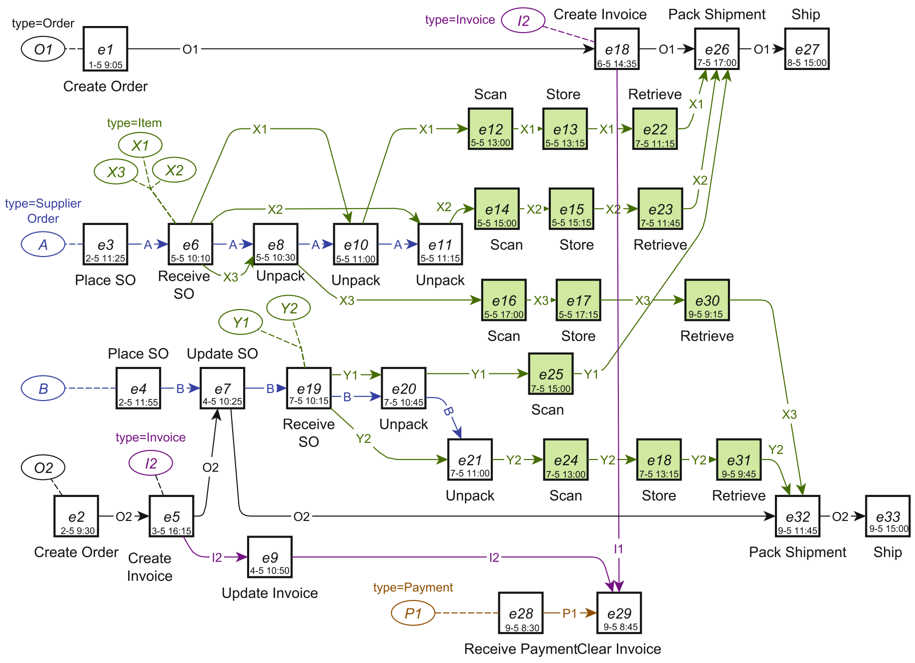

Fig. 15. Event graph after extending Fig. 6 with Fig. 14 (new events highlighted).

6.2 Adding Activities as Entities Reveals Queues

We defined entity inference in Definition 10 for the entity type attributes of the source event table. However, Definition 10 can be applied on *any* property of an event node.

For example, if we pick the *Activity* property as "entity identifier", we infer entities such as *Receive SO*, *Unpack*, *Scan*, *Store*, *Retrieve*, *Pack Shipment*. These are not entities handled by the process. No, these entities are the actual building blocks of the process. For example, each *Item* handled has to "pass through" each of these entities to be completely processed. We can visualize how other entities "pass through" activities by inferring in the graph of Fig. 15 the entity nodes for *Activity* and their df-paths[8]. Figure 16 shows the resulting graph (limited to a subset of events for readability).

We can see that the (red) *Activity* df-paths "go across" all the existing df-paths while the (green) *Item* df-paths traverse the different *Activity* df-paths largely "in parallel". Whenever an *Item* df-path synchronizes with an *Activity*

[8] Note that the *Entity* nodes identified by the activity property are *semantically different* from the *Class* nodes identified by the activity property that we obtained in Sect. 5.4. The *Class* nodes semantically aggregate the existing *df* relationships between events observed for other entities to *df* relationships between *Class* nodes. *Entity* nodes of the entity type *Activity* instead derive new df relationships in addition to existing df relationships for other entities.

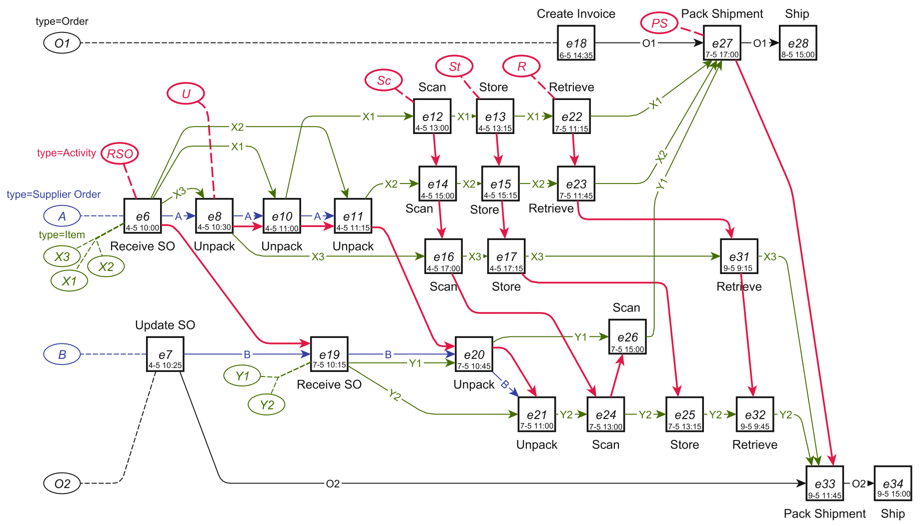

Fig. 16. Inferring *Activity* as entities in the graph of Fig. 15 reveals *Queues*. (Color figure online)

df-path in an event, the item is being worked on. Thus, we can interpret each *Activity* entity A as an abstract "work station" and its events as the work that is being performed there.

The space between two work stations A and B is a *queue* $A : B$, i.e., the space where *Items* after being worked on at A wait until being worked on at B. We can see in the graph in Fig. 16 that the *Items* do not always leave a queue in the same order they entered it: $X1$ entered *Unpack:Scan* after $X3$ (e_{10} follows e_8 in the df-path for *Unpack*) but leaves before $X3$ (e_{12} precedes e_{16} in the df-path for *Scan*).

We can better understand this behavior by changing the layout of the graph in Fig. 16. We select from Fig. 16 only *Item* and *Activity* entities. Setting the x-coordinate of each event by its time property and the y-coordinate by its *Activity* entity results in the graph in Fig. 17, which is called the *Performance Spectrum* [16].

The Performance Spectrum shows us that batching happens at *Receive SO* and *Pack Shipment* (diverging/converging *Item* df-paths), that *Scan:Store* and *Store:Retrieve* are being FIFO queues, that *Unpack:Scan* is *not* a FIFO queue, e.g., $X3$ is overtaken by $X1, X2$ and $Y1$ is overtaken by $Y2$.

We already identified in Sect. 5 reasons why Order $O2$ was not shipped within the 6 days promised by the retailed (see Sect. 1). We now can also clarify the reasons for $O1$. Figure 17 shows that although the second supplier order B with the required item $Y1$ was received on *7-5* (the 6th day of $O1$), order $O1$ was only packed *after* the *15:00* pick-up time. The non-FIFO handling in *Unpack:Scan* seems to be at fault. We observe

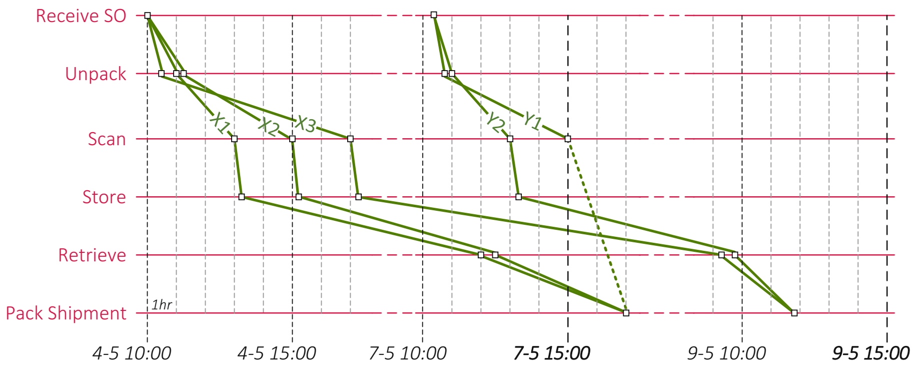

Fig. 17. Sub-graph of Fig. 16 for *Activity* entities and *Item* entities, with event coordinates defined by *Activity* (y-axis) and *time* (x-axis), results in the *Performance Spectrum*.

1. a consistent *2-h minimum waiting time* between two subsequent *Scan* activities (along its *Activity* df-path) causing $Y1$ to finish *Scan* after $Y2$ at *7-5 15:00*, and
2. a consistent *2-h minimum soujourn time* for the last *Item* reaching *Pack Shipment*, i.e., Pack Shipment for $X1, X2, Y1$ completes at *7-5 17:00.*

Thus, if *Unpack:Scan* had followed a strict FIFO policy, $Y1$ could have completed its *Scan* activity at *7-5 12:45*; the subsequent *Pack Shipment* event over $X1, X2, Y1$ could have completed at *7-5 14:45* just before the scheduled pick-up at *7-5 15:00.*

The Performance Spectrum reveals further, far more involved patterns of process performance over time than just batching and FIFO [16]. It is also implemented as a visual analytics tool over event data [15] and in combination with process models [54]. Mining performance patterns from it [36] allows to engineer so called inter-case features for improving the accuracy of remaining time prediction [37].

6.3 Adding Actors as Entities Reveals Complex Tasks

We found in Sect. 6.2 that *Activity* entities describe the abstract "work stations" where other entities are being worked on. Workers are performing this actual work. Often called "resources" in process management literature [18], we prefer the term *Actor* used in organizations research [32], as each actor follows its own behavior. To study actor behavior in the graph of Fig. 6, we only have to (1) infer the *Actor* entities from the event nodes (see Table 1), and (2) infer each actor's df-path. Figure 18 shows the resulting graph.

We can see actors $R1, R2, R3$ working "intertwined" in the same part of the process. In contrast, $R4$ and $R5$ work more separated from the other actors. Also, the actor df-paths actors show very different characteristics. The df-path $\langle e_1, e_2, e_3, e_7 \rangle$ of $R1$ synchronizes with any other entity only in one event, and

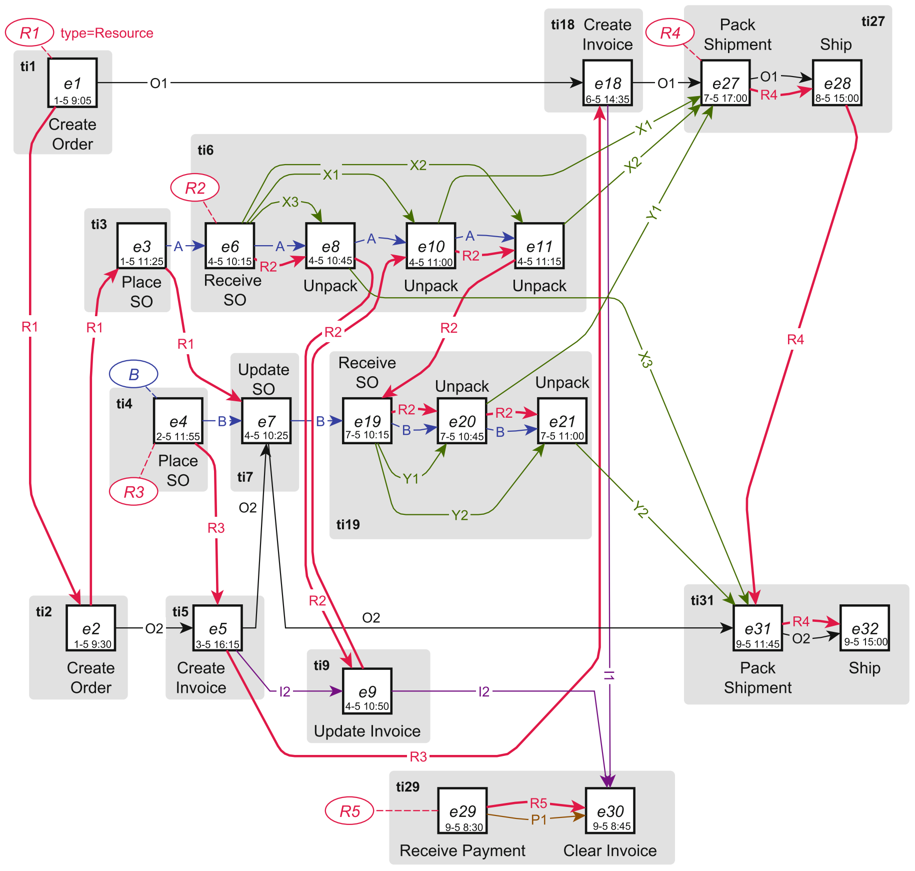
R1 type=Resource
ti1 e1 1-5 9:05 Create Order
ti18 Create Invoice e18 6-5 14:35
R4 Pack Shipment ti27 Ship e27 7-5 17:00 e28 8-5 15:00
ti6 R2 e6 4-5 10:15 Receive SO e8 4-5 10:45 Unpack e10 4-5 11:00 Unpack e11 4-5 11:15 Unpack
ti3 e3 1-5 11:25 Place SO
B ti4 e4 2-5 11:55 Place SO R3
Update SO e7 4-5 10:25 ti7
Receive SO Unpack Unpack e19 7-5 10:15 e20 7-5 10:45 e21 7-5 11:00 ti19
ti2 e2 1-5 9:30 Create Order
ti5 e5 3-5 16:15 Create Invoice
ti9 e9 4-5 10:50 Update Invoice
ti31 e31 9-5 11:45 Pack Shipment e32 9-5 15:00 Ship
ti29 R5 e29 9-5 8:30 Receive Payment e30 9-5 8:45 Clear Invoice
O1 O2 A B X1 X2 X3 Y1 Y2 R1 R2 R3 R4 R5 I1 I2 P1

Fig. 18. Adding actors (*Resource*) entities to the event knowledge graph reveals task execution patterns

then moves on to the next entity $O1$, $O2$, A, B, always performing just a single activity on each. In contrast, the df-path of $R4$ synchronizes over multiple subsequent events with the same entity, i.e., e_{27}, e_{28} in $O1$ and e_{31}, e_{32} in $O2$, meaning $R4$ always performs a "unit of work" that consists of two subsequent activities. Such a larger unit of work of multiple related activities is called a *task* [32,38].

A *task instance* of an actor R working on an entity X materializes in an event knowledge graph as a specific subgraph over event nodes $e_1, \ldots, e_k$: (1) the df-paths of R and X both meet in e_1, (2) diverge in e_k, (3) synchronize in each event node $e_1, \ldots, e_k$, and (4) at least one of their df-paths has no other event in between $e_1, \ldots, e_k$ [38]. The grey rectangles highlighted in Fig. 18 shows several task instance. The task instances themselves and the way they are ordered in the graph reveal unique characteristics of performing work.

- Actor $R1$ only performs a series of singleton tasks ti_1, ti_2, ti_3, ti_4. The df-path of $R1$ describes that *Supplier Order A* has been placed only after both *Orders* $O1$ and $O2$ were created.
- Actor $R4$ performs two instances ti_{27} and ti_{31} of the same task (first *Pack Shipment* then *Ship*) directly after each other on two different *Orders*.
- Actor $R2$ also performs two instances ti_6 and ti_{19} of the same task (first *Receive* then repeatedly *Unpack*) directly after each other; however $R2$ interrupts ti_6 on A to perform ti_9 (*Update Invoice*) on $I2$.

Further, more complex types of task instances can be identified in event knowledge graphs [38]. The df-relationships between task instances also reveal patterns of how work is handed over between actors. For example $R1$ hands work over to $R2$ in all *Supplier Orders*, to $R3$ in all *Orders*, and to $R4$ in $O2$; $R2$ hands work over to $R4$ in all *Items* and to $R5$ in $I2$. Such patterns are studied in the area of routines research [32].

We clearly can see some undesirable behavior in how actors collaborate over the different entities.

- $R1$ created both *Orders* $O1$ and $O2$ but only placed *Supplier Order A*. Instead, $R3$ placed B and we cannot observe a handover from $R1$ to $R3$; this lack of collaboration may have led to $R3$ placing a wrongly *Supplier Order B* (with only one item Y). The *Update SO* by $R1$ remedies the problem but caused to the delay in delivering $Y1$ we identified in Sect. 5. The problem may have been avoided by $R1$ completing the "larger task" $\langle e_1, \ldots, e_4 \rangle$ alone.
- $R2$ is *interrupting* their work on unpacking *Supplier Order A* after $X3$ (e_8) to *Update Invoice* $I2$ before continuing on unpacking $X1$ (e_{10}). This "context switch" between handling *Items* and *Invoice* results in a longer delay between two subsequent *Unpack* events (30mins) than usual (15mins), which we can directly see in the Performance Spectrum in Fig. 17. The longer delay is a risk to packing shipments on time as we analyzed in Sect. 6.2. The risk could be reduced by ensuring that $R2$ is not interrupting their task; $R3$ could have updated the invoice instead.

The process model shown in Fig. 21, and further explained in Sect. 7, describes for each actor behavioral routines that could avoid undesirable behavior.

6.4 Inference in Event Knowledge Graphs with Multiple Layers

Our discussions so far focused on constructing, understanding, and finding patterns in graphs over *Entity* and *Event* nodes and the *df* and *corr* relationships. As the model of event knowledge graphs (Definition 8) is based on labeled property graphs (Definition 7), we can extend an event knowledge graph with further node and relationship types, to describe more knowledge about the process. We already did that in Sect. 5.4 when aggregating multiple *Event* nodes of the same activity to a new node with label *Class*. In the following we expand on this idea by an example. We do so in the style of a process mining analyst applying all the concepts of the previous sections as data processing operations. In fact all steps shown here can be realized through Cypher queries over a graph database.

Suppose we want to create a concise summary of how actors organize the work of handling *Supplier Orders*, based on the graph with actor df-paths shown in Fig. 18. The actors correlated to *Supplier Order* events are $R1, R2, R3$. We select entities A and B and $R1, R2, R3$ and then project onto events of a *Supplier Order* or between two *Supplier Order* events (to keep e_9). The resulting graph is shown in Fig. 19 as "Event Entity Layer".

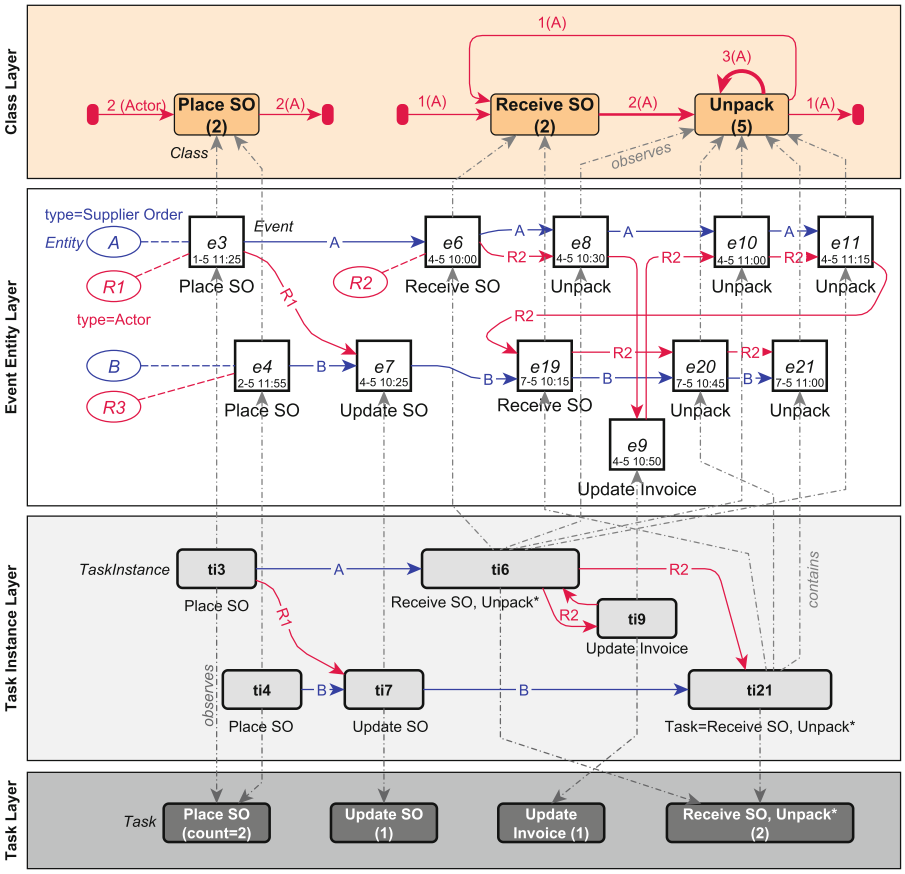

Fig. 19. An event knowledge graph extended with additional layers into a "process knowledge graph".

Next, we aggregate the event layer into a new "Task Instance Layer".

1. For each task instances, i.e., each subgraph of an *Actor* df-path and an *Supplier Order* df-path synchronizing on consecutive events as defined in Sect. 6.3,

we extend the graph with a new node with label *TaskInstance*, resulting in the nodes $ti_3, ti_4, ti_6, ti_7, ti_9, ti_{21}$ shown in the “Task Instance Layer” of Fig. 19.

2. We add a new *contains* relationship $(ti, e) \in R^{contains}$ from each *TaskInstance* node ti to each *Event* node e that is part of the task instance, e.g., $(ti_9, e_{19}), (ti_9, e_{20}), (ti_9, e_{21}) \in R^{contains}$ in Fig. 19. This connects the nodes in both layers.
3. Each $ti \in N^{TaskInstance}$ gets the property $ti.Task$ by concatenating the *Activity* values of the event nodes it contains along their df-path (abstracting repetitions with a Kleene star), e.g., $ti_6.Task = \langle Receive\ SO, Unpack^* \rangle$.
4. We then lift the df-relationships from *Event* nodes to *TaskInstance* nodes. For each df-relationship $(e, e') \in R^{df}$ between events e, e' contained in different task instances $ti \neq ti', (ti, e), (ti', e') \in R^{contains}$, we create a new df-relationship $(ti, ti') \in R^{df}$ between task instance nodes ti and ti' (and copy the properties of the df relationship).

The resulting “Task Instance Layer” in Fig. 19 represents the “Event Entity Layer” at the aggregation level of task executions instead of activity executions.

5. To understand which tasks are performed and how often, we aggregate *TaskInstance* nodes into *Task* nodes by their *Task* property (see Sect. 5.4).

The resulting “Task Layer” in Fig. 19 shows four tasks *Place SO* (performed twice in ti_3, ti_4), *Update SO* (performed once in ti_7), *Update Invoice* (performed once in ti_9), and $\langle Receive\ SO, Unpack^* \rangle$ (performed twice in ti_6, ti_{21}).

We now want to visualize the behavior all actors regarding the *frequent tasks* in handling *Supplier Orders*, e.g., tasks performed at least twice. The visualization shall be on the abstraction level of the activities performed by actors, i.e., a multi-entity DFG. To achieve this, we aggregate the “Event Entity Layer” into a “Class Layer” using the “Task Layer” as context.

1. Select from the “Event Entity Layer” only the *Actor* entities; this removes all df-relationships for A and B.
2. Project onto all event nodes $e \in N^{Event}$ having a path $\langle e, ti, t \rangle$ to a task node $t \in N^{Task}$ with $t.count \geq 2$, i.e., only events that are contained in a task instance ti of a frequently occurring task. This removes e_7 and e_9 from the graph in Fig. 19 and introduces $(e_8, e_{10}) \in R^{df}_{R2}$.
3. Aggregate the *Event* nodes to *Class* nodes by their *Activity* property and lift df-relationships from *Event* nodes to *Class* nodes (see Sect. 5.4).

The resulting multi-entity DFG forms a new “Class Layer” in the graph, that is connected to the “Event Entity Layer” by *observes* relationships, as shown in Fig. 19. The multi-entity DFG shows that $R1$ and $R2$ work on disjoint sets of activities, and that $R2$ indeed follows a cyclic, structured behavior. The paths from *Class* nodes *Receive SO* and *Unpack* to the *Task* nodes show that all activities belong to the same task, i.e., one cycle is one “unit of work”.

The multi-entity DFG is a filtered DFG: it lacks df-relationships for *Supplier Orders* and it omits *Update SO*. Thus, the multi-entity DFG *does not fit* or *deviates* from the “Event Entity Layer”. We can identify the deviations in multi-layered process knowledge graph in Fig. 19 similar to alignments [9]; see [8]. For instance, for df-relationship (*Unpack*, *Unpack*) $\in R^{df}$ in the "Class Layer", we see

- two corresponding "synchronous" df-relationship $(e_{10}, e_{11}), (e_{20}, e_{21}) \in R^{df}_{R2}$ with $(e_i, \textit{Unpack}) \in R^{observes}$; and
- one corresponding "log-move" df-path $\langle e_8, e_9, e_{10} \rangle \in (R^{df}_{R2})^*$ with $(e_8, \textit{Unpackt}), (e_9, \textit{Update Invoice}), (e_{10}, \textit{Unpack}) \in R^{observes}$, i.e., e_9 occurs in between *Unpack* and *Unpack*.

7 Conclusion and Outlook

The preceding sections studied different forms of process mining over multiple behavioral dimensions that are summarized in Fig. 20. We showed in Sect. 3 how classical process mining techniques fail when the assumption of a single entity handled by a single execution (bottom left quadrant in Fig. 20) is violated.

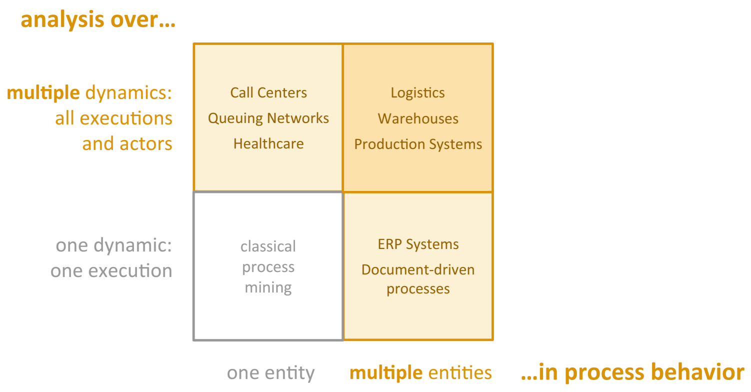

Fig. 20. Quadrants of process analysis over multiple behavioral dimensions

To overcome these assumptions, we introduced process mining with event knowledge graphs, that rests on three simple, but fundamental principles:

1. Explicitly represent every entity that an event is correlated to as a node. An entity thereby can be anything: a specific object, a person or actor, or even an abstract concept such as an activity.
2. Infer directly-follows relations over events per entity. This results in directly-follows paths forming complex, but meaningful structures that can be filtered for.
3. Aggregate any structure of interest formed by directly-follows paths into new nodes describing process-related concepts, explicitly linked to the structures that generate them. This allows to infer interactions between related entities (see Sect. 4.4), multi-entity process models (see Sect. 5.5), and task instances (see Sect. 6.3).

Applying these principles, we constructed event knowledge graphs from standard event data through simple concepts in Sect. 4.3. We showed in Sect. 5 how to analyze processes where each execution involves *multiple related entities*, such as ERP systems and document-driven processes (bottom right quadrant in Fig. 20). We showed in Sect. 6 how event knowledge graphs also allow to analyze *multiple dynamics* together. We added actor and queue behavior to study how entities pass through queues or actors perform tasks across multiple entities, which are dynamics studied in call centers or in healthcare (top left quadrant in Fig. 20). Note that, in Sect. 6 we always focused on a single entity processed in a queue or in a task. How to analyze the combination of *multiple dynamics* over *multiple entities* (top right quadrant in Fig. 20) is an open question.

Event knowledge graphs give rise to a number of novel research questions.

We have shown how to construct event knowledge graphs from event tables, even automatically [24,25]. We also need techniques to construct event knowledge graphs from relational database while preserving the existing entities and relations. Existing automated conversion techniques from relational to graph databases [50] only convert records into entity nodes, while event knowledge graphs require to construct event nodes.

The quality of a process mining analysis on event knowledge graphs relies on having identified the relevant structural relations (between entities) and behavioral or cause-effect relations (between events) (see Sect. 4.4). We need automated techniques to infer relevant relations that take the temporal semantics of the df-relationship into account. Promising first steps are techniques that explicitly allow to incorporate domain knowledge when inferring causal relationships from relational data [56], or use ontologies [6,7] for extraction. Specifically, dynamically changing relationships and changes of object properties [39,40] still need to be considered.

We have sketched the possibility of structuring a complex process mining analysis by adding analysis layers to the graph, but limited ourselves to simple selection, projection, and aggregation queries. Adequate query languages that also can handle process-relevant phenomena such as frequency, noise, performance in relation to multiple entities need to be considered. Also, more complex behavioral dynamics can be discovered. For example, enriching the event knowledge graph with the activity dimension to derive the performance spectrum (see Sect. 6.2) allows detecting subgraphs that indicate high workload (many events in a short interval) or a dynamic bottleneck (a short-term increase in waiting time) [51]. Aggregating these to "high-level events" and mining for cause-effect relations among them reveals how performance anomalies cascade through a process [51].

Finally, while we did discuss how to discover multi-entity directly-follows graphs through aggregation, true process discovery of models with precise semantics from event knowledge graphs still has to be addressed. In principle, such models can be discovered through principles of artifact-centric process mining [41,46]: First obtain a classical event log per entity type, e.g., by extracting the df-paths per entity type from the graph, and discover a classical process

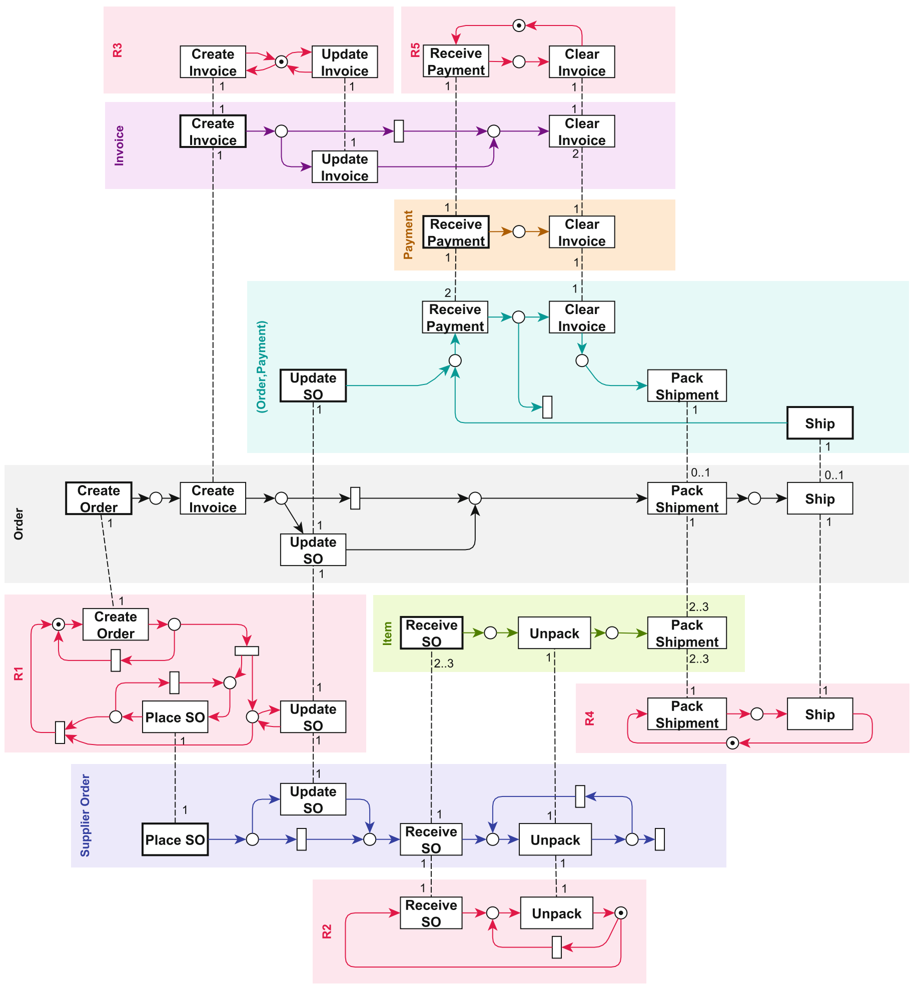

Fig. 21. Synchronous proclet model for the graph of Fig. 9 extended with proclets describing the *intended* (not the observed) behavior for all actors.

model per entity type. Then compose the models of the different entity types to express their synchronization.

Figure 21 shows a possible process model that could be obtained in this way for our example, using a multi-entity extension for Petri nets, called *synchronous proclets* [26]. Each proclet is a Petri net that describes the behavior of one entity type; bold-bordered initial transitions describe the creation of a new entity. The dashed *synchronization edges* describe which transitions occur together; the multiplicity annotations indicate how many entities of each type have to be involved. Note that the proclet model in Fig. 21 is a hybrid between discovered and manually created model. The proclets for *Order*, *Supplier Order*, *Invoice*, *Item*, *Payment*, and *(Order,Payment)* are each discovered from the entity type's df-paths of the graph in Fig. 9. The proclets for the *Actors* however are created manually[9], describing the intended routine for each actor based on the insights in Sect. 6.3. Bold-bordered initial transitions describe the creation of a new entity; note that the proclets for actors do not have an initial transition but an initial marking as actors are not created in the process. Dashed *synchronization edges* between transitions describe that the transitions have to occur together; the multiplicity annotations indicate how many entities of each type have to be involved. For instance, $R1$ creates 1 new *Order* in each occurrence of *Create Order*, but $R4$ always packs 2–3 *Items* into 1 *Shipment* in each occurrence of *Pack Shipment*.

An alternative formalization of this concept are *object-centric Petri nets* [53]. Object-centric Petri nets also first discover one Petri net per entity type, then annotate the places and arcs with entity identifiers, and then compose all entity nets along transitions for the same activity, resulting in a coloured Petri net model that is accessible for analysis [53] and measuring model quality [3]. However, synchronization by composition prevents explicitly modeling (and thus discovering) interactions between entities such as the relation from *Order* to *Payment* described by proclet *(Order,Payment)* in Fig. 21.

Though, while proclets can describe entity interactions, the behavior of entity interactions tends to be rather unstructured resulting in overly complex models [41]. Extensions of declarative models (see [10]) such as modular DCR graphs [14], that apply similar principles as synchronous proclets, could be more suitable. Alternatively, scenario-based models [31] that specify conditional partial orders of events over multiple entities could be applied. For instance, the conditional scenario in Fig. 22 specifies the interaction between *Orders* and *Payments* observed in the graph of Fig. 9.

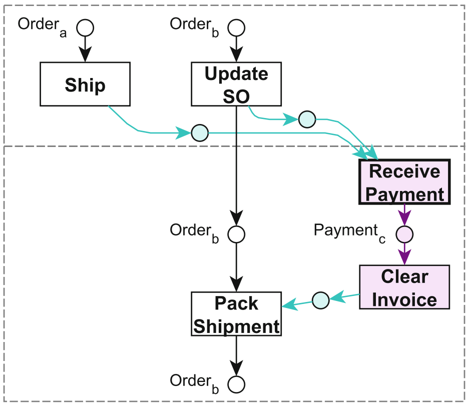

Fig. 22. Conditional scenario describing an interaction of 2 *Orders* and 1 *Payment*.

Altogether, event knowledge graphs give rise to entirely novel forms of process mining that support novel forms of process management [17].

[9] We created one proclet per actor as introducing a proclet for all actors would result in a very complex proclet as different actors follow very different behavior. Further, the manually created model conveniently avoids the issue of having to layout how $R2$ synchronizes both with *Supplier Order* and with *Invoice*.

References

1. van der Aalst, W.M.P.: Process mining: a 360 degrees overview. In: van der Aalst, W.M.P., Carmona, J. (eds.) Process Mining Handbook. LNBIP, vol. 448, pp. 3–34. Springer, Cham (2022)
2. Accorsi, R., Lebherz, J.: A practitioner's view on process mining adoption, event log engineering and data challenges. In: van der Aalst, W.M.P., Carmona, J. (eds.) Process Mining Handbook. LNBIP, vol. 448, pp. 212–240. Springer, Cham (2022)
3. Adams, J.N., van der Aalst, W.M.P.: Precision and fitness in object-centric process mining. In: ICPM 2021, pp. 128–135. IEEE (2021)
4. Berti, A., van der Aalst, W.: Extracting multiple viewpoint models from relational databases. In: Ceravolo, P., van Keulen, M., Gómez-López, M.T. (eds.) SIMPDA 2018-2019. LNBIP, vol. 379, pp. 24–51. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-46633-6_2
5. Bonifati, A., Fletcher, G.H.L., Voigt, H., Yakovets, N.: Querying Graphs. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, San Rafael (2018)
6. Calvanese, D., Kalayci, T.E., Montali, M., Santoso, A.: OBDA for log extraction in process mining. In: Ianni, G., et al. (eds.) Reasoning Web 2017. LNCS, vol. 10370, pp. 292–345. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61033-7_9
7. Calvanese, D., Kalayci, T.E., Montali, M., Tinella, S.: Ontology-based data access for extracting event logs from legacy data: the onprom tool and methodology. In: Abramowicz, W. (ed.) BIS 2017. LNBIP, vol. 288, pp. 220–236. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59336-4_16
8. Carmona, J., van Dongen, B., Weidlich, M.: Conformance checking: foundations, milestones and challenges. In: van der Aalst, W.M.P., Carmona, J. (eds.) Process Mining Handbook. LNBIP, vol. 448, pp. 155–190. Springer, Cham (2022)
9. Carmona, J., van Dongen, B.F., Solti, A., Weidlich, M.: Conformance Checking - Relating Processes and Models. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99414-7
10. Di Ciccio, C., Montali, M.: Declarative process specifications: reasoning, discovery, monitoring. In: van der Aalst, W.M.P., Carmona, J. (eds.) Process Mining Handbook. LNBIP, vol. 448, pp. 108–152. Springer, Cham (2022)
11. Cyganiak, R., Hyland-Wood, D., Lanthaler, M.: RDF 1.1 concepts and abstract syntax. W3C Proposed Recommendation (2014)
12. de Murillas, E.G.L., Reijers, H.A., van der Aalst, W.M.P.: Case notion discovery and recommendation: automated event log building on databases. Knowl. Inf. Syst. **62**(7), 2539–2575 (2019). https://doi.org/10.1007/s10115-019-01430-6
13. De Weerdt, J., Wynn, M.T.: Foundations of process event data. In: van der Aalst, W.M.P., Carmona, J. (eds.) Process Mining Handbook. LNBIP, vol. 448, pp. 193–211. Springer, Cham (2022)
14. Debois, S., López, H.A., Slaats, T., Andaloussi, A.A., Hildebrandt, T.T.: Chain of events: modular process models for the law. In: Dongol, B., Troubitsyna, E. (eds.) IFM 2020. LNCS, vol. 12546, pp. 368–386. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-63461-2_20
15. Denisov, V., Belkina, E., Fahland, D., van der Aalst, W.M.P.: The performance spectrum miner: visual analytics for fine-grained performance analysis of processes. In: BPM 2018 Demos. CEUR Workshop Proceedings, vol. 2196, pp. 96–100. CEUR-WS.org (2018)

16. Denisov, V., Fahland, D., van der Aalst, W.M.P.: Unbiased, fine-grained description of processes performance from event data. In: Weske, M., Montali, M., Weber, I., vom Brocke, J. (eds.) BPM 2018. LNCS, vol. 11080, pp. 139–157. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98648-7_9
17. Dumas, M., et al.: Augmented business process management systems: a research manifesto. CoRR, abs/2201.12855 (2022)
18. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management, 2nd edn. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-662-56509-4
19. Esser, S., Fahland, D.: Event Graph of BPI Challenge 2014. Dataset. https://doi.org/10.4121/14169494
20. Esser, S., Fahland, D.: Event Graph of BPI Challenge 2015. Dataset. https://doi.org/10.4121/14169569
21. Esser, S., Fahland, D.: Event Graph of BPI Challenge 2016. Dataset. https://doi.org/10.4121/14164220
22. Esser, S., Fahland, D.: Event Graph of BPI Challenge 2017. Dataset. https://doi.org/10.4121/14169584
23. Esser, S., Fahland, D.: Event Graph of BPI Challenge 2019. Dataset. https://doi.org/10.4121/14169614
24. Esser, S., Fahland, D.: Event Data and Queries for Multi-Dimensional Event Data in the Neo4j Graph Database, April 2021. https://doi.org/10.5281/zenodo.4708117
25. 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/s13740-021-0122-1
26. Fahland, D.: Describing behavior of processes with many-to-many interactions. In: Donatelli, S., Haar, S. (eds.) PETRI NETS 2019. LNCS, vol. 11522, pp. 3–24. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21571-2_1
27. Fahland, D.: Petri's understanding of nets. In: Reisig, W., Rozenberg, G. (eds.) Carl Adam Petri: Ideas, Personality, Impact, pp. 31–36. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-96154-5_5
28. Fahland, D.: multi-dimensional-process-mining/eventgraph_tutorial, April 2022
29. Fahland, D., de Leoni, M., van Dongen, B.F., van der Aalst, W.M.P.: Behavioral conformance of artifact-centric process models. In: Abramowicz, W. (ed.) BIS 2011. LNBIP, vol. 87, pp. 37–49. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21863-7_4
30. Fahland, D., Denisov, V., van der Aalst, W.M.P.: Inferring unobserved events in systems with shared resources and queues. Fundam. Informaticae **183**(3–4), 203–242 (2021). https://doi.org/10.3233/FI-2021-2087
31. Fahland, D., Prüfer, R.: Data and abstraction for scenario-based modeling with petri nets. In: Haddad, S., Pomello, L. (eds.) PETRI NETS 2012. LNCS, vol. 7347, pp. 168–187. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31131-4_10
32. Goh, K., Pentland, B.: From actions to paths to patterning: toward a dynamic theory of patterning in routines. Acad. Manag. Ann. **62**, 1901–1929 (2019)
33. Hogan, A., et al.: Knowledge graphs. ACM Comput. Surv. **54**(4) (2021). https://doi.org/10.1145/3447772
34. Jalali, A.: Graph-based process mining. In: Leemans, S., Leopold, H. (eds.) ICPM 2020. LNBIP, vol. 406, pp. 273–285. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72693-5_21
35. Jans, M., Soffer, P.: From relational database to event log: decisions with quality impact. In: Teniente, E., Weidlich, M. (eds.) BPM 2017. LNBIP, vol. 308, pp. 588–599. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74030-0_46

36. Klijn, E.L., Fahland, D.: Performance mining for batch processing using the performance spectrum. In: Di Francescomarino, C., Dijkman, R., Zdun, U. (eds.) BPM 2019. LNBIP, vol. 362, pp. 172–185. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-37453-2_15
37. Klijn, E.L., Fahland, D.: Identifying and reducing errors in remaining time prediction due to inter-case dynamics. In: ICPM 2020, pp. 25–32. IEEE (2020). https://doi.org/10.1109/ICPM49681.2020.00015
38. Klijn, E.L., Mannhardt, F., Fahland, D.: Classifying and detecting task executions and routines in processes using event graphs. In: Polyvyanyy, A., Wynn, M.T., Van Looy, A., Reichert, M. (eds.) BPM 2021. LNBIP, vol. 427, pp. 212–229. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85440-9_13
39. Li, G., de Carvalho, R.M., van der Aalst, W.M.P.: Automatic discovery of object-centric behavioral constraint models. In: Abramowicz, W. (ed.) BIS 2017. LNBIP, vol. 288, pp. 43–58. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59336-4_4
40. Li, G., de Murillas, E.G.L., de Carvalho, R.M., van der Aalst, W.M.P.: Extracting object-centric event logs to support process mining on databases. In: Mendling, J., Mouratidis, H. (eds.) CAiSE 2018. LNBIP, vol. 317, pp. 182–199. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92901-9_16
41. Xixi, L., Nagelkerke, M., van de Wiel, D., Fahland, D.: Discovering interacting artifacts from ERP systems. IEEE Trans. Serv. Comput. **8**(6), 861–873 (2015)
42. Martin, N., Pufahl, L., Mannhardt, F.: Detection of batch activities from event logs. Inf. Syst. **95**, 101642 (2021)
43. Nooijen, E.H.J., van Dongen, B.F., Fahland, D.: Automatic discovery of data-centric and artifact-centric processes. In: La Rosa, M., Soffer, P. (eds.) BPM 2012. LNBIP, vol. 132, pp. 316–327. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36285-9_36
44. Pegoraro, M., Bakullari, B., Uysal, M.S., van der Aalst, W.M.P.: Probability estimation of uncertain process trace realizations. In: Munoz-Gama, J., Lu, X. (eds.) ICPM 2021. LNBIP, vol. 433, pp. 21–33. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-98581-3_2
45. Piessens, D.A.M.: Event log extraction from SAP ECC 6.0. Master's thesis, Eindhoven University of Technology (2011)
46. Popova, V., Fahland, D., Dumas, M.: Artifact lifecycle discovery. Int. J. Cooperative Inf. Syst. **24**(1), 1550001:1–1550001:44 (2015). https://doi.org/10.1142/S021884301550001X
47. Pourmirza, S., Dijkman, R.M., Grefen, P.: Correlation miner: mining business process models and event correlations without case identifiers. Int. J. Cooperative Inf. Syst. **26**(2):1742002:1–1742002:32 (2017)
48. Robinson, I., Webber, J., Eifrem, E.: Graph Databases. O'Reilly Media, Sebastopol (2013)
49. Schruben, L.: Simulation modeling with event graphs. Commun. ACM **26**(11), 957–963 (1983)
50. Stoica, R., Fletcher, G.H.L., Sequeda, J.F.: On directly mapping relational databases to property graphs. In: 13th Alberto Mendelzon International Workshop on Foundations of Data Management. CEUR Workshop Proceedings, vol. 2369. CEUR-WS.org (2019)
51. Toosinezhad, Z., Fahland, D., Köroglu, Ö., van der Aalst, W.M.P.: Detecting system-level behavior leading to dynamic bottlenecks. In: ICPM 2020, pp. 17–24. IEEE (2020). https://doi.org/10.1109/ICPM49681.2020.00014

52. Aalst, W.M.P.: Object-centric process mining: dealing with divergence and convergence in event data. In: Ölveczky, P.C., Salaün, G. (eds.) SEFM 2019. LNCS, vol. 11724, pp. 3–25. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30446-1_1
53. van der Wil, M.P.: Aalst and Alessandro Berti. Discovering object-centric petri nets. Fundam. Informaticae **175**(1–4), 1–40 (2020)
54. van der Aalst, W.M.P., Tacke Genannt Unterberg, D., Denisov, V., Fahland, D.: Visualizing token flows using interactive performance spectra. In: Janicki, R., Sidorova, N., Chatain, T. (eds.) PETRI NETS 2020. LNCS, vol. 12152, pp. 369–380. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51831-8_18
55. Waibel, P., Novak, C., Bala, S., Revoredo, K., Mendling, J.: Analysis of business process batching using causal event models. In: Leemans, S., Leopold, H. (eds.) ICPM 2020. LNBIP, vol. 406, pp. 17–29. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72693-5_2
56. Waibel, P., Pfahlsberger, L., Revoredo, K., Mendling, J.: Causal process mining from relational databases with domain knowledge (2022). https://doi.org/10.48550/ARXIV.2202.08314