The Relational Process Structure

Sebastian Steinau, Kevin Andrews, and Manfred Reichert

Institute of Databases and Information Systems, Ulm University, Germany
{sebastian.steinau,kevin.andrews,manfred.reichert}@uni-ulm.de

Abstract. Using data-centric process paradigms, small processes such as artifacts, object lifecycles, or Proclets have become an alternative to large, monolithic models. In these paradigms, a business process arises from the interactions between small processes. However, many-to-many relationships may exist between different process types, requiring careful consideration to ensure that the interactions between processes can be purposefully coordinated. Although several concepts exist for modeling interrelated processes, a concept that considers both many-to-many relationships and cardinality constraints is missing. Furthermore, existing concepts focus on design-time, neglecting the complexity introduced by many-to-many relationships when enacting extensive process structures at run-time. The knowledge which process instances are related to which other process instances is essential. This paper proposes the relational process structure, a concept providing full support for many-to-many-relationships and cardinality constraints at both design- and run-time. The relational process structure represents a cornerstone to the proper coordination of interrelated processes.

Keywords: Process interactions, data-centric processes, many-to-many relationships, relational process structure

1 Introduction

With the emergence of data-centric process support paradigms as, for example, the object-aware [4] or artifact-centric [8] approaches, the focus has shifted away from large, monolithic process models. Instead, the use of small processes showing limited complexity is preferred. An example of such a small process may be the lifecycle process of an artifact or object. With the advent of microservices, which may be used to implement processes, and the vision that devices in the Internet of Things become capable of running their own processes, small process models are an enticing way of specifying business processes. In general, each of these small processes does not constitute a business process by itself. Instead, to reach a specific business goal, these processes need to interact and collaborate, i.e., small processes are not executed in isolation, but are interdependent. Therefore, a coordination mechanism is needed to properly manage these process interactions. It is paramount for a coordination mechanism to be aware of every process relationship at design-time and, especially, at run-time.

Existing approaches for the interaction-focused modeling of business processes, e.g., Proclets [11], have investigated the multiplicity of process relationships in the context of *one-to-many relationships*. These approaches use cardinality constraints to manage the relations between different processes. However, it is not possible to have the necessary awareness of process relationships by solely using cardinality constraints. Furthermore, processes may exhibit *many-to-many relationships*, which previously have not been considered. While the challenges regarding many-to-many relationships in an artifact-centric business process setting have been investigated, a solution that enables a coordination mechanism to have complete awareness over all process relationships is still missing. Specifically, [2] states an open challenge: *The need for a coordination mechanism to describe which processes interact with which other processes in settings that use many-to-many-relationships.* This challenge reveals its complexity when considering the processes at run-time, i.e., when considering process instances. In general, multiple instances of a process may be related to multiple instances of another process. Figure 1 shows a schematic example of interrelated process instances and their dependencies. Identifying the relations constitutes a prerequisite to adequately resolve and consider these dependencies at run-time, i.e., for coordinating processes. This challenge is central to every coordination mechanism that considers one-to-many and many-to-many relationships.

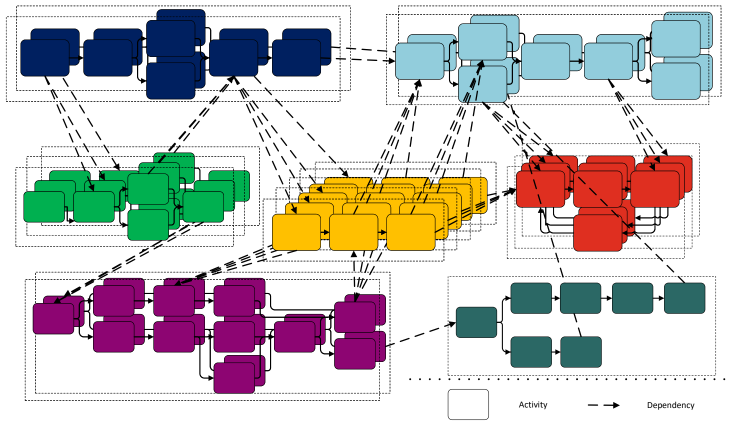


Fig. 1: Processes and their dependencies at run-time

This paper proposes a concept that solves this challenge and enables *process relation awareness* at both design- and run-time: the *relational process structure*. The complexities of managing and monitoring processes and their relations are covered by the relational process structure, which may be used in a generic fashion, i.e., the relational process structure is agnostic to specific coordination mechanisms or approaches. At design-time, the relational process structure allows identifying processes and capturing the relations between them. It inherently supports many-to-many relationships between processes and considers cardinality constraints. At run-time, the approach automatically keeps track of instantiated processes and relations created between different process instances. This enables a coordination mechanisms to apply queries to the relational process structure, e.g., to determine which processes are related to a particular object instance. It also enforces the cardinality constraints of the different process relationships with the process instances. As the number of instances and, therefore, the relational process structure may become large, different measures are employed to reduce query times. The run-time considerations set the relational process structure apart form conventional approaches, e.g., ER diagrams.

The remainder of the paper is organized as follows. The problem statement is elaborated in Section 2. Section 3 discusses the design-time aspects of the relational process structure. Section 4 deliberates on the dynamic aspects and functions of the relational process structure during run-time. Considerations concerning the optimizations of a relational process structure at run-time are examined in Section 5. Section 6 presents the application of the concept in the PHILharmonicFlows prototype. Section 7 discusses related work before concluding the paper with a summary and an outlook in Section 8.

2 Problem Statement

The primary challenge of achieving *process relation awareness* is keeping track of process instances and their relations at run-time. In other words, it should be possible to identify the number and specific identifiers of related process instances at any point in time and for all process instances. The logistics process described in Example 1 exhibits many of the characteristics of interrelated processes. Moreover, the problem of interrelated processes may also be observed in domains such as Human Resources or Healthcare [5], indicating the need for keeping track of processes and their relations.

Example 1. (Simplified logistics process)
An online retailer has various products on offer. A customer may place an order at the website of the retailer, requesting one or more products. The retailer handles the order by creating a bill for the customer. Once the bill has been paid, the retailer gets the ordered products from the storage and packages them for delivery. An order may be split into multiple packages, which may be distributed among multiple deliveries. A delivery may further comprise packages from different orders. If packages cannot be delivered, they are assigned to a subsequent delivery. The customer order is completed once all packages of the order have been delivered.

Order, product, bill, package, and delivery represent the processes in this example, i.e., they can be viewed as *entities* with a *lifecycle process*. The processes are interdependent. For example, a product cannot be delivered without an existing order, a product must be packaged in order to be delivered, and an order must have all its deliveries completed before it may be completed itself. Furthermore, these processes exhibit one-to-many and many-to-many relationships. For example, an order may contain one or more products, as does a package. A delivery may contain multiple packages. An order may be distributed across deliveries, and a delivery may be associated with one or more orders, i.e., it contains products from different orders, constituting a many-to-many relationship. Therefore, relations also indicate a dependency between processes.

At run-time, an instance of an order is connected to specific product instances. Other order instances may be connected to specific, but different product instances. Later, product instances will establish relations to a specific package instance. A delivery will obtain relations to specific package instances and, consequently, will establish a relation to the order instances the packages belong to. Furthermore, a delivery is not directly connected to instances of product, but transitively via a *path* of relations. In particular, this means that dependencies exist between processes having no direct relation, e.g., the execution of a delivery instance depends on the execution status of its transitively related product instances. It is crucial that a coordination mechanism is aware of these relations.

Given the description of Example 1, it may be perceived as fairly static, with little changes in the number of related process instances over time. In principle, however, the nature of the run-time is highly dynamic. Process instances may be created or deleted at any point in time. During this time, relations may be established between process instances and, consequently, may be removed later. Due to the existence of transitive relations, the creation of new relations may not only make a connection to one process instance, but to an entire substructure of related process instances. For example, with the creation of a relation between a package and a delivery and assuming the package contains products, the delivery is now related transitively to a specific number of product instances. In general, this might have significant consequences for the coordination of these processes.

The concept of the *relational process structure* aims to provide a complete map to the relations of different process instances at run-time. Further, it must keep track of the dynamic changes that occur during run-time, delivering accurate and up-to-date information to the coordination mechanism that manages these interdependent processes. The relational process structure is intended as a generic, but capable solution to this challenge. Any possible coordination approach is required to have process relation awareness. The relational process structure serves as a foundation on which specialized approaches for coordinating interdependent process instances may build on.

To be capable of monitoring process instances and their relations, a design-time model must first identify which types of processes may exist in a given context and what relations may be established between them. This achieves process relation awareness at design-time, and coordination mechanisms are able to use the explicitly known process and relation types to define the coordination needed between these processes.

3 Design-time Specification of the Relational Process Structure

At design-time, a relational process structure serves to capture the types of processes and relations that exist in the context of the overall business process. A relational process structure distinguishes between design-time entities, denoted as *types*, and run-time entities, denoted as *instances*. At run-time, several instances may be created by instantiating a type. The relational process structure does not assume that processes have a specific structure or use a predefined modeling notation; it is agnostic to the modeling paradigm and notation of the process, as well as to the specifics of the coordination mechanism used.

In principle, this enables a relational process structure to be used with any approach that deals with multiple processes and their relations. At design-time, the relational process structure is denoted as a *relational process type structure*. A formal definition of a relational process type structure and the basic process type definition are given in Definitions 1 and 2, respectively. The definitions use shorthand notations to identify types and instances. Superscript notation ^{T} denotes a design-time entity, i.e., a type, whereas superscript ^{I} denotes a run-time entity, i.e., an instance. The dot (.) represents the access operator.

Definition 1. *A relational process type structure d^T has the form (n, Ω^T, Π^T) where*

- *n is a unique identifier (name) of the relational process type structure.*
- *Ω^T is the set of object types ω^T (cf. Definition 2).*
- *Π^T is the set of relation types π^T (cf. Definition 3).*

The relational process type structure defines the context in which process types and relation types exist. Through Ω^T and Π^T, it provides an entry point for external clients, e.g., a coordination mechanism. It is represented as a graph where process types are the vertices and relation types are represented as edges.

Definition 2. *A process type ω^T has the form (d^T, n, θ^T) where*

- *d^T is the relational process type structure to which this process type belongs (cf. Definition 1).*
- *n is the unique identifier (name) of the process type.*
- *θ^T is a process specification.*

A process type requires an identifier, which has to be unique in the given context d^T, i.e, the relational process type structure. The identifier n may be indicated as ω^T_n. The details of the process specification θ^T is unimportant for the relational process structure, i.e., the relational process structure is paradigm-agnostic. Regarding Example 1, the process types include ω^T_{Order}, ω^T_{Bill}, $\omega^T_{Delivery}$, $\omega^T_{Package}$, and $\omega^T_{Product}$. However, their relations have not been identified yet. A process type stores two sets of *relation types*, identifying its incoming and outgoing relations. A relation type is represented as a directed edge between process types. A formal definition of a relation type is given in Definition 3.

Definition 3. *A relation type π^T represents an m:n relation and has the form $(\omega^T_{source}, \omega^T_{target}, m_{upper}, m_{lower}, n_{upper}, n_{lower})$ where*

- *ω^T_{source} is the source process type, i.e., π^T is directed.*

- ω^T_{target} *is the target process type.*
- m_{upper} *is an upper bound on the number of process instances* ω^I_{target} *with which* ω^I_{source} *may be related. Default:* $m_{upper} = \infty$.
- m_{lower} *is a lower bound on the number of process instances* ω^I_{target} *with which* ω^I_{source} *may be related. Default:* $m_{lower} = 0$.
- n_{upper} *is an upper bound on the number of process instances* ω^I_{source} *with which* ω^I_{target} *may be related. Default:* $n_{upper} = \infty$.
- n_{lower} *is a lower bound on the number of process instances* ω^I_{source} *with which* ω^I_{target} *may be related. Default:* $n_{lower} = 0$.

A relation type possesses a source and a target process type, i.e., the relation type is *directed*. It represents a many-to-many ($m : n$) relationship between its source and target. The directed edges are used to indicate a semantic hierarchy among processes, which can often be observed (cf. Figure 2). Each of the cardinalities m and n may be restricted by an upper and lower bound. Using the cardinality restrictions, a relation type may also be reduced to a one-to-many (1:n, m:1) or one-to-one (1:1) relationship. The actual number of process instances related to each other at run-time is restricted by the lower and upper bounds. For example, the online shop from Example 1 may establish that a delivery must contain at least 3 packages before it will be sent out, in order to reduce shipping costs. A relational process type structure uses a cardinality annotation for relation types of the following form:

$$m_{lower}..m_{upper} : n_{lower}..n_{upper}$$

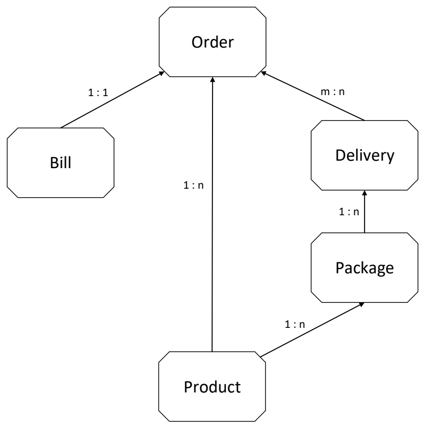


Fig. 2: Relational process model of the logistics process example

If the lower and upper bound of a cardinality are equal, or each bound possesses its default value, the notation may be shortened accordingly or replaced by the arbitrary cardinality, i.e., m or n. Figure 2 shows the relational process type structure for the processes from Example 1. Relation types have been identified and added between the process types ω^T_{Order}, ω^T_{Bill}, $\omega^T_{Delivery}$, $\omega^T_{Package}$, and $\omega^T_{Product}$. From Definitions 1-3, it can be seen that the relational process structure supports many-to-many relationships and cardinality constraints.

Two process types are said to be *related*, either *transitively* or *directly*, if there exists any *path* of relation types between them. The existence of a path may be determined with function $path^T$ as set out in Definition 4. Note that a path itself, as an entity, is defined as an ordered set of relation types, whereas function $path^T$ determines whether a path exists between two process types.

Definition 4. *Let $d^T = (n, \Omega^T, \Pi^T)$ be a relational process type structure. Then: function $path^T : \Omega^T \times \Omega^T \to \mathbb{B}$ determines whether a directed path of relations π^T from $\omega_i^T \in \Omega^T$ to $\omega_j^T \in \Omega^T$ exists, i.e., if ω_i^T, ω_j^T are related.*

$$path^T(\omega_i^T, \omega_j^T) := \begin{cases} true & \exists \pi_{out}^T \in w_i^T.\Pi_{out}^T : \pi_{out}^T.\omega_{target}^T = \omega_j^T \\ path^T(\omega_k^T, \omega_j^T) & \exists \pi_{out}^T \in w_i^T.\Pi_{out}^T : \pi_{out}^T.\omega_{target}^T = \omega_k^T, \\ & \omega_i^T \neq \omega_k^T \neq \omega_j^T \\ false & otherwise \end{cases}$$

With function $path^T$, it becomes possible to define two sets $L_{\omega^T}^T$ and $H_{\omega^T}^T$ (cf. Definition 5) that describe other processes in relation to a particular process type ω^T. The terms *lower-level* and *higher-level* hereby refer to the direction of the relations of the respective paths. There terms may describe the kind of a process relation directly, i.e., a process type may be a lower-level process of ω^T.

Definition 5. *Let $\omega^T = (d^T, n)$ be a process type.*

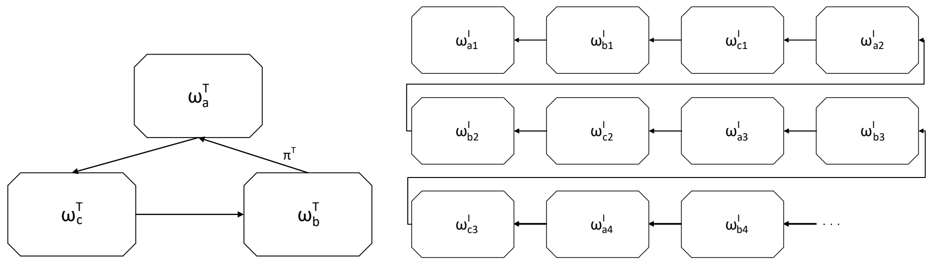
a) The set of lower-level process types $L_{\omega^T}^T$ is defined as
$$L_{\omega^T}^T = \{\, \omega_k^T \mid \omega_k^T \in \omega^T.d^T.\Omega^T \,,\, path(\omega_k^T, \omega^T) \,\} \cup \{\omega^T\}$$
b) The set of higher-level process types $H_{\omega^T}^T$ is defined as
$$H_{\omega^T}^T = \{\, \omega_k^T \mid \omega_k^T \in \omega^T.d^T.\Omega^T \,,\, path(\omega^T, \omega_k^T) \,\} \cup \{\omega^T\}$$

Function $path^T$ and sets $L_{\omega^T}^T$ and $H_{\omega^T}^T$ can be defined for process instances in the same way. For the sake of clarity, process type ω^T in the subscript of sets L and H may be replaced by the identifier $\omega^T.n$. The sets facilitate the definition of several concepts in respect to process relations, and can be used for a run-time optimization as proposed in Section 5.

A relational process structure is represented as a directed, acyclic graph. Using directed edges to represent relations provides several benefits. First, the direction of a relation corresponds directly to its cardinalities. The target process type always possesses cardinality m, whereas the source process type always has cardinality n, allowing for easy assignment of cardinalities in the relational process structure. Second, directed edges allow two processes to be related to one common process without the relational process structure containing a cycle. In Figure 2, process types ω_{Order}^T and $\omega_{Delivery}^T$ are (transitively) related to process type $\omega_{Product}^T$, i.e., ω_{Order}^T, $\omega_{Delivery}^T \in L_{Product}^T$. With directed edges, the relations do not form a cycle. If undirected edges had been used, the same relational process structure would contain a cycle. Acyclic graphs with undirected edges correspond to trees, which are too restrictive to represent a relational process structure, as they prohibit commonly found relations between processes, such as the one involving ω_{Order}^T, $\omega_{Delivery}^T$, and $\omega_{Product}^T$.

A relational process type structure interdicts the existence of cycles in its graph, i.e., it is represented as an acyclic graph. The reason for the acyclicity of the relational process structure is that cycles circumvent the cardinality restrictions of a relation type. This is caused by the fact that a relational process type structure explicitly considers transitive relations. Assume that the graph of a relational process type structure contains a cycle and a specific relation π^T with

$\pi^T.n_{upper} = 3$ and $\pi^T.\omega^T_{target} = \omega^T_a$ is part of this cycle, as shown exemplarily in Figure 3a. Then, a process instance ω^I_{a1} of type ω^T_a could be related to more than three process instances of type $\pi^T.\omega^T_{source} = \omega^T_b$. Each relation type of the cycle may be instantiated multiple times, first connecting ω^I_{a1} to an instance ω^I_{b1}. Subsequently, ω^I_a is transitively connected to another instance ω^I_{a2} of the same process type. The cycle can be repeated arbitrarily often, as shown in Figure 3b. Ultimately, ω^I_{a1} is transitively related to more than three instances of ω^T_b using relation type π^T, which renders the cardinality restriction on π^T moot. With the acyclicity of the graph, the relational process structure ensures that cardinality constraints must not be circumvented, as would be the case when using ER-diagrams having undirected edges and cycles.



(a) Relational process type structure with cycle

(b) A possible relational process instance structure resulting from the cyclic type structure

Fig. 3: Transitive relations at run-time with cyclic type structure

Note that a process instance may still be related to more instances of a specific type than allowed by the cardinality restrictions on the direct relation between them. There may be other relations that transitively connect to the same process type, using a different path through the relational process type structure graph. For example, ω^T_{Order} in Figure 2 is connected to $\omega^T_{Product}$ directly via $\pi^T_{Order-Product}$ and transitively via $\pi^T_{Order-Delivery}$. As a consequence, an order instance may be related to products of other orders sharing the same delivery. It is therefore crucial to take specific relation types into account when determining related instances of type $\omega^T_{Product}$ at run-time. The ability to define such sophisticated queries is only enabled by the relational process type structure. In turn, this also highlights the complexity that arises when considering transitive one-to-many and, especially, many-to-many relationships.

Transitive relations also possess a cardinality, determined by the cardinality of the relations on the path representing the transitive relation. It is possible to determine the cardinality of all transitive relations by calculating the transitive closure of the relational process type graph, e.g., by using a modified Floyd-Warshall Algorithm. Since the relational process type structure employs directed edges to represent many-to-many relationships, the modified Floyd-Warshall Algorithm needs to be employed twice: One time going in the direction of the edges and the second time going against. Transitive cardinalities help to discover modeling errors of the relational process structure at design-time.

The relational process structure shows similarity to diagrams at design-time. This may help a domain expert or process modeling expert to gain an entry to the relational process structure and to use it effectively. A relational process structure captures process types and allows displaying the relations between them. This enables process relation awareness at design-time. A coordination mechanism may use this information to specify the necessary coordination restrictions and enforce them at run-time. However, the relational process structure extends beyond the design-time and also demonstrates benefits at run-time. The specifics of these benefits will be discussed in Section 4.

4 Run-time Support of the Relational Process Structure

At run-time, the process and relation types may be instantiated. The created process instances and the interconnecting relation instances form a *relational process instance structure* (cf. Definition 6). At run-time, the continuous instantiation of processes and relations creates a highly dynamic environment, in which the relational process structure evolves dynamically as well.

Definition 6. *A relational process instance structure d^I has the form (d^T, Ω^I, Π^I) where*

- *d^T is the relational process type structure from which d^I has been instantiated.*
- *Ω^I is the set of process instances ω^I (cf. Definition 7).*
- *Π^I is the set of relation instances π^I (cf. Definition 8).*

Analogous to the relational process type structure, a relational process instance structure provides context in which instantiated processes and relations exist. In general, a multitude of different process instance structures, i.e., contexts, may exist in parallel during run-time. Furthermore, the graph of the relational process instance structure consists of process instances (cf. Definition 7) as vertices and relation instances (cf. Definition 8) as edges.

Definition 7. *A process instance ω^I has the form $(\omega^T, d^I, l, \theta^I)$ where*

- *ω^T is the process type from which ω^I has been instantiated.*
- *d^I is the relational process instance structure to which this object instance belongs (cf. Definition 6).*
- *l is the identifier (label) of the process instance. Default is $\omega^T.n$.*
- *θ^I is a process instance specification derived from $\omega^T.\theta^T$.*

Definition 8. *A relation instance π^I represents an m:n relation and has the form $(\pi^T, \omega^I_{source}, \omega^I_{target})$ where*

- *π^T is the relation type from which π^I has been instantiated.*
- *ω^I_{source} in the source process instance, i.e., π^I is directed.*
- *ω^I_{target} is the target process instance.*

As process and relation types may not just be instantiated once, but many times, additional complexity ensues in comparison to the relational process type structure. The process instances create a large and complex network, with possibly several independent sub-structures. However, in its basic structure, the process instances and relations evolve according to the specification of the process type structure. Thereby, a relational process instance structure resembles the corresponding relational process type structure. Regarding Example 1 and the relational process type structure depicted in Figure 2, one possible relational process instance structure is depicted in Figure 4.

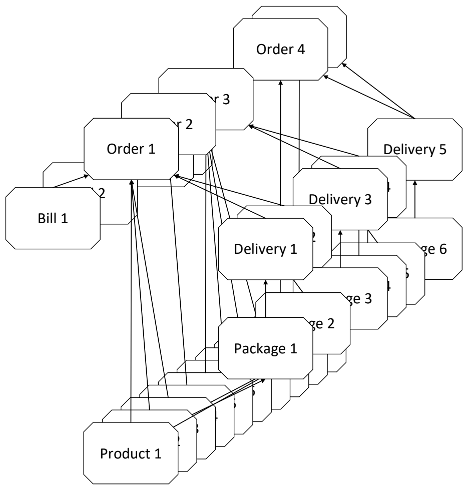


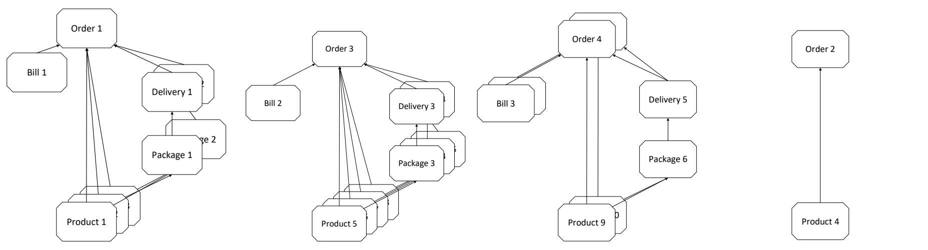
Fig. 4: Relational process instance structure for the logistics example

As instances of processes and relations may only be created if a corresponding type has been specified, tracking instances at run-time is greatly facilitated. When creating a relation π^I between two process instances ω_i^I and ω_j^I, it is first checked with the type structure whether the cardinality constraints of the relation type permit creating more relation instances between these specific process instances. The corresponding check can be performed efficiently, as process instances store their incoming and outgoing relation instances. By counting the number of relations instances π^I in $d^I.\Pi^I$ where $\pi^I.\omega_{target}^I = \omega_i^I$ that are instantiated from type π^T. This number can then simply be checked against the cardinality restrictions of π^T; the relation type may then be instantiated accordingly. By checking the minimum cardinality, the relational process type structure is capable of determining whether additional instances are needed.

The relational process instance structure keeps track of all instances by interconnecting them through $\pi^I.\omega_{target}^I$ and $\pi^I.\omega_{source}^I$ and storing them in $d^I.\Pi^I$ By keeping track which process types have been instantiated and what relation instances have subsequently been created between them, the relational process instance structure has full process relation awareness.

The relational process instance structure evolves over time and alters shape. When a coordination mechanism makes a query, e.g., about which products are contained in a specific package, the relational process instance structure is able to provide a reliable result. Subsequent additions to the package, i.e., new products are instantiated and new relations are created between them and the package, alter the result of the query. Should the same query be made at a later point in time, the relational process instance structure returns the updated result. Regarding the relational process structure, the term “query” is used to indicate the extraction of data from the relational process structure by an external agent. It is assumed that the agent corresponds to the coordination mechanism that requires the knowledge of process relationships to properly coordinate the process instances involved. Regarding the formal specification of queries, this paper remains intentionally vague to avoid unnecessary limitations, also regarding future extensions of the concept. However, examples are given throughout this paper, and the query "Which process instances are related to process instance ω^I?" serves as a reference.

The main run-time benefit of the relational process structure is the provision of detailed information on the relations of any given process. The creation of the graph representing the relational process instance structure comes at very little cost in terms of computation time. However, querying the graph in order to obtain desired information is, in general, computationally expensive. Some kinds of queries can be answered efficiently by the relational process instance structure, e.g., obtaining all directly related process instances of specific process instance ω^I_a requires the aggregation of all source processes $\{\omega^I \,|\, \omega^I \in d^I.\Omega^I, \exists \pi^I : \pi^I.\omega^I_{target} = \omega^I_a \wedge \pi^I.\omega^I_{source} = \omega^I\}$. However, as transitive relations constitute integral parts of the structure, most queries require a *depth-first search* or *breadth-first search* of the relational process structure. These possess time complexity of $O(|\Omega| + |\Pi|)$, in terms of processes Ω and relations Π of the relational process structure. Obviously, the time needed for queries increases when the relational process structure instance grows. For the remainder of the paper, it is assumed that queries use a depth-first search.



(a) Sub-structure A (b) Sub-structure B (c) Sub–structure C (d) Sub-structure D

Fig. 5: Sub-structures of the overall process instance structure

The main problem, however, is a different one. The continuously evolving process instance structure requires that a coordination mechanism also queries the relational process instance structure continuously in order to keep updated. Figures 5a-5d show different sub-structures of the relational process instance structure from Figure 4. The sub-structures show all interrelated process instances. When any of these sub-structures are altered, e.g., by adding a new relation to another process instance, another query must be made so that a coordination mechanism can discover the change. The sub-structures may even be combined, e.g., by connecting Delivery 5 in Figure 5c with Order 2 in Figure 5d. A significant change in (transitive) process relations may be observed, requiring even more queries to discover the specific changes in regard to each process instance involved.

While the relational process instance structure is responsible for increasing the query count due to dynamic changes and therefore prolonging the individual query execution time, another factor has not been considered yet. A coordination mechanism is not only affected by dynamic changes in process relations and process instance count, but also by changes regarding the progress of process instance execution. In principle, the progress of process instances is independent from the evolution of the relational process instance structure, i.e., process instances may change their execution status while the relational process instance structure remains unchanged. However, to discover changes in process status, additional queries must be issued to determine execution status changes of process instances. In general, not only process status, but any metadata might be queried. An optimization to address this problem is presented in Section 5.

5 Query Performance Optimization

For alleviating the problem of continuously querying the relational process instance structure and the resulting degraded performance due to the many depth-first searches, the caching of query results is not feasible. The continuous changes to the relational process instance structure and the progress of the process instances frequently require the invalidation of the cached query results. For this reason, a practical use of *query result caching* would have negligible performance benefits. As the query count lies outside the control of the relational process structure, reducing this number to improve overall performance is impossible.

However, as many queries are likely to include the determination of related process instances, reducing individual query time becomes necessary. Therefore, caching the related process instances for each process instance would reduce the number of depth-first searches that have to be performed. In effect, for each process instance $\omega_i^I \in d^I.\Omega^I$, sets L_i^I and H_i^I (cf. Definition 5) cache the result of depth-first searches and, thereby, maintain references to the current lower- and higher-level process instances. We denote this as *related process caching*. Then, a query may use these sets directly while the relational process structure remains unchanged. If the cache can be used, the query complexity is $O(|L^I| + |H^I|) \subset O(|\Omega| + |\Pi|)$. If only one of the sets is used for the query, the query time is reduced accordingly. This allows speeding up query execution time significantly by reducing the number of depth-first searches.

In fact, it is feasible to eliminate depth-first searches entirely by maintaining L_i^I and H_i^I for each process instance i along the construction of the relational process instance structure. The basic idea is as follows: when a relation is newly instantiated, the sets of the target and source objects are updated with the newly related process instances. If this is done beginning with the first instantiated relation, the overall state of the relational process instance structure is always kept consistent. The construction of the relational process instance structure with related process caching and its correctness will be shown exemplarily for set L^I. The construction works analogously for set H^I. In the beginning, suppose two process instances ω_a^I and ω_b^I have been instantiated and no relation exists. Then $L_a^I = \{\omega_a^I\}$ and $L_b^I = \{\omega_b^I\}$. Without loss of generality, a relation is created with ω_b^I as source and ω_a^I as target. Accordingly set $L_a^I = L_a^I \cup L_b^I = \{\omega_a^I, \omega_b^I\}$ is obtained. This is correct, as ω_b^I now has a path to ω_a^I. Set L_b^I remains unchanged, as ω_b^I has gained to new lower-level instances. In the general case, if a new relation is created with ω_j^I as source and ω_i^I as target, and with the postulation that both sets L_i^I and L_j^I contain the correct process instances, set $L_i^I = L_i^I \cup L_j^I$. Every $\omega_k^I \in L_j^I$ now fulfills $path^I(\omega_k^I, \omega_i^I)$ due to the newly created relation.

While L_i^I is correct, the overall process instance structure is inconsistent, as higher-level process instances of ω_i^I, i.e., set H_i^I, are related to the new lower-level process instances in L_j^I. Therefore, L_j^I must be propagated to the process instances in H_i^I, i.e., $\forall \omega_h^I \in H_i^I : L_h^I \cup L_j^I$. As can be seen, the overall process instance structure is consistent, i.e., every process instance has the correct lower-level process instances cached. Consequently, querying may be performed faster. As a drawback, the related process instance caching increases the time for creating a new relation between process instances. However, as the number of newly instantiated relations is expected to be significantly lower than the number of queries, a significant overall performance increase can be achieved.

6 Application to the Object-aware Approach

To provide a first evaluation of the relational process structure, it is applied to the object-aware process support paradigm [4]. The relational process structure is used to organize object types and their corresponding lifecycle processes, extending the previous data model with many-to-many relationships. The lifecycle processes of different objects are coordinated using *semantic relationships* [9], a concept that explicitly requires the identification and monitoring of relations between objects. Thereby, the object-aware approach takes full advantage of the capabilities of the relational process structure. The specification of semantic relationships requires the identification of relation types at design-time, i.e., it uses the relational process type model. At run-time, the exact relations between objects are crucial for a proper coordination. The necessary information is provided by the relational process instance structure.

The relational process structure, as presented in this paper, has been fully realized in the implementation of the object-aware approach, named PHILharmonicFlows. More precisely, the rudimentary implementation of the data model has been replaced with the relational process structure, offering many-to-many relation support. This replacement also improved the run-time capabilities of PHILharmonicFlows significantly, offering full-fledged run-time support. Various preliminary test results of the performance of the PHILharmonicFlows system show that the optimizations presented in Section 5 reduces query time by orders of magnitude, while increasing the time for creating a relation instance only moderately. A thorough evaluation with reliable numbers of PHILharmonicFlows performance will be presented in a future publication. Currently, PHILharmonicFlows, and with it the relational process structure, is moving towards a highly scalable architecture using microservices [1].

7 Related Work

The coordination of large process structures with focus on the engineering domain is considered in [6,7]. The COREPRO approach explicitly considers process relations with one-to-many cardinality. However, it does not consider many-to-many relationships and transitive relations. Furthermore, relationships cannot be restricted by cardinality constraints.

Artifacts consist of a lifecycle model and an information model [8] . The lifecycle model is described suing the Guard-Stage-Milestone (GSM) metamodel [3]. The information model may store any information required for the operation of the artifact. While relations of artifacts play a significant role in the specification of a business process, a concept similar to the relational process structure is missing. Nonetheless, process modelers may use the information model and the GSM lifecycle to replicate the same functionality. However, it creates high efforts for the process modeler and is error-prone. Leveraging the functionality of the relational process structure is therefore beneficial and easily realizable.

Proclets [10,11] are lightweight processes with a focus on process interactions. They interact via messages called *performatives.* Proclets allow specifying the cardinality for a message multicast, i.e., the number of Proclets receiving a performative. However, this number is fixed at design-time. The specification of Proclets does not include many-to-many-relationships between Proclets. Each Proclet requires a direct channel to exchange a performative. For this reason, transitive relations are not considered in the Proclet approach. While Proclets specify cardinality constraints, the exact recipients of a performative at run-time are unknown. Common to all these approaches is that their focus is almost exclusively on design-time issues. The many challenges arising from providing run-time support are not considered.

8 Summary and Outlook

The concept of the relational process structure allows modeling processes and their relations at design-time, accounting for many-to-many relationships, transitive relations, and cardinality constraints between processes. At run-time, the design-time information is used to automatically track process instances and their relations to other process instances. Thereby, the cardinality constraints may be enforced. At any point in time it is possible to obtain accurate information about process instances and their relations. Optimizations for increasing performance during run-time have been proposed. The whole relational process structure has been implemented in the PHILharmonicFlows system.

The relational process structure, as presented in this paper, allows tracking many-to-many relationships and enforcing cardinality constraints. However, several extensions to enhance the functionality and performance may be added in the future, e.g., restricting the overall number of process instances of a certain type, which also poses new challenges. While the implementation in PHILharmonicsFlows shows the applicability and general viability of the relational process structure concept, an evaluation with user studies and performance assessments of the system will be conducted to ascertain and solidify the benefits of this concept.

Acknowledgments. This work is part of the project ZAFH Intralogistik, funded by the European Regional Development Fund and the Ministry of Science, Research and the Arts of Baden-Württemberg, Germany (F.No. 32-7545.24-17/3/1)

References

1. Andrews, K., Steinau, S., Reichert, M.: Towards Hyperscale Process Management. In: 8th Int'l Workshop on Enterprise Modeling and Information Systems Architectures (EMISA). pp. 148–152. CEUR-WS.org (2017)
2. Fahland, D., de Leoni, M., van Dongen, B.F., van der Aalst, W.M.P.: Many-to-Many: Some Observations on Interactions in Artifact Choreographies. In: 3rd Central-European Workshop on Services and their Composition (ZEUS), 2011. vol. 705, pp. 9–15. CEUR-WS.org (2011)
3. Hull, R., Damaggio, E., Fournier, F., Gupta, M., Heath,III, Fenno Terry, Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P.N., Vaculín, R.: Introducing the Guard-Stage-Milestone Approach for Specifying Business Entity Lifecycles. In: 7th Int'l Workshop on Web Services and Formal Methods (WS-FM) 2010. LNCS, vol. 6551, pp. 1–24. Springer (2011)
4. Künzle, V., Reichert, M.: PHILharmonicFlows: Towards a Framework for Object-aware Process Management. Journal of Software Maintenance and Evolution: Research and Practice 23(4), 205–244 (2011)
5. Lenz, R., Reichert, M.: IT Support for Healthcare Processes – Premises, Challenges, Perspectives. Data & Knowledge Engineering 61(1), 39–58 (2007)
6. Müller, D., Reichert, M., Herbst, J.: Data-driven Modeling and Coordination of Large Process Structures. In: 15th Int'l Conf. on Cooperative Information Systems (CoopIS). pp. 131–149. LNCS, Springer (2007)
7. Müller, D., Reichert, M., Herbst, J.: A New Paradigm for the Enactment and Dynamic Adaptation of Data-driven Process Structures. In: 20th Int'l Conf. on Advanced Information Systems Engineering (CAiSE). pp. 48–63. LNCS, Springer (2008)
8. Nigam, A., Caswell, N.S.: Business Artifacts: An Approach to Operational Specification. IBM Systems Journal 42(3), 428–445 (2003)
9. Steinau, S., Künzle, V., Andrews, K., Reichert, M.: Coordinating Business Processes Using Semantic Relationships. In: 19th IEEE Conf. on Business Informatics (CBI). pp. 33–43. IEEE Computer Society Press (2017)
10. van der Aalst, W.M.P., Barthelmess, P., Ellis, C.A., Wainer, J.: Workflow Modeling using Proclets. In: 7th Int'l Conf. on Cooperative Information Systems (CoopIS), 2000. pp. 198–209. Springer (2000)
11. van der Aalst, W.M.P., Barthelmess, P., Ellis, C.A., Wainer, J.: Proclets: A Framework for Lightweight Interacting Workflow Processes. International Journal of Cooperative Information Systems 10(04), 443–481 (2001)