Modeling and enacting complex data dependencies in business processes

Modeling and Enacting Complex Data Dependencies in Business Processes

Andreas Meyer, Luise Pufahl, Dirk Fahland,
Mathias Weske

Technische Berichte Nr. 74

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam | 74

Andreas Meyer | Luise Pufahl | Dirk Fahland | Mathias Weske

Modeling and Enacting Complex Data Dependencies in Business Processes

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.de/ abrufbar.

Universitätsverlag Potsdam 2013
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2013/6510/
URN urn:nbn:de:kobv:517-opus-65103
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-65103

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-245-2

Modeling and Enacting Complex Data Dependencies in Business Processes

Andreas Meyer[1], Luise Pufahl[1], Dirk Fahland[2], and Mathias Weske[1]

[1] Hasso Plattner Institute at the University of Potsdam
{Andreas.Meyer,Luise.Pufahl,Mathias.Weske}@hpi.uni-potsdam.de
[2] Eindhoven University of Technology
d.fahland@tue.nl

Abstract. Enacting business processes in process engines requires the coverage of control flow, resource assignments, and process data. While the first two aspects are well supported in current process engines, data dependencies need to be added and maintained manually by a process engineer. Thus, this task is error-prone and time-consuming. In this report, we address the problem of modeling processes with complex data dependencies, e.g., m:n relationships, and their automatic enactment from process models. First, we extend BPMN data objects with few annotations to allow data dependency handling as well as data instance differentiation. Second, we introduce a pattern-based approach to derive SQL queries from process models utilizing the above mentioned extensions. Therewith, we allow automatic enactment of data-aware BPMN process models. We implemented our approach for the Activiti process engine to show applicability.

Keywords: Process Modeling, Data Modeling, Process Enactment, BPMN, SQL

1 Motivation

The purpose of enacting processes in process engines or process-aware information systems is to query, process, transform, and provide data to process stakeholders. Process engines such as Activiti [4], Bonita [5] or AristaFlow [12] are able to enact the control flow of a process and to allocate required resources based on a given process model in an automated fashion. Also simple data dependencies can be enacted from a process model, for example, that an activity can only be executed if a particular data object is in a particular state. However, when *m:n relationships* arise between processes and data objects, modeling and enactment becomes more difficult.

For example, Fig. 1 shows a typical *build-to-order process* of a computer manufacturer in which customers order products that will be custom built. For an incoming *Customer order*, the manufacturer devises all *Components* needed to build the product. Components are not held in stock, but the manufacturer on demand creates and executes a number of *Purchase orders* to be sent to various *Suppliers* to procure the Components required. To reduce costs, Components of multiple Customer orders are bundled in joint Purchase orders. The two subprocesses of Fig. 1 handle complex m:n relationships between the different orders: one Purchase order contains Components of multiple Customer orders and one Customer order depends on Components of multiple Purchase orders.

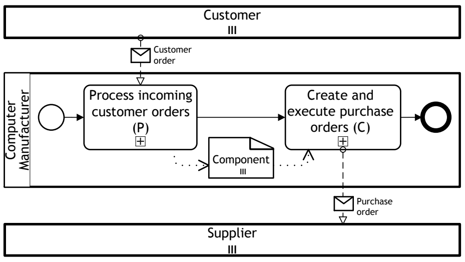

Customer
Customer order
Computer Manufacturer
Process incoming customer orders (P)
Create and execute purchase orders (C)
Component
Purchase order
Supplier

Fig. 1: Build-to-order process, where subprocess P collects multiple orders from several Customers in an internal loop and where C sends multiple Purchase orders to several Suppliers using a multi instance subprocess internally.

Widely accepted process modeling languages such as BPMN [16] do not provide sufficient modeling concepts for capturing m:n relationships between data objects, activities, and processes. As a consequence, actual data dependencies are often not derived from a process model. They are rather implemented manually in services and application code, which yields high development efforts and may lead to errors.

Explicitly adding data dependencies to process models provides multiple advantages. In contrast to having data only specified inside services and applications called from the process, an integrated view facilitates *communication with stakeholders* about processes and their data manipulations; there are *no hidden dependencies*. With execution semantics one can *automatically enact* processes with complex data dependencies from a model only. Finally, an integrated conceptual model allows for *analyzing control and data flow combined* regarding their consistency [11, 23] and correctness. Also *different process representations can be generated automatically*, for instance, models showing how a data object evolves throughout a process [9, 13].

Existing techniques for integrating data and control flow follow the “object-centric” paradigm [3, 6, 10, 14]: a process is modeled by its involved objects; each one has a life cycle and multiple objects synchronize on their state changes. This paradigm is beneficial when process flow follows from process objects, e.g., in manufacturing processes [14]. However, there are many domains, where processes are rather “activity-centric” such as accounting, insurance handling, or municipal procedures. In these, execution follows an explicitly prescribed ordering of domain activities, not necessarily tied to a particular object life cycle. For such processes, changing from an activity-centric view to an object-centric view for the sake of data support has disadvantages. Besides having to redesign all processes in a new paradigm and training process modelers, one also has to switch to new process engines and may no longer be supported by existing standards. This gives rise to a first requirement (**RQ1-activity**): processes can be modeled in an activity-centric way using well-established industrial standards for describing process dynamics and data dependencies.

In this paper, we address the problem of modeling and enacting *activity-centric* processes with complex data dependencies. The problem itself was researched for more than a decade revealing numerous requirements as summarized in [10]. The following requirements of [10] have to be met to enact activity-centric processes with complex data dependencies directly from a process model:

(RQ2-data integration) The process model refers to data in terms of object types, defines pre- and post-conditions for activities (cf. requirements R01 and R14 in [10]), and

(RQ3-object behavior) expresses how data objects change (cf. R04 in [10])

(RQ4-object interaction) in relation and interaction with other data objects; objects are in 1:1, 1:n, or m:n relationships. Thereby, process execution depends on the state of its interrelated data objects (cf. R05 in [10]) and

(RQ5-variable granularity) an activity changes a single object, multiple related objects of different types, or multiple objects of the same type (cf. R17 in [10]).

In this paper, we propose a technique that addresses the requirements (RQ1)-(RQ5). The technique combines classical activity-centric modeling in BPMN [16] with relational data modeling as known from relational databases [20]. To this end, we introduce few extensions to BPMN data objects: Each data object gets dedicated life cycle information, an object identifier, and fields to express any type of correlation, even m:n relationships, to other objects with identifiers. We build on BPMN's extension points ensuring conformance to the specification [16]. These data annotations define pre- and post-conditions of activities with respect to data. We show how to automatically derive SQL queries from annotated BPMN data objects that check and implement the conditions on data stored in a relational database. For demonstration, we extended the Activiti process engine [4] to automatically derive SQL queries from data-annotated BPMN models.

The remainder of this paper is structured as follows. In Section 2, we discuss the current data modeling capabilities of BPMN including shortcomings. Then, in Section 3, we present our technique for data-aware process modeling with BPMN, which we give operational semantics in Section 4. There, we also discuss the SQL derivation. Section 5 introduces all patterns required to apply the semantics in the presented setting. We discuss our implementation in Section 6 before we review related work in Section 7 and conclude in Section 8.

2 Data Modeling in BPMN

BPMN [16], a rich and expressive modeling notation, is the industry standard for business process management and provides means for modeling as well as execution of business processes. In this section, we introduce BPMN's existing capabilities for data modeling and its shortcomings with respect to the requirements introduced above.

So far, we used the term "data object" with a loose interpretation in mind. For the remainder, we use the terminology of BPMN [16], which provides the concept of *data objects* to describe different types of data in a process. Data flow edges describe which activities read or write which data objects. The same data object may be *represented* multiple times in the process distinguishing distinct read or write accesses. A data flow edge from a *data object representation* to an activity describes a read access to an *instance* of the data object, which has to be present in order to execute the activity. A data object instance is a concrete data entry of the corresponding data object. A data flow edge from an activity to a data object representation describes a write access, which creates a data object instance, if it did not exist, or updates the instance, if it existed before. Fig. 2 shows two representations of data object *D*, one is read by activity *A* and one is written. Data object representations can be modeled as a *single instance* or as a *multi instance* (indicated by three parallel bars) that comprises a set of instances of one data object. Further, a data object can be either *persistent* (stored in a database) or *non-persistent* (exists only while the process instance is active). Our approach focuses on persistent single and multi instance data objects.

The notion of an *object life cycle* emerged over the last years for giving data objects a behavior. The idea is that each data object *D* can be in a number of different states. A process activity *A* reading *D* may only get enabled if *D* is in a particular state; when *A* is executed object *D* may transition to a new state. To express this behavior, BPMN provides the concept of *data states*, which allows to annotate each data object with a *[state]*. Fig. 2 shows an example: Activity *A* may only be executed when the respective object instance is indeed in *state X*; after executing the activity, this object instance is in *state Y*.

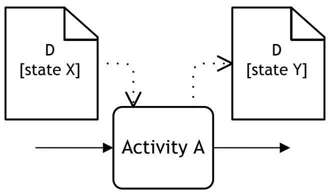


Fig. 2: Object life cycle of data object D with two representations.

The BPMN semantics is not sufficient to express all data dependencies in a process model with respect to the following four aspects. The annotations to data object representations in Fig. 2 do not allow to distinguish different object instances of *D* in the same process instance, e.g., two different customer orders. Likewise, we cannot express how several instances of different data objects relate to each other. Further, the type of a write access on data objects, e.g., creation or update, is not clear from the annotations shown above. Finally, the correlation between a process instance and its object instances is not supported. Next, we propose a set of extensions to BPMN data objects to overcome the presented shortcomings.

3 Extending BPMN Data Modeling

In this section, we introduce annotations to BPMN data objects to overcome the shortcomings utilizing *extension points*, which allow to extend BPMN and still being standard conform. With these, we address requirements (RQ1)-(RQ5) from the introduction. In the second part, we illustrate the extensions on a build-to-order process.

3.1 Modeling Data Dependencies in BPMN

To distinguish and reference data object instances, we utilize proven concepts from relational databases: primary and foreign keys [20]. We introduce *object identifiers* as an annotation that describes the attribute by which different data object instances can be distinguished (i.e., primary keys). Along the same lines, we introduce attributes, which allow to refer to the identifier of another object (cf. foreign keys in [20]).

Fig. 3 shows annotations for primary key (pk) and foreign key (fk) attributes in BPMN data object representations. Instances of *D* are distinguishable by attribute *d_id* and instances of *E* by attribute *e_id*. In Fig. 3a, each instance of *D* is related to one instance of *E* by the fk attribute *e_id*, i.e., a 1:1 relationship. The activity *A* can only

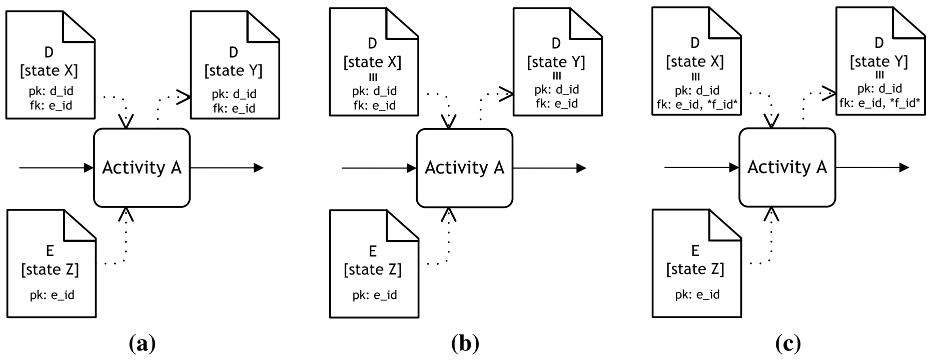


Fig. 3: Describing object interactions in (a) 1:1, (b) 1:n, and (c) m:n cardinality.

execute when one instance e of *E* is in *state Z* and one instance d of *D* is in *state X* that is related to e exist. Upon execution, d enters *state Y* whereas e remains unchanged. A *multi instance* representation of D expresses a 1:n relationship from E to D as shown in Fig. 3b, e.g., several computer components for one customer order. To execute activity *A*, *all* instances of D related to e have to be in *state X*; the execution will put all instances of D into *state Y*. We allow *multi-attribute foreign keys* to express m:n relationships between data objects as follows. Assume, data objects D, E, F have primary keys *d_id*, *e_id*, *f_id*, respectively, and D has foreign key attributes *e_id*, *f_id*. Each instance of D (e.g., a component) refers to one instance of E (e.g., a customer order it originated from) and one instance of F (e.g., a purchase order in which it is handled). Different instances of D may refer to the same instance e of E (e.g., all components of the same customer order) but to different instances of F (e.g., handled by different purchase orders) and vice versa. This yields an m:n relationship between E and F via D. We allow to *all-quantify* over foreign keys by enclosing them in asterisks, e.g., **f_id** in Fig. 3c. Here, activity A updates *all instances* of D from *state X* to *state Y* if they are related to the instance e of E and to any instance of F, that is, we quantify over **f_id**. A foreign key attribute can be *null* indicating that the specific reference is not yet set. A data object may have further attributes, however, these are not specified in the object itself but in a data model, possibly given as UML class diagram [17], accompanying the process model.

In order to derive all data dependencies from a process model, we need to be able to express the four major data operations: *create*, *read*, *update*, and *delete* for a data object instance (see Fig. 4). Read and update are already provided through BPMN's data flow edges. To express create or delete operations, we need to add two annotations shown in the upper right corner: *[new]* expresses the creation of a new data object instance having

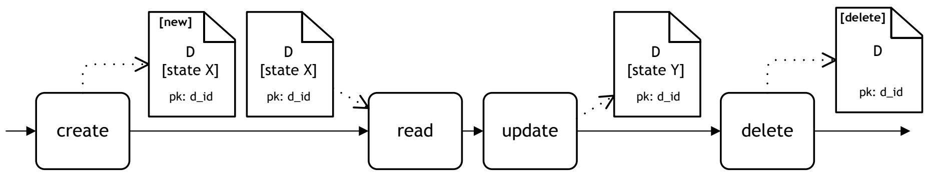


Fig. 4: Describing create, read, update, and delete of a data object.

a completely fresh identifier and *[delete]* expresses its deletion. Note that one activity can apply several data operations to different data objects. For example, activity A in Fig. 3a reads and updates an instance of D and reads an instance of E.

The introduced extensions require that a data object contains a *name* and a set of attributes, from which one needs to describe a *data state*, an *object identifier* (primary key), and a *set of relations* to other data objects (foreign keys). Fig. 5 summarizes these extensions for a data object representation. Based on the informal considerations above, we formally define such extended representation of a BPMN data object as follows.

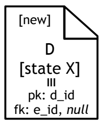


Fig. 5: Extended data object representation.

Definition 1 (Data object representation). A *data object representation* $r = (name, state, pk, FK, FK^*, \eta, \omega)$ refers to the *name* of the data object, has a $state$, a *primary key* (pk), a finite set FK of *foreign keys*, a set $FK^* \subseteq FK$ of all-quantified foreign keys, and a data operation type $\eta \in \{new, delete, \bot\}$. $\omega \in \{singleInstance, multiInstance\}$ defines the instance multiplicity property. ◇

$\bot$ as element of set η refers to a blank data operation description for which the data access is derived from the data flow: an input data flow requires a read operation while an output data flow requires an update operation.

To let a specific *process instance* create or update specific data object instances, we need to link these two. For this, we adopt an idea from business artifacts [15] that each process instance is "driven" by a specific data object instance. We call this object *case object*; all other objects have to be related to it by means of foreign keys. This idea naturally extends to instances of subprocesses or multi-instance activities. Each of them defines a *scope* which has a dedicated instance id. An annotation in a scope defines which data object acts as case object. A case object instance is either freshly created by its scope instance based on a *new* annotation (the object instance gets the id of its scope instance as primary key value). Alternatively, the case object instance already exists and is passed to the scope instance upon creation (the scope instance gets the id of its case object instance). By all means, a case object is always *single instance*. For the presentation of our approach in Section 3 and 4, we assume that all non case objects are directly related to the case object. In Section Section 5, lift our approach to the general case such that data objects can be also indirectly related to the case object, i.e., via an other data object. We make data objects and case objects part of the process model as follows, utilizing a subset of BPMN [16].

Definition 2 (Process model). A *process model* $M = (N, R, DS, C, F, P, type_A, case, type_G, \kappa)$ consists of a finite non-empty set $N \subseteq A \cup G \cup E$ of *nodes* being *activities* A, *gateways* G, and *events* E, a finite non-empty set R of *data object representations*, and the finite set DS of *data stores* used for persistence of data objects (N, R, DS are pairwise disjoint). $C \subseteq N \times N$ is the *control flow* relation, $F \subseteq (A \times R) \cup (R \times A)$ is the *data flow* relation, and $P \subseteq (R \times DS) \cup (DS \times R)$ is the *data persistence* relation; $type_A : A \to \{task, subprocess, multiInstanceTask, multiInstanceSubprocess\}$ gives each activity a type; $case(a)$ defines for each $a \in A$ where $type_A(a) \neq task$ the case object. Function $type_G : G \to \{xor, and\}$ gives each gateway a type; partial function $\kappa : F \nrightarrow exp$ optionally assigns an expression exp to a data flow edge. ◇

An expression at a data flow edge allows to refer to data attributes that are neither state nor key attribute, as we show later. As usual, a process model M is assumed to be structural sound, i.e., M contains exactly one start and one end event and every node of M is on a path from the start to the end event. Further, each activity has at most one incoming and one outgoing control flow edge.

3.2 Example

In this section, we apply the syntax introduced above to model the build-to-order scenario presented in the introduction. The scenario consists of two interlinked process models and the corresponding data model. The scenario comprises the collection of customer orders, presented in Fig. 7, and the arrangement of purchase orders based on the customer orders received, presented in Fig. 8. Each customer order can be fulfilled by a set of purchase orders and each purchase order consolidates the components required for several customer orders. This m:n relationship is expressed in the data model in Fig. 6.

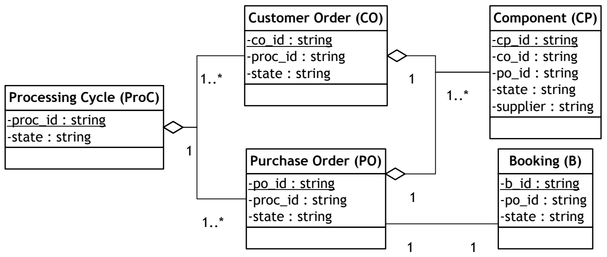


Fig. 6: Data model.

Data model. The *processing cycle* (ProC) contains information about *customer orders* (CO) being placed by customers and *purchase orders* (PO) used to organize the purchase of components within a particular time frame. Data object *component* (CP) links CO and PO in an m:n-fashion, i.e., CP has two foreign keys, one to CO and one to PO. CO and PO each have one foreign key to ProC. Accounting of the manufacturer is performed utilizing data object *booking* (B). For simplicity, we assume that all data is persisted in the same data store, e.g., the database of the manufacturer, and omit representations of the data store in the process diagrams.

Customer order collection process. In Fig. 7, the first task starts a new processing cycle allowing customers to send in orders for computers. By annotation *new*, a new ProC object instance is created for each task execution. As this is the case object of the

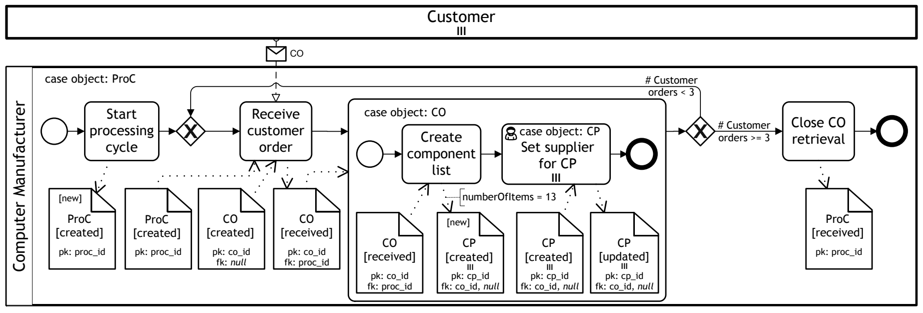


Fig. 7: Build-to-order scenario: customer order collection.

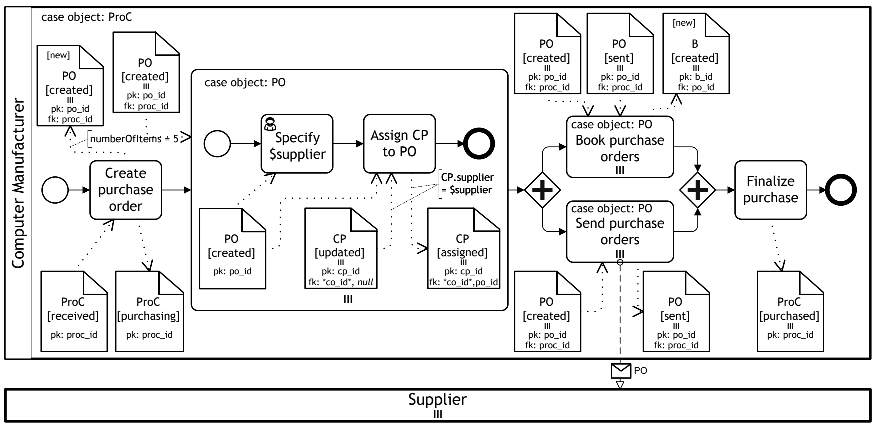


Fig. 8: Build-to-order scenario: purchase order arrangement.

process, the primary key *proc_id* gets the id of the process instance as value. Next, COs are collected in a loop structure until three COs have been successfully processed. Task *Receive customer order* receives one CO from a customer and correlates this CO instance to the ProC instance of the process instance (annotation *fk: proc_id*) before it is analyzed in a subprocess. CO is the case object of the subprocess, which gets its instance id from the primary key of the received CO instance. Task *Create component list* determines the components needed to handle the CO: several CP instances are created (annotation *new* on a multi instance object representation). Each CP instance has a unique primary key value; the foreign key attribute *co_id* referring to CO is set to the current CO instance; the foreign key attribute referring to PO is still *null*. The number of CP instances to create is given in the expression on the data output flow edge. Here, we give an explicit number, but it could also be a process variable holding the result of the task execution (e.g., user input, result of a service invocation). Next, an user updates the attribute *CP.supplier* for each component (CP) to indicate where it can be purchased, e.g., by using a form. The loop structure is conducted for each received CO and repeated until three COs are collected. CO retrieval is closed by moving the current ProC to state *received*.

Purchase order arrangement process. The second process model in Fig. 8 describes how components (extracted from different COs) are associated to purchase orders (POs), building an m:n relationship between POs and COs. Object ProC links both processes, the process in Fig. 8 can only start when there is a ProC object instance in state *received*.

Create purchase order creates multiple PO object instances correlated to the ProC instance. All PO instances are handled in the subsequent multi instance subprocess: for each PO instance one subprocess instance is created, having the PO instance as case object and the corresponding *po_id* value as instance identifier. Per PO, first, one supplier is selected that will handle the PO; here we assume that the task *Select supplier* sets a process variable *$supplier* local to the subprocess instance. Task *Assign CP to PO* relates to the PO *all* CP instances in state *updated* that have no *po_id* value yet and where attribute *CP.supplier* equals the chosen *$supplier*. The relation is built by setting the value of *CP.po_id* to the primary key *PO.po_id* of the case object. The update quantifies over all values of *co_id* as indicated by the asterisks.

The execution of the multi instance subprocess results in several CP subsets each being related to one PO. The POs along with the contained information about the CPs are sent to the corresponding supplier. In parallel, *Book purchase orders* creates a new booking for each PO; it may start when either all POs are in *created* or in *sent*.

Object life cycle. Altogether, our extension to BPMN data objects increases the expressiveness of a BPMN process model with information about process-data-correlation on instance level. As such, it does not interfere with standard BPMN semantics.

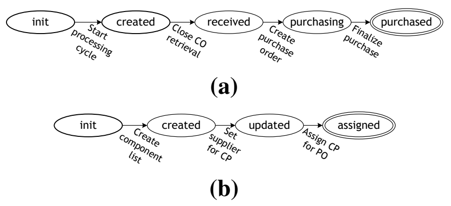


Fig. 9: Object life cycles of objects (a) *ProC* and (b) *CP* derived from the process model.

In addition, our extension is compatible with the object life cycle oriented techniques allowing to derive object life cycles from sufficiently annotated process models [9, 13]. Taking our build-to-order process, we can derive the object life cycles shown in Fig. 9.

4 Executing Data-annotated BPMN Models

This section presents *operational execution semantics* for the data annotated process models defined in Section 3. Aiming at standardized techniques, we *refine* the standard BPMN semantics [16, Section 13] with SQL database queries (see Section 4.1) that are derived from annotated input and output data objects (see Section 4.2).

4.1 Process Model Semantics

Our semantics distinguishes control flow and data flow aspects of a process model M. A *state* $s = (C, \mathcal{D})$ of M consists of a control flow state C describing a distribution of tokens on sequence flow edges and activities and a database $\mathcal{D}$ storing the data objects of M in tables. To distinguish the states of different process instances, each token in C is an identifier id. The data model of the process is implemented in a relational database $\mathcal{D}$ (shared by all processes). Each data object is represented in $\mathcal{D}$ as a table; columns represent attributes, having at least columns for primary key, foreign keys (if any), and state. Each row in a table describes an instance of this data object with concrete values.

An activity A has several input and output data object representations, grouped into *input sets* and *output sets*; different input/output sets represent alternative pre-/postconditions for A. A representation R of an input object is *available* in instance id if the corresponding table in $\mathcal{D}$ holds a particular row. We can define a *select* query $Q_R(id)$ on $\mathcal{D}$ and a guard $g_R(id)$ that compares the result of $Q_R(id)$ to a constant or to another select query; $g_R(id)$ is *true* iff R is available in id. A representation R of an output object of A has to become available when A completes. We operationalize this by executing an *insert*, *update*, or *delete* query $Q_R(id)$ on $\mathcal{D}$ depending on R.

Activity A is *enabled* in instance id in state $s = (C, \mathcal{D})$ iff a token with id id is on the input edge of A and for some input set $\{R_1, \ldots, R_n\}$ of A, each guard $g_{R_i}(id)$ is *true*. If A is enabled in C, then A gets *started*, i.e., the token id moves “inside” A in step $(C, \mathcal{D}) \rightarrow (C', \mathcal{D})$ and depending on the type of activity services are called, forms are shown, etc. When this instance of A *completes*, the outgoing edge of A gets a token id and the database gets updated in a step $(C', \mathcal{D}) \to (C'', \mathcal{D}')$, where $\mathcal{D}'$ is the result of executing queries $Q_{R_1}(id), \ldots, Q_{R_m}(id)$ for some output set $\{R_1, \ldots, R_m\}$ of A. The semantics for gateways and events is extended correspondingly. If activity A is a subprocess with case object D, and A has D as data input object, then we create a *new instance* of subprocess A for each entry returned by query $Q_D(id)$. Each subprocess instance is identified by the primary key value of the corresponding row of D. Next, we explain how to derive queries from the data object representations.

4.2 Deriving Database Queries from Data Annotations

The annotated data object representations defined in Section 3 describe pre- and post-conditions for the execution of activities. In this section, we show how to derive from a data object representation R (and its context) a guard g_R or a query Q_R that realizes this pre- or post-condition.

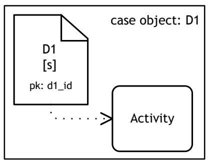
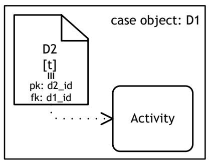
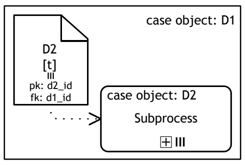
In a combinatorial analysis, we considered the occurrence of a data object as *case object*, as single dependent object with 1:1 relationship to another object, and as multiple dependent object with 1:n or m:n relationship in the context of a *create*, *read*, *update*, and *delete* operation. Additionally, we considered process instantiation based on existing data and reading/updating object attributes other than state. Altogether, we obtained a complete collection of *43 parameterized patterns* regarding the use of data objects as pre- or post-conditions in BPMN (see Section 5). For each of these patterns, we defined a corresponding database query or guard. During process execution, each input/output object is matched against the patterns. The guard/query of the matching pattern is then used as described in Section 4.1. Here, we present the five patterns that are needed to execute the subprocess in the model in Fig. 8; Tab. 1 and 2 list the patterns and their formalization that we explain next. All 43 patterns and their formalization are given in Section 5.

As introduced in Section 3, we assume that each scope (e.g., subprocess) is driven by a particular case object. Each scope instance has a dedicated instance id. The symbol \$ID refers to the instance id of the directly enclosing scope; \$PID refers to the process instance id.

Read single object instance. Pattern 1 describes a read operation on a single data object D1 that is also the case object of the scope. The activity is only enabled when this case object is in the given state s. The guard shown below P1 in Tab. 1 operationalizes this behavior: it is *true* iff table D1 in the database has a row where the state attribute has value ‘s’ and the primary key *d1_id* is equal to the scope instance id.

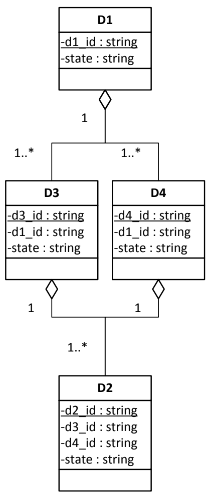
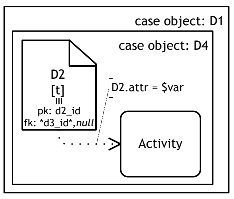
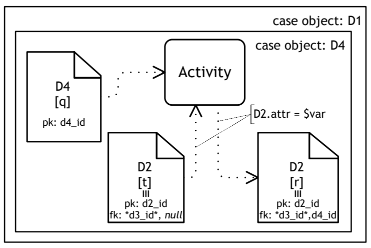
Read multiple object instances. Pattern 2 describes a read operation on multiple data object instances of D2 that are linked to the case object D1 via foreign key *d1_id*. The activity is only enabled when all instances of D2 are in the given state t. This is captured by the guard shown below P2 in Tab. 1 that is *true* iff the rows in table D2 that are linked to the D1 instance with primary key value \$ID are also the rows in table D2 where state = ‘t’ (and the same link to D1); see Section 5 for the general case of arbitrary tables between D1 and D2. For example, consider the second process of the build-to-order scenario (see Fig. 8). Let us assume that activity *Create purchase order* was just executed

Tab. 1: SQL queries for patterns 1 to 3 for subprocess in Fig. 8.

P1	P2	P3
	 case object: D1 D2 [t] pk: d2_id fk: d1_id Activity	 case object: D1 D2 [t] pk: d2_id fk: d1_id case object: D2 Subprocess

guard:
(**SELECT** **COUNT**(d1.d1_id)
FROM d1
WHERE d1.d1_id = $ID
AND d1.state='s') ≥ 1
	guard:
	(**SELECT** **COUNT**(d2.d2_id)
	FROM d2
	WHERE d2.d1_id = $ID
	AND d2.state='t') =
	(**SELECT** **COUNT**(d2.d2_id)
	FROM d2
	WHERE d2.d1_id =$ID)
		For each *d2_id* ∈ (
		SELECT d2.d2_id
		FROM d2
		WHERE d2.d1_id = $ID)
		start subprocess
		with id *d2_id*

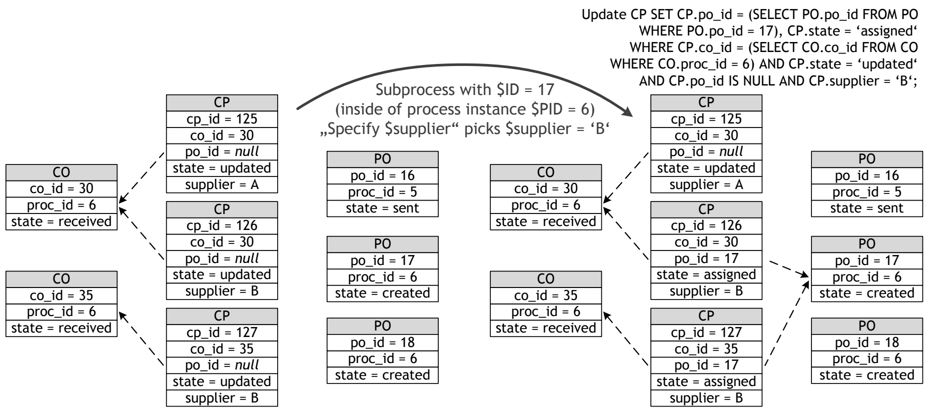
Tab. 2: SQL queries for patterns 4 and 5 for subprocess in Fig. 8.

Data model	P4	P5
 D1 -d1_id : string -state : string 1 1..* 1..* D3 -d3_id : string -d1_id : string -state : string D4 -d4_id : string -d1_id : string -state : string 1 1 1..* D2 -d2_id : string -d3_id : string -d4_id : string -state : string	 case object: D1 case object: D4 D2 [t] pk: d2_id fk: "d3_id",null D2.attr = $var Activity	 case object: D1 case object: D4 D4 [q] pk: d4_id Activity D2.attr = $var D2 [t] pk: d2_id fk: "d3_id", null D2 [r] pk: d2_id fk: "d3_id",d4_id

	guard:
	(**SELECT** **COUNT**(d2.d2_id)
	FROM d2
	WHERE d2.state = 't'
	AND d2.d4_id IS **NULL**
	AND d2.attr = $var
	AND d2.d3_id = (
	SELECT d3.d3_id
	FROM d3
	WHERE d3.d1_id = $PID)
	) >= 1
		UPDATE d2
		SET d2.d4_id = (
		SELECT d4.d4_id
		FROM d4
		WHERE d4.d4_id = $ID),
		state = 'r'
		WHERE d2.state = 't'
		AND d2.d4_id IS **NULL**
		AND d2.attr = $var
		AND d2.d3_id = (
		SELECT d3.d3_id
		FROM d3
		WHERE d3.d1_id = $PID)

for process instance 6 and the database table of the purchase order (PO) contains the entries shown in Fig. 10a. All rows with $proc\_id = 6$ are in state *created*, i.e., both queries of pattern 2 yield the same result and the subprocess gets instantiated.

Instantiate subprocesses from data. Pattern 3 deals with the instantiation of a multi instance subprocess combined with a read operation on the dependent multi instance data object D2. As described in Section 4.1, we create a new instance of the subprocess for each id returned by the query shown below P3 in Tab. 1. For our example, where



```
Update CP SET CP.po_id = (SELECT PO.po_id FROM PO
WHERE PO.po_id = 17), CP.state = 'assigned'
WHERE CP.co_id = (SELECT CO.co_id FROM CO
WHERE CO.proc_id = 6) AND CP.state = 'updated'
AND CP.po_id IS NULL AND CP.supplier = 'B';
```

(a) Before update (b) After update

Fig. 10: Setting missing foreign key relation of m:n object *Component*: Concrete update statement of subprocess 17 to relate all CPs referring to supplier B to the PO with ID 17 indicated by arrows.

process instance 6 is currently executed, the subprocess having the PO as case object is instantiated twice, once with id 17 and once with id 18. In each subprocess instance, control flow reaches activity *Select supplier* for which pattern 1 applies. For the subprocess instance with id 17, the guard of Pattern 1 evaluates to true of the state in Fig. 10a: activity *Select Supplier* is enabled.

Transactional properties. Patterns 4 and 5 illustrate how our approach updates m:n-relationships. Pattern 4 describes a read operation on multiple data object instances D2 that share a particular attribute value and are *not* related to the case object (in contrast to Pattern 2). We have to ensure that another process instance does not interfere with reading (and later updating) these instances of D2, that is, we have to provide *basic transactional properties*. We achieve this by accessing only those instances of D2 that are in some way related to the current process instance. Therefore, this read operation assumes a data model as shown in Tab. 2(left): D2 defines an m:n relationship between D3 and D4 via foreign keys d3_id and d4_id; D3 and D4 both have foreign keys to D1 which is the case object of the process; see Section 5 for the general case. The guard shown below P4 in Tab. 2 is true iff there is at least one instance of D2 in state t, with a particular attribute value, not linked to D4, and where the link to D3 points to an instance that itself is linked to the case object instance of the process (i.e., foreign key of D3 points to $PID). The link to D3 ensures that the process instance only reads D2 instances and no other process instance can read. In our example, the pattern occurs at task *Assign CP to PO* reading all instances of object component (CP), which are not yet assigned to a PO (i.e., *null* value as foreign key) and where $CP.supplier = \$supplier$. Assume the state shown in Fig. 10a and that $\$supplier = B$ was set by task *Select* $\$supplier$ for the subprocess instance with ID 17. In this state, the queries of pattern 4 return two rows having a *null* value for *po_id*, *B* as supplier value, and *updated* as state value: the activity is enabled.

Updating m:n relationships. Finally, pattern 5 describes an update operation on multiple data object instances of D2, which sets the foreign key d4_id that is not set yet and moves them to state r. All instances of D2 get as value for d4_id the instance id of the current instance of case object D4. Semantically, this turns the select statement of pattern 4 into an update statement that sets attributes *d4_id* and *state* for all rows where the pre-condition holds; see the SQL query of pattern 5 in Tab. 2. In our example, pattern 5 occurs at task *Assign CP to PO* for assigning a specific set of components (CP) to a purchase order (PO) based on the chosen supplier. As assumed for the subprocess instance with ID 17, the process variable *$supplier* has the value B. The entire derived query is shown in Fig. 10b (top right); executing the query gives components with ID 126 and ID 127 concrete references to PO ($po_id = 17$), and the state *assigned*. The resulting state of the database in Fig. 10b shows the m:n relationship that was set.

5 Patterns

This section is dedicated to introduce all patterns required to derive data dependencies from a process model and enact them on a process engine. A relational database is used for persistence. For each pattern, the corresponding generic SQL pattern is presented; Tab. 3 provides an overview about the assignment of each pattern to the following classification schema. The patterns are classified with respect to two dimensions: (i) type of data condition (horizontally) and (ii) data object function (vertically). Horizontally, they are classified into *pre-* and *post-conditions* of an activity with respect to the data operation. While the fulfillment of pre-conditions decides about the enablement of an activity, post-conditions must apply at termination of an activity (cf. Section 3.1). Pre-conditions (guards, cf. Section 4.1) are logical expressions, which consist of one or more *select* statements. Post-conditions are further subdivided into *insert*, *update*, and *delete* statements.

Vertically, the patterns are classified regarding whether the operation is executed on the case object (that is bound to the process instance), on a single dependent data object (being in 1:1 relationship with another object), or multiple dependent data objects (being

Tab. 3: Pattern classification overview.

Data operation	Case object (Section 5.1)	Dependent 1:1 (Section 5.2)	Dependent 1:n (Section 5.3)	Dependent m:n (Section 5.4)
select	CR1 CR2	$D^{1:1}$R1 $D^{1:1}$R2 $D^{1:1}$R3	$D^{1:n}$R1 $D^{1:n}$R2 $D^{1:n}$R3	$D^{m:n}$R1 $D^{m:n}$R2 $D^{m:n}$R3 $D^{m:n}$R4
insert	CC1 CC2	$D^{1:1}$C1 $D^{1:1}$C2	$D^{1:n}$C1 $D^{1:n}$C2	$D^{m:n}$C1 $D^{m:n}$C2
update	CU1 CU2	$D^{1:1}$U1 $D^{1:1}$U2 $D^{1:1}$U3	$D^{1:n}$U1 $D^{1:n}$U2 $D^{1:n}$U3	$D^{m:n}$U1 $D^{m:n}$U2 $D^{m:n}$U3 $D^{m:n}$U4
delete	CD1	$D^{1:1}$D1	$D^{1:n}$D1	$D^{m:n}$D1 $D^{m:n}$D2
instantiation (Section 5.5)	I1, I2, I3, I4			
attribute (Section 5.6)	A1, A2			

in 1:n or m:n relationship with another object). In our running example, *ProC* is the case object, *customer order* directly depends on *ProC* in 1:1-fashion, *booking* data objects indirectly depend on *ProC* in 1:n-fashion via *purchase order* objects, and *component* objects indirectly depend on *ProC* in m:n-fashion via *purchase order* and *customer order* objects.

Further, we need patterns to distinguish different cases of instantiation, i.e., which object is used as a case object and how identifiers of case object and process instance are set. Finally, data objects may contain more attributes than those shown in a data object representation (cf. Definition 1). Thus, we also provide means to use data object attributes for control flow decisions and to automatically update these attributes. For example, activity *Update supplier to $supplier* in Fig. 7 updates the supplier attribute of CO to the value specified in the process variable *$supplier*.

The remainder of this section presents the patterns for each vertical category as well as the two final mentioned ones in separate subsections starting with the patterns for the case object in 5.1. Referring to Section 3.1, an activity might also be of type multi instance. Basically, this is a short form of the multi instance subprocess such that each multi instance activity can be remodeled as multi instance subprocess containing the activity and the corresponding data objects in their single type. The corresponding procedure is presented in Section 5.7.

5.1 Patterns for Case Object

In this section, all patterns and their SQL queries for the case object are presented. The corresponding data model, shown in Fig. 11, consists only of the case object D1 being in the focus of the subsequent queries. Thereby, the case object has the following by our approach required attributes: a primary key attribute *d1_id* and a state attribute *state*.

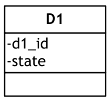
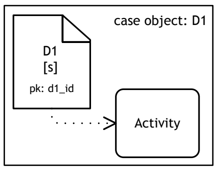


Fig. 11: Corresponding data model for case object.

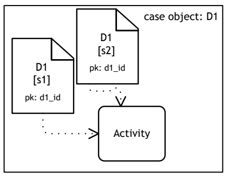
Tab. 4: Patterns for case object.

CR1 – Read single state



```
guard:
  (SELECT COUNT(d1_id)
  FROM d1
  WHERE d1_id = $ID
  AND state = 's') ≥ 1
```

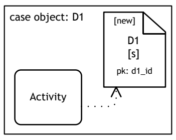
CR2 – Read multiple states



```
guard:
  (SELECT COUNT(d1_id)
  FROM d1
  WHERE d1_id = $ID
  AND state = ('s1' OR 's2')) ≥ 1
```

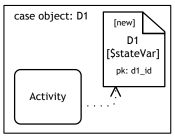
Tab. 4: Patterns for case object (ctd.).

CC1 – Create single state



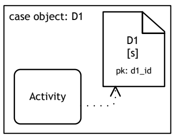
```
INSERT INTO d1
(d1_id, state)
VALUES ($ID,'s')
```

CC2 – Create multiple states



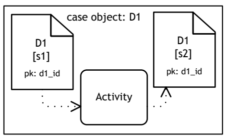
```
INSERT INTO d1
(d1_id, state)
VALUES ($ID,$stateVar)
```

CU1 – Update



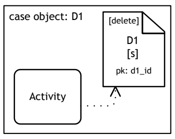
```
UPDATE d1
SET state = 's'
WHERE d1_id = $ID
```

CU2 – Update with required input



```
UPDATE d1
SET state = 's2'
WHERE d1_id = $ID
AND state = 's1'
```

CD1 – Delete



```
DELETE FROM d1
WHERE d1_id = $ID
AND state = 's'
```

CR1 – Read single state. The pattern describes a read operation on the case object of the surrounding scope. Read requires that the corresponding instance (i.e., the case object instance, which is related to the current scope instance via its primary key value) being in state *s* is available. This is checked by the SQL statement to the right returning all rows of the respective database table for the case object which are related to $*ID* and have state *s*. The guard ensures that the activity is only enabled if the result set of the query returns 1 or more rows.

CR2 – Read multiple states. The pattern describes a read operation on the case object similar to CR1, but it allows that the data object can be present in different states. In the pattern, the corresponding instance has to be available either in state *s1* or in state *s2*. As described in the SQL statement to the right, all rows of the case object table are selected which are related to $*ID* and have *s1* or *s2* as state value. The activity is enabled as soon one or more rows are returned.

CC1 – Create single state. The pattern describes a create operation on the case object. *Create* results in a new entry in the case object table with $*ID* of the current instance as primary key value and *s* as state value. This is achieved by the SQL query to the right executed at the termination of the activity.

CC2 – Create multiple states. The pattern describes a create operation on the case object similar to CC1, but the state is not statically given by the process model; it is dynamically set during activity execution by means of a process variable. So, a new entry is added to the case object table with $*ID* as primary key value and the process variable value of $*stateVar* as state value covered by the corresponding SQL query.

CU1 – Update. The pattern describes an update operation on the case object. At the termination of the activity, a new state is set for the corresponding case object instance. In terms of database design, the state value of the corresponding row in the case object table related to $*ID* is updated to *s* as shown in the SQL statement. Alternatively, also the process variable $*stateVar* can be used in the update statement for dynamically setting the state during activity execution as done in pattern CC2.

CU2 – Update with required input. The pattern describes an update operation on the case object similar to pattern CU1, but it additionally requires that the current case object instance is in the given state of the data input. The corresponding SQL statement selects only the row related to $*ID* with the state value *s1* and updates it to *s2*.

CD1 – Delete. The pattern describes a delete operation on the case object. At the termination of the activity, the corresponding case object instance is deleted, whereby the instance has to be in the given state. This is covered by the SQL statement, which considers as well the given state *s* in the WHERE-clause in order to avoid the deletion of wrong data object instances.

5.2 Patterns for Dependent$^{1:1}$ Objects

This section describes the patterns and their SQL queries for single instance data objects, which are in 1:1 relationship with another object and which are dependent to the case object. These patterns consider the generalized case where the dependent data object D2 has no foreign key directly pointing to the case object D1 but rather to another data object D3, which itself points to D1, directly or indirectly. The data dependencies are expressed in the data model shown in Fig. 12. In the data model, the case object has the following attributes required by our approach: a primary key attribute *d1_id* and a state attribute *state*. The dependent single instance data object D2, which is in the focus of the subsequent queries, has besides the primary key attribute *d2_id* and the state attribute *state* also a foreign key attribute *d3_id* pointing to D3. This holds as well for the other

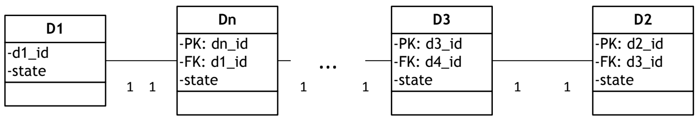
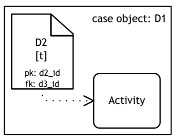


Fig. 12: Corresponding data model for dependent$^{1:1}$ objects.

dependent data objects D3, ..., Dn. From the data model, we can find a path D2, D3, ... Dn, D1 of data objects (or tables) from D2 to D1 along the foreign key relations. In terms of a database design, the inner join on all tables D2, D3, ..., Dn, D1 connects entries in D2 with entries in D1 using the respective identifiers as join attribute. We define the JOINALL statement to build this join for our queries, e.g., JOINALL(D2, D3, D4, D1)[3].

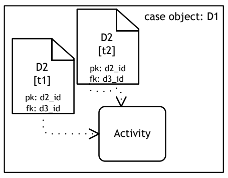
Tab. 5: Patterns for dependent$^{1:1}$ objects.

D$^{1:1}$R1 – Read single state



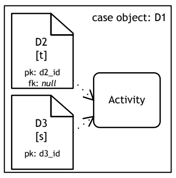
```
guard:
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL(d2, d3, ..., dn, d1)
  WHERE d1.d1_id = $ID
  AND d2.state = 't') ≥ 1
```

D$^{1:1}$R2 – Read multiple states



```
guard:
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL(d2, d3, ..., dn, d1)
  WHERE d1.d1_id = $ID
  AND d2.state =
  ('t1' OR 't2')) ≥ 1
```

D$^{1:1}$R3 – Read without foreign key

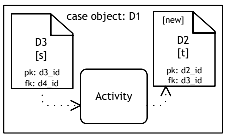


```
guard:
  (SELECT COUNT(d2_id)
  FROM d2
  WHERE d3_id IS NULL
  AND state = 't') ≥ 1
```

[3] `(((d2 INNER JOIN d3 USING d3_id) INNER JOIN d4 USING d4_id) INNER JOIN d1 USING d1_id)`

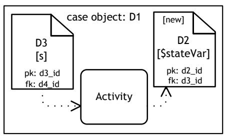
Tab. 5: Patterns for dependent$^{1:1}$ objects (ctd.).

$D^{1:1}$C1 – Create single state



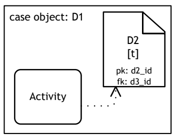
```
INSERT INTO d2
(d2_id, d3_id, state)
VALUES (DEFAULT, (SELECT d3.d3_id
FROM JOINALL(d3, ..., dn, d1)
WHERE d1.d1_id = $ID), 't')
```

$D^{1:1}$C2 – Create multiple states



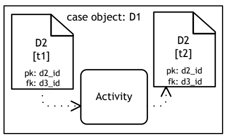
```
INSERT INTO d2
(d2_id, d3_id, state)
VALUES (DEFAULT,(SELECT d3.d3_id
FROM JOINALL(d3, ..., dn, d1)
WHERE d1.d1_id = $ID),
$stateVar)
```

$D^{1:1}$U1 – Update



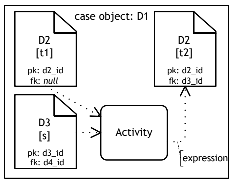
```
UPDATE d2
SET state = 't'
WHERE d3_id = (
  SELECT d3.d3_id
  FROM JOINALL(d3, ..., dn, d1)
  WHERE d1.d1_id = $ID)
```

$D^{1:1}$U2 – Update with required input



```
UPDATE d2
SET state = 't2'
WHERE d3_id = (
  SELECT d3.d3_id
  FROM JOINALL(d3, ..., dn, d1)
  WHERE d1.d1_id = $ID)
AND state = 't1'
```

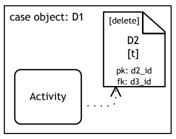
$D^{1:1}$U3 – Update missing foreign key



```
UPDATE d2
SET d3_id = (
  SELECT d3.d3_id
  FROM JOINALL(d3, ..., dn, d1)
  WHERE d1.d1_id = $ID),
 state = 't2'
WHERE d3_id IS NULL
AND state = 't1'
```

Tab. 5: Patterns for dependent$^{1:1}$ objects (ctd.).

D$^{1:1}$D1 – Delete



```
DELETE FROM d2
WHERE d3_id = (
  SELECT d3.d3_id
  FROM JOINALL(d3, ..., dn, d1)
  WHERE d1.d1_id = $ID)
AND state='t'
```

D$^{1:1}$R1 – Read single state. This pattern describes a read operation on a dependent single instance data object. Read requires that the respective instance of the data object D2 being in state *t* is available. Using the statement JOINALL(D2, D3, ..., D1), we can build the join-table between D2 and D1 by means of the foreign key relations. In the join-table, each row with *d1_id* = $*ID* describes an instance of D2 that is related to the case object instance of the corresponding scope instance. This is used by the SQL statement to return all rows of the respective database table for D2 which are related to $*ID* and have state *t*. The guard ensures that the activity is only enabled if the result set of the query returns 1 or more rows.

D$^{1:1}$R2 – Read multiple states. The pattern describes a read operation on a dependent single instance data object similar to D$^{1:1}$R1, but it allows that the data object can be present in different states. In the pattern, the corresponding instance has to be available either in state *t1* or in state *t2*. As described in the SQL statement, all rows of the data object table of D2 are selected which are related to $*ID* and have *t1* or *t2* as state value. The activity is enabled as soon as one or more rows are returned.

D$^{1:1}$R3 – Read without foreign key. The pattern describes a read operation on a dependent single instance data object for which the foreign key value is not yet set, i.e., the data object instance is not yet correlated to a scope instance. The activity is enabled, if any instance of D2 exists with an empty foreign key relationship and being in state *t*. Covered by the corresponding SQL statement, all rows of the data object table of D2 are selected which have a *null*-value for the foreign key *d3_id* and *t* as state value. If one or more rows are returned, the activity can be started.

D$^{1:1}$C1 – Create single state. The pattern describes a create operation on a dependent single instance data object. *Create* results in a new entry in the data object table of D2 with a default primary key value, the respective D3 primary key value as foreign key value, and *t* as state value. The respective D3 primary key value is extracted by joining the table of D3 with the case object table D1 with the JOINALL statement and selecting the *d3_id* value of the row with *d1_id* = $*ID*, which is related to the current scope instance. This select statement is considered from the SQL query inserting a new row for D2 at the termination of the activity.

$D^{1:1}$C2 – Create multiple states. The pattern describes a create operation on a dependent single instance data object similar to $D^{1:1}$C1, but the state is not statically given by the process model; it is dynamically set during activity execution by means of a process variable. A new entry is added to the data object table of D2 with a default primary key value, the respective D3 primary key value as foreign key value, and the process variable value of $*stateVar* as state value covered by the corresponding SQL query.

$D^{1:1}$U1 – Update. The pattern describes an update operation on a dependent single instance data object. At the termination of the activity, a new state is set for the corresponding data object instance. In terms of database design, the state value of the corresponding row in the data object table of D2 related to $*ID* is updated to *t*. For the update, the data table row of D2 is selected where the foreign key value *d3_id* points to an entry in the table of D3 which is related to $*ID* determined by means of the JOINALL statement from D3 until the case object D1. This is covered by the corresponding SQL statement. Alternatively, also the process variable $*stateVar* can be used in the update statement for dynamically setting the state during activity execution as done in pattern $D^{1:1}$C2.

$D^{1:1}$U2 – Update with required input. The pattern describes an update operation on a dependent single instance data object similar to pattern $D^{1:1}$U1, but it additionally requires that the respective data object instance is in the given state of the data input. The corresponding SQL statement only selects the row related to $ID with the state value *t1* and updates it to *t2*.

$D^{1:1}$U3 – Update missing foreign key. The pattern describes an update operation on a dependent single instance data object, which has a not yet specified foreign key. Goal of this pattern is to link an uncorrelated data object instance of D2 to a scope instance by setting the corresponding foreign key value. The assignment is done randomly: The uncorrelated instance is taken and processed by this scope instance which is currently running. Thereby, the foreign key value is extracted by selecting the primary key value of the corresponding data object instance of D3 shown as input data. For the select statement, the JOINALL statement is used to join the table of D3 with the case object table D1 and to choose the *d3_id* value of the row with *d1_id* = $*ID*, which is related to the current scope instance. For the update, the row of the data object table of D2 is selected which has currently a *null*-value for *d3_id* and *t1* as state value. Then, the foreign key *d3_id* is set to a concrete value and the state is set to *t2*. This is covered by the corresponding SQL statement, which is executed at the termination of the activity.

$D^{1:1}$D1 – Delete. The pattern describes a delete operation on a dependent single instance data object. At the termination of the activity, the corresponding data object instance is deleted, whereby the instance has to be in the given state. This is covered by the SQL statement, which also considers the given state *t* in the WHERE-clause in order to avoid the deletion of wrong data object instances. For the deletion, the data table row of D2 is selected where the foreign key value *d3_id* points to an entry in the table of D3 which is related to $*ID* determined by means of the JOINALL statement from D3 until the case object D1. This is covered by the corresponding SQL statement.

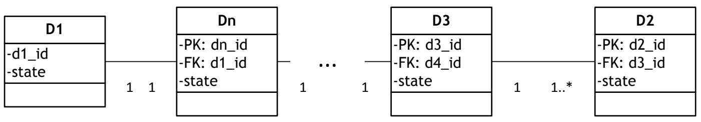


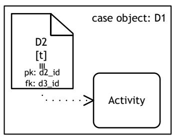
Fig. 13: Corresponding data model for dependent$^{1:n}$ objects.

5.3 Patterns for Dependent$^{1:n}$ Objects

This section describes the patterns and their SQL queries for multi instance data objects, which are in 1:n relationship with another object and which are dependent to the case object. These patterns consider the generalized case where dependent data object D2 has no foreign key directly pointing to the case object D1 but rather to another data object D3, which itself points to D1, directly or indirectly. The data dependencies are expressed in the data model shown in Fig. 13. In the data model, the case object D1 has the following attributes required by our approach: a primary key attribute *d1_id* and a state attribute *state*. The dependent multi instance data object D2, which is in the focus of the subsequent queries, has besides the primary key attribute *d2_id* and the state attribute *state* also a foreign key attribute *d3_id* pointing to D3. This holds as well for the other dependent data objects D3, ..., Dn. For the following queries, we will use the JOINALL statement for joins with the case object as described in Section 5.2.

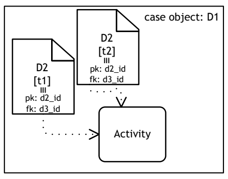
Tab. 6: Patterns for dependent$^{1:n}$ objects.

D$^{1:n}$R1 – Read single state



```
guard:
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL(d2, d3, ..., dn,d1)
  WHERE d1.d1_id = $ID
  AND d2.state = 't') =
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL(d2, d3, ..., dn, d1)
  WHERE d1_id = $ID )
```

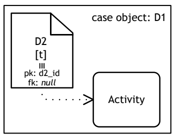
D$^{1:n}$R2 – Read multiple states



```
guard: (
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL(d2, d3, ..., dn, d1)
  WHERE d1.d1_id = $ID
  AND d2.state= 't1') =
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL(d2, d3, ..., dn, d1)
  WHERE d1_id = $ID ))
  xor (
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL(d2, d3, ..., dn, d1)
  WHERE d1.d1_id = $ID
  AND d2.state = 't2') =
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL(d2, d3, ..., dn, d1)
  WHERE d1.d1_id = $ID ))
```

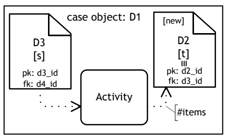
Tab. 6: Patterns for dependent$^{1:n}$ objects (ctd.).

D$^{1:n}$R3 – Read without foreign key



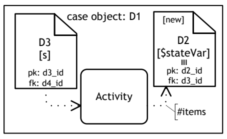
```
guard:
  (SELECT COUNT(d2_id)
  FROM d2
  WHERE d3_id IS NULL
  AND state = 't') ≥ 1
```

D$^{1:n}$C1 – Create single state



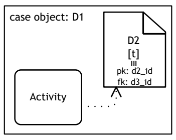
```
INSERT INTO d2
(d2_id, d3_id, state) VALUES
(DEFAULT, fk, 't')
...
(DEFAULT, fk, 't')
//#items times

fk = SELECT d3.d3_id
FROM JOINALL(d3,...,dn,d1)
WHERE d1.d1_id=$ID
```

D$^{1:n}$C2 – Create multiple states



```
INSERT INTO d2
(d2_id, d3_id, state) VALUES
(DEFAULT, fk, $stateVar)
...
(DEFAULT, fk, $stateVar)
//#items times

fk = SELECT d3.d3_id
FROM JOINALL(d3, ..., dn, d1)
WHERE d1.d1_id = $ID
```

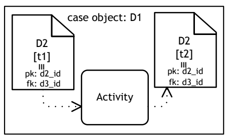
D$^{1:n}$U1 – Update



```
UPDATE d2
SET state = 't'
WHERE d3_id = (
  SELECT d3.d3_id
  FROM JOINALL(d3, ..., dn, d1)
  WHERE d1.d1_id = $ID)
```

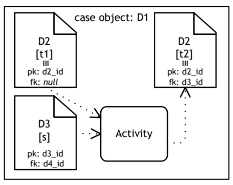
Tab. 6: Patterns for dependent$^{1:n}$ objects (ctd.).

D$^{1:n}$U2 – Update with required input



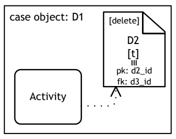
```
UPDATE d2
SET state = 't2'
WHERE d3_id = (
   SELECT d3.d3_id
   FROM JOINALL(d3, ..., dn, d1)
   WHERE d1.d1_id=$ID)
AND state = 't1'
```

D$^{1:n}$U3 – Update missing foreign key



```
UPDATE d2
SET d3_id = (
   SELECT d3.d3_id
   FROM JOINALL(d3, ..., dn, d1)
   WHERE d1.d1_id = $ID),
 state = 't2'
WHERE d3_id IS NULL
AND state = 't1'
```

D$^{1:n}$D1 – Delete



```
DELETE FROM d2
WHERE d3_id = (
   SELECT d3.d3_id
   FROM JOINALL (d3, ..., dn, d1)
   WHERE d1.d1_id = $ID)
AND state='t'
```

D$^{1:n}$R1 – Read single state. This pattern describes a read operation on a dependent multi instance data object. Read requires that the respective set of instances of the data object D2 being in state *t* is available. Using the statement JOINALL(D2, D3, ..., D1), we can build the join-table between D2 and D1 by means of the foreign key relations. In the join-table, each row with *d1_id* = $*ID* describes an instance of D2 that is related to the case object instance of the corresponding scope instance. This is used by the SQL statement to return all rows of the respective database table for D2 which are related to $*ID* and have state *t* (first select) and to return all rows of this table which are related to $*ID* independently from the state attribute (second select). The guard ensures that the activity is only enabled if all instances related to $*ID* are in state *t*.

D$^{1:n}$R2 – Read multiple states. The pattern describes a read operation on a dependent multi instance data object similar to D$^{1:n}$R1, but it allows that the data object can be present in different states. In the pattern, the corresponding instances have to be available either in state *t1* or in state *t2*; a mixture of states is not allowed due to BPMN semantics [16]. This is ensured by the guard expression to the right. For state *t1*, all rows of the data object table of D2 are selected which are related to $*ID* and have state *t1*. These are compared to all rows being related to $*ID* independently from the state. If both return the same number, the condition holds true. A similar check is done for other state where all rows related to $*ID* and have state *t2* are compared to all rows being related to $*ID*. The activity is enabled as soon as one of the conditions holds true.

$D^{1:n}$R3 – Read without foreign key. The pattern describes a read operation on a dependent multi instance data object for which the foreign key value is not yet set, i.e., the data object instances are not yet correlated to a scope instance. The activity is enabled, if any set of instances of D2 exists, where each instance has the same empty foreign key relationship and is in state *t*. Covered by the corresponding SQL statement, all rows of the data object table of D2 are selected which have a *null*-value for the foreign key *d3_id* and *t* as state value. If one or more rows are returned, the activity can be started.

$D^{1:n}$C1 – Create single state. The pattern describes a create operation on a dependent multi instance data object. *Create* results in new entries in the data object table of D2, each with a default primary key value, the respective D3 primary key value as foreign key value, and *t* as state value. The respective D3 primary key value is extracted by joining the table of D3 with the case object table D1 with the JOINALL statement and selecting the *d3_id* value of the row with *d1_id* = $*ID*, which is related to the current scope instance. This select statement is executed at first and the returned foreign key value is saved in the variable *fk*. The variable is used by the SQL query for each insertion of a new row for D2 at the termination of the activity. The number instances to be created is determined by the process variable *#items*, which is attached to the output data flow edge.

$D^{1:n}$C2 – Create multiple states. The pattern describes a create operation on a dependent multi instance data object similar to $D^{1:n}$C1, but the state is not statically given by the process model; it is dynamically set during activity execution by means of a process variable. Each new entry is added to the data object table of D2 with a default primary key value, the respective D3 primary key value as foreign key, and the process variable value of $*stateVar* as state value covered by the corresponding SQL query. Similar to $D^{1:n}$C1, the number instances to be created is determined by the process variable *#items*, which is attached to the output data flow edge.

$D^{1:n}$U1 – Update. The pattern describes an update operation on a dependent multi instance data object. At the termination of the activity, a new state is set for each of the corresponding data object instances. In terms of database design, the state value of the corresponding rows in the data object table of D2 related to $*ID* is updated to *t*. For the update, all data table rows of D2 are selected where the foreign key value *d3_id* points to an entry in the table of D3 which is related to $*ID* determined by means of the JOINALL statement from D3 until the case object D1. This is covered by the corresponding SQL statement. Alternatively, also the process variable $*stateVar* can be used in the update statement for dynamically setting the state during activity execution as done in pattern $D^{1:n}$C2.

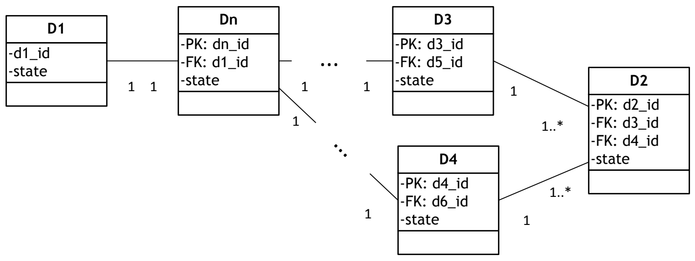


Fig. 14: Corresponding data model for dependent$^{m:n}$ objects.

$D^{1:n}$U2 – Update with required input. The pattern describes an update operation on a dependent multi instance data object similar to pattern $D^{1:n}$U1, but it additionally requires that the respective data object instances are in the given state of the data input. The corresponding SQL statement only selects the rows related to $ID with the state value *t1* and updates it to *t2*.

$D^{1:n}$U3 – Update missing foreign key. The pattern describes an update operation on a dependent multi instance data object, which has a not yet specified foreign key. Goal of this pattern is to link uncorrelated data object instances of D2 to a scope instance by setting the corresponding foreign key value. The assignment is done randomly: The uncorrelated instances are taken and processed by this scope instance which is currently running. Thereby, the foreign key value is extracted by selecting the primary key value of the corresponding data object instance of D3 shown as input data. For the select statement, the JOINALL statement is used to join the table of D3 with the case object table D1 and to choose the *d3_id* value of the row with *d1_id* = $*ID*, which is related to the current scope instance. For the update, all rows of the data object table of D2 are selected which have currently a *null*-value for *d3_id* and *t1* as state value. Then, the foreign key *d3_id* is set to a concrete value and the state is set to *t2*. This is covered by the corresponding SQL statement, which is executed at the termination of the activity.

$D^{1:n}$D1 – Delete. The pattern describes a delete operation on a dependent multi instance data object. At the termination of the activity, the corresponding data object instances are deleted, whereby the instances have to be in the given state. This is covered by the SQL statement, which also considers the given state *t* in the WHERE-clause in order to avoid the deletion of wrong data object instances. For the deletion, all data table rows of D2 are selected where the foreign key value *d3_id* points to an entry in the table of D3 which is related to $*ID* determined by means of the JOINALL statement from D3 until the case object D1. This is covered by the corresponding SQL statement.

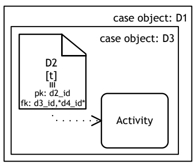
5.4 Patterns for Dependent$^{m:n}$ Objects

This section describes the patterns and their SQL queries for multi instance data objects, which represent a m:n relationship between two other data objects to which they are dependent. Additionally, these two data objects are in 1:n relationship with another object. Both are dependent to the case object and may point directly or indirectly to the case object. The data dependencies are expressed in the data model shown in Fig. 14. In the data model, the case object D1 has the following attributes required by our approach: a primary key attribute *d1_id* and a state attribute *state*. The dependent multi instance data objects D3 and D4 have besides the primary key attribute *d3_id* respectively *d4_id* and the state attribute *state* also a foreign key attribute *d3_id* respectively *d4_id*. This applies for the other dependent data objects D5, ..., Dn as well. The dependent m:n data object D2, which is in the focus of the subsequent queries, has a primary key attribute *d2_id*, a state attribute *state*, and additionally a set of two foreign key attributes *d3_id* and *d4_id*. For the following queries, we will use the JOINALL statement for joins with the case object as described in Section 5.2.

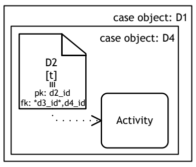
As shown in the data model, several data object instances of D3 (respectively D4) are related indirectly to one case object instance and in turn, each instance of D3 (respectively D4) relates to multiple instances of D2. Thus, several instance subsets of D2 can be observed each belonging to one instance of D3 (respectively D4). For queries on such a m:n data object, the process modeler can decide if the set of all data object instances is needed or specific subsets. We differentiate between all subsets and a specific subset by means of asterisks. If a foreign key is surrounded by these asterisks, all subsets are utilized for the query and if not, only one specific subset is utilized. We will use this notation in the following patterns.

Tab. 7: Patterns for dependent$^{m:n}$ objects.

D$^{m:n}$R1 – Read subset



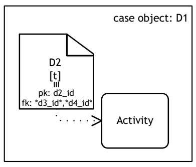
```
guard:
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL (d2, d3)
  WHERE d3.d3_id = $ID
  AND d2.state = 't') =
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL (d2, d3)
  WHERE d3.d3_id = $ID )
```



```
guard:
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL (d2, d4)
  WHERE d4.d4_id = $ID
  AND d2.state = 't') =
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL (d2, d4)
  WHERE d4.d4_id = $ID )
```

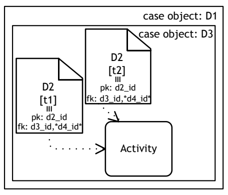
Tab. 7: Patterns for dependent$^{m:n}$ objects (ctd.).

D$^{m:n}$R2 – Read multiple subset



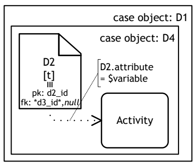
```
guard:
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL (d2,d3,..,dn,d1)
  WHERE d1.d1_id = $ID
  AND d2.state = 't') =
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL (d2, d3, ..., dn, d1)
  WHERE d1.d1_id = $ID)
```

D$^{m:n}$R3 – Read multiple states



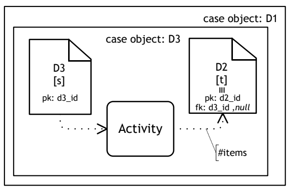
```
guard:(
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL (d2, d3)
  WHERE d3.d3_id = $ID
  AND d2.state = 't1') =
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL (d2, d3)
  WHERE d3.d3_id = $ID))
  xor (
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL (d2, d3)
  WHERE d3.d3_id = $ID
  AND d2.state = 't2') =
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL (d2, d3)
  WHERE d3.d3_id = $ID))
```

D$^{m:n}$R4 – Read without foreign key



```
guard:
  (SELECT COUNT(d2.d2_id)
  FROM JOINALL (d2, d3, ..., dn, d1)
  WHERE d1.d1_id = $ID
  AND d2.state = 't'
  AND d2.d4_id IS NULL
  AND d2.attribute = $variable) >= 1
```

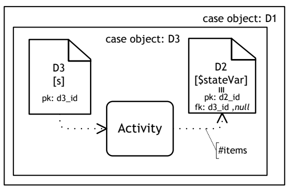
D$^{m:n}$C1 – Create single state



```
INSERT INTO d2
(d2_id, d3_id, d4_id, state) VALUES
(DEFAULT, fk,   NULL, 't')
...
(DEFAULT, fk, NULL, 't')
//#items times

fk = SELECT d3_id
FROM d3
WHERE d3_id = $ID
```

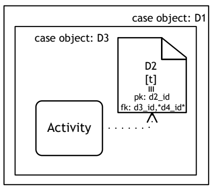
Tab. 7: Patterns for dependent$^{m:n}$ objects (ctd.).

D$^{m:n}$C2 – Create multiple states



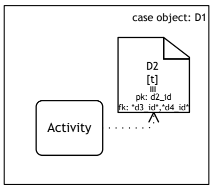
```
INSERT INTO d2
(d2_id, d3_id, d4_id, state) VALUES
(DEFAULT, fk, NULL, $stateVar)
...
(DEFAULT, fk,   NULL, $stateVar)
//#items times

fk = SELECT d3_id
FROM d3
WHERE d3_id=$ID
```

D$^{m:n}$U1 – Update subset



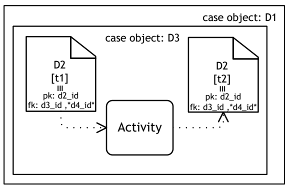
```
UPDATE d2
SET state = 't'
WHERE d3_id = (
  SELECT d3_id FROM d3
  WHERE d3_id = $ID)
```

D$^{m:n}$U2 – Update multiple subsets



```
UPDATE d2
SET state = 't'
WHERE d3_id = (
  SELECT d3.d3_id
  FROM JOINALL(d3, ..., dn, d1)
  WHERE d1.d1_id = $ID)
```

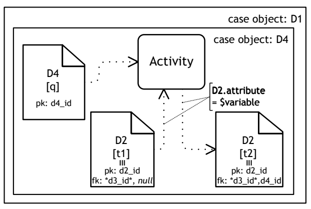
D$^{m:n}$U3 – Update with required input



```
UPDATE d2
SET state = 't2'
WHERE d3_id = (
  SELECT d3_id FROM d3
  WHERE d3_id = $ID)
AND state = 't1'
```

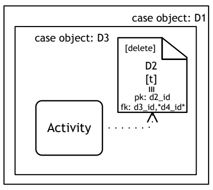
Tab. 7: Patterns for dependent$^{m:n}$ objects (ctd.).

D$^{m:n}$U4 – Update missing foreign key



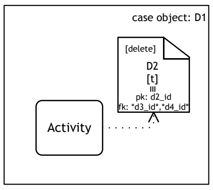
```
UPDATE d2
SET d4_id = (
  SELECT d4_id
  FROM d4
  WHERE d4_id = $ID),
 state = 't1'
WHERE d3_id = (
  SELECT d3.d3_id
  FROM JOINALL(d3, ..., dn, d1)
  WHERE d1.d1_id = $PID)
AND state = 't2'
AND d4_id IS NULL
AND attribute = $variable
```

D$^{m:n}$D1 – Delete subset



```
DELETE FROM d2
WHERE d3_id = (
  SELECT d3_id FROM d3
  WHERE d3_id = $ID)
AND state = 't'
```

D$^{m:n}$D1 – Delete multiple subsets



```
DELETE FROM d2
WHERE d3_id = (
  SELECT d3.d3_id
  FROM JOINALL(d3, ..., dn, d1))
  WHERE d1.d1_id = $ID)
AND state = 't'
```

D$^{m:n}$R1 – Read subset. This pattern describes a read operation on a specific instance subset of a dependent m:n data object. In the upper pattern, the foreign key *d3_id* is not surrounded by asterisks; this indicates that a subset is requested belonging to a particular data object instance of D3 being the case object of the surrounding scope. Read requires that the respective instance subset of the data object D2 being in state *t* is available. Using the statement JOINALL(D2, D3), we can build the join-table between m:n data object D2 and the case object D3 by means of their foreign key relation. In the join-table, each row with *d3_id* = $*ID* describes an instance of D2 that is related to the case object instance of the corresponding scope instance. This is used by the SQL statement to return all rows of the respective database table for the m:n data object D2 which are related to $*ID* and have state *t* (first select) and to return all rows which are related to $*ID* independently from the state attribute (second select). The guard ensures that the activity is only enabled if the instance subset related to $*ID* is in state *t*. The same applies if a subset of the m:n data object D2 belonging to a particular data object instance of D4 is requested as it is shown in the lower pattern.

$D^{m:n}$R2 – Read multiple subsets. This pattern describes a read operation on all instance subsets of a dependent m:n data object. Both foreign key attributes of the data object D2 are surrounded by asterisks; this indicates that no specific instance subset of D2 is requested but rather all subsets relating to the case object D1 of the surrounding scope. Read requires that the respective instance subsets of the data object D2 being in state *t* are available. Using the statement JOINALL(D2,D3,...Dn,D1), we can build the join-table between m:n data object D2 and the case object D1 by means of their foreign key relation. For the join, both foreign key relations of D2 can be used supposing that the related data object D3 as well as D4 are in turn in a direct or indirect relation with the case object D1. Thus, the SQL query compares the number of rows in table D2 related to $*ID* and being in state *t* to all rows being related to $*ID* independently from the state. As soon as both selects return the same number, the activity can be started.

$D^{m:n}$R3 – Read multiple states. The pattern describes a read operation on a specific instance subset of a dependent m:n data object similar to $D^{m:n}$R1, but it allows that the data object can be present in different states. In the pattern, all instances of the subset of D2 corresponding to a particular instance of D3 have to be available either in state *t1* or in state *t2*; a mixture of states is not allowed due to BPMN semantics [16]. This is ensured by the guard expression to the right. For the state *t1*, all rows of the data object table of D2 are selected which are related to $*ID* and have the state value *t1*. These are compared to all rows being related to $*ID* independently from the state. If both return the same number, the condition holds true. A similar check is done for other state where all rows related to $*ID* and have the state value *t2* are compared to all rows being related to $*ID*. The activity is enabled as soon as one of the conditions holds true.

$D^{m:n}$R4 – Read without foreign key. The pattern describes a read operation on a specific instance subset of a dependent m:n data object which instances have one not yet specified foreign key value. Due to the missing foreign key value, a join with case object D4 of the directly surrounding scope cannot be created. However, we have to assure that only data object instances of D2 are selected which belong to the process execution. The process is the top-level of a scope hierarchy and has in this pattern D1 as case object. We assume that data object D3, to which D2 has already an existing second foreign key relation, is directly related to the process case object. This foreign key relation to D3 is used to select the respective rows of table D2. In terms of database design, the rows of D2 are selected where the foreign key value *d3_id* points to rows in the table D3 which are in turn related to $PID – the current process instance – over their foreign key relation to the process case object. Additionally, these rows of D2 have to be in state *t* and have to have a *null*-value for the second foreign key attribute. Furthermore, the process designer can provide an expression at the input data flow edge specifying a specific set of all instances with no foreign key relation being read by the activity. This expression compares a given data object attribute with a process variable being set during process execution. If the expression is not further specified, it will not be further considered. All these aspects are

captured by the SQL query to the right. The activity is enabled as soon as one or more rows are in the result set.

$D^{m:n}$C1 – Create single state. The pattern describes a create operation on a dependent m:n data object. *Create* results in a specific subset of entries belonging to a particular instance of D3 in the data object table of D2, each entry with a default primary key value, the values for the specified foreign keys, and *t* as state value. The m:n relations presented by the data object D2 have to be set in two steps because an activity instance can only relate one instance of D3 (respectively D4) to an instance subset of D2. Therefore, a particular value is set for the foreign key attribute *d3_id* and a *null*-value for the other foreign key attribute. The non-empty foreign key value of D2 is extracted by selecting the primary key value *d3_id* from the row in the case object table D3 where *d3_id* = $*ID*. This select statement is executed at first and the returned foreign key value is saved in a variable *fk*. The variable is used by the SQL query for each insertion of a new row for D2 at the termination of the activity. The number of object instances to be created is determined by the process variable *#items*, which is attached to the output data flow edge.

$D^{m:n}$C2 – Create multiple states. The pattern describes a create operation on a dependent m:n data object similar to $D^{m:n}$C1, but the state is not statically given by the process model; it is dynamically set during activity execution by means of a process variable. Each new entry is added to the data object table of D2 with a default primary key value, the values for the specified foreign keys, and the process variable value of $*stateVar* as state value covered by the corresponding SQL query. Similar to $D^{m:n}$C1, the number of object instances to be created is determined by the process variable *#items*, which is attached to the output data flow edge.

$D^{m:n}$U1 – Update subset. The pattern describes an update operation on a specific instance subset of a dependent m:n data object. At the termination of the activity, a new state is set for the instance subset, where each object instance belongs to the same particular data object instance of D3. In terms of database design, the state value of all rows in the data object table of D2 related to the current case object instance with *d3_id* = $*ID* is updated to *t*. Alternatively, also the process variable $*stateVar* can be used in the update statement for dynamically setting the state during activity execution as done in pattern $D^{m:n}$C2.

$D^{m:n}$U2 – Update multiple subsets. The pattern describes an update operation on all instance subsets of a dependent m:n data object. Both foreign key attributes of the data object D2 are surrounded by asterisks; this indicates that no specific instance subset of D2 is requested but rather all subsets relating to the case object D1 of the surrounding scope. At the termination of the activity, a new state is set for the respective instance subsets. In terms of database design, the state value of the corresponding rows in the data object table of D2 related to $*ID* is updated to *t*. Therefore, the $*ID* is determined by means of the JOINALL statement from one of the via foreign key related data objects (either D3 or D4) with the case object D1. Alternatively, also the process variable $*stateVar* can be used in the update statement for dynamically setting the state during activity execution as done in pattern $D^{m:n}$C2.

$D^{m:n}$U3 – Update with required input. The pattern describes an update operation on a specific instance subset of a dependent m:n data object similar to pattern $D^{m:n}$U1, but it additionally requires that all instances of the subset are in the given state of the data input. The corresponding SQL statement only selects the rows related to $ID with the state value *t1* and updates it to *t2*.

$D^{m:n}$U4 – Update missing foreign key. The pattern describes an update operation on a specific instance subset of a dependent m:n data object, which instances have one not yet specified foreign key value. This pattern is the continuation of pattern $D^{m:n}$C1. Thereby, the foreign key value for this instance subset of D2 is extracted by selecting the primary key value of the corresponding data object instance of D4 as shown in the corresponding SQL statement. This pattern uses an expression at the output data flow edge for specifying which subset of all D2 instances should be assigned to one specific instance of D4. This expression compares a given data object attribute with a process variable, which is set during process execution. In the SQL statement, it is used for the update WHERE-clause. Additionally, all data object instances of D2, which have a missing foreign key, have to be selected in a similar manner as in pattern $D^{m:n}$R4 over the WHERE-clause.

$D^{m:n}$D1 – Delete subset. The pattern describes a delete operation on a specific instance subset of a dependent m:n data object. At the termination of the activity, the instance subset being related to the current case object instance with *d3_id* = $*ID* is deleted, whereby all instances of the subset have to be in the given state. This is covered by the SQL statement, which also considers the given state *t* in the WHERE-clause in order to avoid the deletion of wrong data object instances.

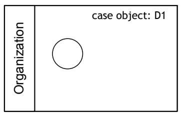
$D^{m:n}$D2 – Delete multiple subsets. The pattern describes a delete operation on all instance subsets of a dependent m:n data object. Both foreign key attributes of the data object D2 are surrounded by asterisks; this indicates that no specific instance subset of D2 is requested but rather all subsets relating to the case object D1 of the surrounding scope. At the termination of the activity, all instance subsets being related to the current case object instance with *d1_id* = $*ID* are deleted, whereby all instances have to be in the given state. This is covered by the SQL statement, which also considers the given state *t* in the WHERE-clause in order to avoid the deletion of wrong data object instances. The $*ID* is determined by means of the JOINALL statement from one of the via foreign key related data objects (either D3 or D4) with the case object D1.

5.5 Instantiation Patterns

Process and activity instantiation is an essential part of the process execution. We specify a set of four instantiation patterns to be able to link the data object instances with the process or activity instance from those they are processed. These and the corresponding SQL queries will be introduced in this section.

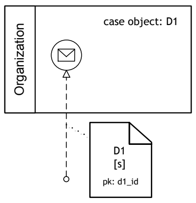
Tab. 8: Patterns for process and activity instantiation.

I1 – Process instantiation without data trigger



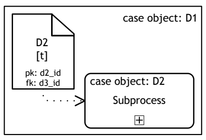
```
Start process instance
with new $ID
```

I2 – Process instantiation with data trigger



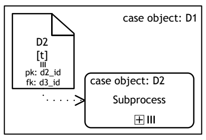
```
Start process instance
with id d2_id
```

I3 – Subprocess instantiation with single data trigger



```
For d2_id ∈ (
  SELECT d2.d2_id
  FROM JOINALL(d2, d3, ..., dn, d1)
  WHERE d1.d1_id = $ID)
start subprocess
with id d2_id
```

I4 – Subprocess instantiation with multiple data trigger



```
For each d2_id ∈ (
  SELECT d2.d2_id
  FROM JOINALL(d2, d3, ..., dn, d1)
  WHERE d1.d1_id = $ID)
start subprocess
with id d2_id
```

I1 – Process instantiation without data trigger. This pattern describes the instantiation of a process without any data trigger by an arbitrary event. The instance of the process gets an unique identifier (id), which is managed by the process engine. As soon an instance of the case object is created within the process instance, it will receive the id of its process instance as primary key value (see pattern CC1).

I2 – Process instantiation with data trigger. This pattern describes the instantiation of a process triggered by a data object, which already exists and is received by the process. At the same time, data object D1 is the case object of the process. Thus, the instantiated process instance gets the primary key value of its case object instance as id in order to correlate these two.

I3 – Subprocess instantiation with single data trigger. This pattern describes the instantiation of a subprocess triggered by its case object D2. The instantiated subprocess instance gets the primary key value of the respective case object instance as id. This is captured by the SQL query that selects the primary key value of the row of the database table for D2 being related to $*ID* of the surrounding scope with D1 as case object. Therefore, D2 and D1 are joined by using the JOINALL statement.

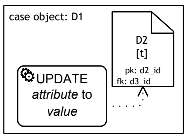
I4 – Subprocess instantiation with multiple data triggers. This pattern describes the instantiation of a multi instance subprocess triggered by its multi instance case object D2. For each data object instance of D2 related to the current scope instance, one instance of the subprocess is created, which gets the primary key value of the respective instance of D2 as id. This is captured by the SQL query that selects the primary key values for all rows of D2 being related to $*ID* of the surrounding scope with D1 as case object. Therefore, D2 and D1 are joined by using the JOINALL statement. This pattern also applies for a multi instance task having a case object (see Section 5.7 for the support of multi instance tasks).

5.6 Attribute Patterns

This section introduces pattern and the corresponding SQL queries to handle database operations on data object attributes other than the ones specified in Definition 1.

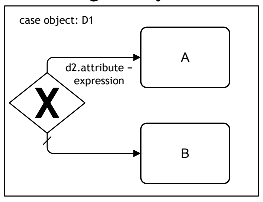
Tab. 9: Patterns for attributes other than primary key, foreign keys, and state.

A1 – Update attribute



```
UPDATE d2 SET
 attribute = 'value'
WHERE d3_id = (
   SELECT d3.d3_id
   FROM JOINALL (d3, ..., dn, d1)
   WHERE d1.d1_id = $ID  )
```

A2 – XOR gateway



```
SELECT d2.attribute
FROM JOINALL(d2, d3, ..., dn, d1)
WHERE d1.d1_id = $ID
```

A1 – Update attribute. This pattern describes the update of data object attributes other than the primary key, foreign key and state. These attributes are not represented in the BPMN data object, but are part of the data model that accompanies the process model. Thus, the corresponding attribute and the value (respectively the process variable holding the value), to which it shall be updated, is specified in the label of the task. In the graphical representation, the task is shown as service task to indicate that it is executed automatically without further human interference (after specifying the value to put into the database). Usually, this information is derived dynamically extracted from a process variable. The differentiation whether a process variable or a specific value is given in the task label needs to be done by surrounding code and included into the query accordingly. The output data object, here D2, indicates on which data object table the update statement is executed. For the update, all data table rows of D2 are selected where the foreign key value *d3_id* points to an entry in the table of D3 which is related to $*ID* determined by means of the JOINALL statement from D3 until the case object D1. This is covered by the corresponding SQL statement.

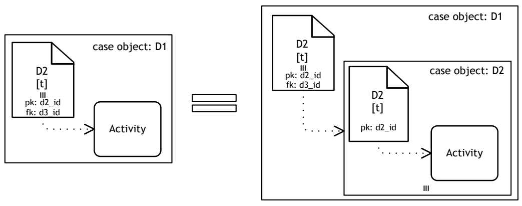
A2 – XOR gateway. This pattern describes how data object attributes can be utilized to decide the path to be taken after an exclusive choice (represented in BPMN by the XOR gateway) in the control flow. The SQL query delivers the current value of the specified attribute belonging to the data object instance of D2, which relates to $*ID* of the surrounding scope. The correlation to $*ID* is done by means of the JOINALL statement with the case object D1. The value returned by the query can be checked against the specified expression to reason about the truth value of the condition attached the upper path.

5.7 Supporting Multi Instance Tasks

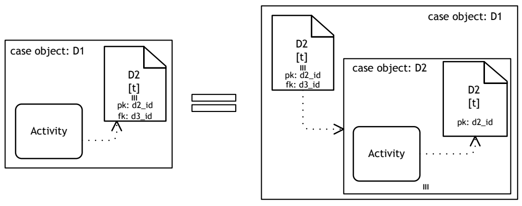
In the sections above, we specified several patterns affecting subprocesses. BPMN also offers the concept of multi instance tasks, which are very similar to subprocesses from an execution point of view: Several task instances are instantiated from which each is executed independently. To allow the handling of multi instance tasks with the introduced set of patterns, we transform each multi instance task representation into a multi instance subprocess. Thereby, we require that each multi instance task only

Tab. 10: Transforming multi instance tasks to multi instance subprocesses.

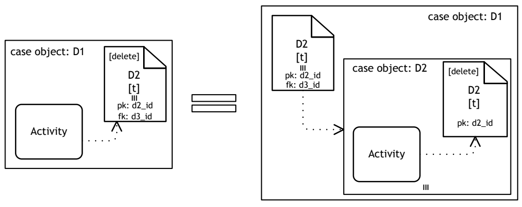
read



update



delete



contains data associations to multi instance data objects. So, multi instance tasks can not be used for the creation of dependent multi instance data objects because they need their related data object being sing-instance as input. Summarized, during the transformation, the multi instance task is mapped to a single instance task that is then surrounded by a multi instance subprocess. The associated case data object is associated as-is to the subprocess as input (to specify the number of instances to be created; see pattern I4 in Section 5.5). Additionally, all multi instance data objects are also mapped into single instance data objects and associated with the single instance activity surrounded by the multi instance subprocess; input and output properties are not changed. Details about the transformation process are given in Tab. 10.

6 Implementation

We evaluated our approach for enacting process models with complex data dependencies by implementation. In the spirit of adding only few data annotations to BPMN, we made an existing BPMN process engine data-aware by only few additions to its control structures. As basis, we chose Activiti [4], a Java-based, lightweight, and open source process engine specifically tailored for a subset of BPMN. Activiti enacts process models given in the BPMN XML format. Activiti supports standard BPMN control flow constructs. Data dependencies are not enacted from the process model, but from properties of model elements. We extended the Activiti engine with our concepts as follows.

We extended the BPMN XML specification with our concepts introduced in Section 3.1 by utilizing *extension elements* explicitly supported by BPMN [16] to add new attributes to existing constructs. Then, we supplemented the BPMN parser of Activiti so that for each activity in a process model the given sets of data object input and output can be derived.

The actual execution engine was extended at just two points: before invoking the execution of an activity to check the pre-conditions of an activity and before completing an activity to realize the post-conditions, both with respect to data objects. At either point, the engine checks and matches for all possible data input (data output) patterns described in Section 4. For each matching pre-condition pattern (i.e., read statement), the corresponding SQL select query is generated and executed on the database. If one of the pre-conditions is not fulfilled the engine suspends process flow for this activity until the condition evaluates to true. For each matching post-condition pattern, the respective SQL insert, update, or delete query is generated and executed on the database. As instantiation of processes and activities is completely handled by Activiti, we chose to not interfere with the assignment of scope ids from case objects. We chose to introduce a separate scope id variable that is set from case objects when needed and resolves to the instance id given by the engine otherwise. The current implementation uses one shared database for all processes.

After extending Activiti with these few concepts at only three points of the code base (parser, activity start, activity termination), we successfully verified each of the patterns we introduced with test processes. The extended engine, example process models, and an appropriate database are set up in a virtual machine, which is available for download together with the source code at `http://bpt.hpi.uni-potsdam.de/Public/BPMNData`.

7 Related Work

In the following, we compare the contributions of this paper to other techniques for modeling and enacting processes with data; our comparison includes all requirements for "object-aware process management" described in [10] and three additional factors.

The requirements cover modeling and enacting of *data*, *processes*, *activities*, authorization of *users*, and support for *flexible* processes. (1) Data should be managed in terms of a data model defining object types, attributes, and relations; (2) cardinality constraints should restrict relations; (3) users can only read/write data they are authorized to access; and (4) users can access data not only during process execution. Processes manage (5) the life cycle of object types and (6) the interaction of different object instances; (7) processes are only executed by authorized users and (8) users see which task they may or have to execute in the form of a task list; (9) it is possible to describe the sequencing of activities independently from the data flow. (10) One can define proper pre- and post-conditions for service activities based on objects and their attributes; (11) forms for user-interaction activities can be generated from the data dependencies; (12) activities can have a variable granularity wrt. data updates, i.e., an activity may read/write objects in 1:1, 1:n, and m:n fashion. (13) Whether a user is authorized to execute a task should depend on the role and on the authorization for the data this task accesses. (14) Flexible processes benefit from data integration in various ways (e.g., tasks that set mandatory data are scheduled when required, tasks can be re-executed, etc.).

In addition to these requirements, we consider *factors* that influence the adaption of a technique, namely, (15) whether the process paradigm is activity-centric or object-centric, (16) whether the approach is backed by standards, and (17) to which extent it can reuse existing methods and tools for modeling, execution, simulation, and analysis. Table 11 shows existing techniques satisfy these requirements and requirements (RQ1)-(RQ5) given in the introduction.

Classical activity-centric techniques such as *workflows* [1] lack a proper integration of data. Purely data-based approaches such as *active database systems* [20] allow to update data based on event-condition-action rules, but lack a genuine process perspective. Many approaches combine activity-centric process models with *object life cycles*, but are largely confined to 1:1 relationships between a process instance and the object instances it can handle, e.g., [9, 13, 22] and also BPMN [16]; some of these techniques allow flexible process execution [19].

Table 11 compares techniques that support at least a basic notion of data integration. *Proclets* [3] define object life cycles in an activity-centric way that interact through channels. In [21], process execution and object interaction are derived from a product data model. *CorePro* [14], the *Object-Process Methodology* [8], *Object-Centric Process Modeling* [18], and the *Artifact-Centric* approach [6] define processes in terms of object life cycles with various kinds of object interaction. Only artifacts support all notions of variable granularity (12), though it is given in a declarative form that cannot always be realized [7]. In *Case Handling* [2], process execution follows updating data such that

Tab. 11: Comparison of data-aware process modeling techniques.

	requirement [in [10]]	Proclets [3]	CorePro [14]	OPM [8]	Obj.-Cent. [18]	PBWS [21]	Artifacts [6]	CH [2]	BPMN [16]	PH.Fl. [10]	**this**
data	1: data integration [R1]	o	o	o	o	o	+	o	-	+	**+ (RQ2)**
	2: cardinalities [R2]	+	o	+	+	-	+	o	o	+	**+**
	3: data authorization [R10]	-	o	-	-	-	-	o	-	+	**-**
	4: data-oriented view [R8]	-	o	-	-	-	o	o	-	+	**o**
process	5: object behavior [R4]	o	+	+	+	-	o	o	o	+	**+ (RQ3)**
	6: object interactions [R5]	+	+	+	+	o	o	o	o	+	**+ (RQ4)**
	7: process authorization [R9]	+	+	+	+	+	o	+	o	+	**o**
	8: process-oriented view [R7]	+	+	+	+	+	+	+	+	+	**+**
	9: explicit sequencing of activities	+	o	o	o	-	-	-	+	o	**+**
activity	10: service calls based on data [R14]	+	+	+	+	+	+	o	o	+	**+ (RQ2)**
	11: forms based on data/flow in forms [R15/R18]	-	-	-	-	-	o/-	+/-	-	+	**-**
	12: variable granularity 1:1/1:n/m:n [R17]	-	-	-	-	-	o	o	-	o	**+ (RQ5)**
users	13: authorization by data and roles [R11/R12]	-	-	-	-	-	-	-	-	+	**-**
flex	14: flexible execution [R3/R6/R13/R16/R19]	-	o	-	-	o	o	o	-	+	**-**
factors	15: process paradigm	A	D	D	D	D	D	D	A	D	**A (RQ1)**
	16: standards	o	o	o	o	-	-	o	+	-	**+ (RQ1)**
	17: reusability of existing techniques	+	-	o	-	-	-	-	+	-	**+**

fully satisfied (+), partially satisfied (o), not satisfied (-), activity-centric (A), object-centric (D)

particular goals are reached in a flexible manner. *PHILharmonic Flows* [10] is the most advanced proposal addressing variable granularity as well as flexible process execution through a combination of *micro processes* (object life cycles) and *macro processes* (object interactions); though variable granularity is not fully supported for service tasks and each activity must be coupled to changes in a data object (limits activity sequencing). More importantly, the focus on an object-centric approach limits the reusability of existing techniques and standards for modeling, execution, and analysis.

The technique proposed in this paper extends BPMN with data integration, cardinalities can be set statically in the data model and dynamically as shown in Section 3.2; a data-oriented view is available by the use of relational databases and SQL. Object behavior and their interactions are managed with variable granularity. Our work did not focus on authorization aspects and forms, but these aspects can clearly be addressed in future work. Our approach, as it builds on BPMN, does not support flexible processes, and thus should primarily be applied in use cases requiring structured processes. Most importantly, we combine two industry standards for processes and data, allowing to leverage on various techniques for modeling and analysis. We demonstrated reusability by our implementation extending an existing engine. Thus, our approach covers more than the requirements (RQ1)-(RQ5) raised in the introduction.

8 Conclusion

In this paper, we presented an approach to model processes incorporating complex data dependencies, even m:n relationships, with classical activity-centric modeling techniques and to automatically enact them. It covers all requirements RQ1-RQ5 presented in the introduction. We combined different proven modeling techniques: the idea of object life cycles, the standard process modeling notation BPMN, and relational data modeling together make BPMN data-aware. This was achieved by introducing few extensions to BPMN data objects, e.g., an object identifier to distinguish object instances. Data objects associated to activities express pre- and post-conditions of activities. We presented a pattern-based approach to automatically derive SQL queries from depicted pre- and post-conditions. It covers all *create*, *read*, *update*, and *delete* operations by activities on different data object types so that data dependencies can be automatically executed from a given process model. Further, we ensure that no two instances of the same process have conflicting data accesses on their data objects. Through combining two standard techniques, BPMN and relational databases, we allow the opportunity to use existing methods, tools, and analysis approaches of both separately as well as combined in the new setting. The downside of this approach is an increased complexity of the process model; however, this complexity can be alleviated through appropriate tool support providing views, abstraction, and scoping.

The integration of complex data dependencies into process execution is the first of few steps towards fully automated process enactment from process models. We support operations on single data attributes beyond life cycle information and object identifiers in one step. In practice, multiple attributes are usually affected simultaneously during a data operation. Further, we assumed the usage of a shared database per process model. Multi-database support may be achieved by utilizing the concept of data stores. We focused on process orchestrations with capabilities to utilize objects created in other processes. Process choreographies with data exchange between different parties is one of the open steps. Fourth, research on formal verification is required to ensure correctness of the processes to be executed. In future work, we will address these limitations.

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language. Information Systems 30(4), 245–275 (2005)
2. van der Aalst, W.M.P., Weske, M., Grünbauer, D.: Case Handling: A New Paradigm for Business Process Support. Data & Knowledge Engineering 53(2), 129–162 (2005)
3. van der Aalst, W., Barthelmess, P., Ellis, C., Wainer, J.: Proclets: A Framework for Lightweight Interacting Workflow Processes. Int. J. Cooperative Inf. Syst. 10(4), 443–481 (2001)
4. Activiti: Activiti BPM Platform. `https://www.activiti.org/`
5. Bonitasoft: Bonita Process Engine. `https://www.bonitasoft.com/`
6. Cohn, D., Hull, R.: Business artifacts: A data-centric approach to modeling business operations and processes. IEEE Data Eng. Bull. 32(3), 3–9 (2009)
7. Damaggio, E., Hull, R., Vaculín, R.: On the equivalence of incremental and fixpoint semantics for business artifacts with guard-stage-milestone lifecycles. Inf. Syst. 38(4), 561–584 (2013)
8. Dori, D.: Object-Process Methodology. Springer (2002)
9. Eshuis, R., Van Gorp, P.: Synthesizing Object Life Cycles from Business Process Models. In: Conceptual Modeling. pp. 307–320. Springer (2012)
10. Künzle, V., Reichert, M.: PHILharmonicFlows: Towards a Framework for Object-aware Process Management. J SOFTW MAINT EVOL-R 23(4), 205–244 (2011)
11. Küster, J., Ryndina, K., Gall, H.: Generation of Business Process Models for Object Life Cycle Compliance. In: Business Process Management. pp. 165–181. Springer (2007)

12. Lanz, A., Reichert, M., Dadam, P.: Robust and flexible error handling in the aristaflow bpm suite. In: CAiSE Forum 2010. LNBIP, vol. 72, pp. 174–189. Springer (2011)
13. Liu, R., Wu, F.Y., Kumaran, S.: Transforming activity-centric business process models into information-centric models for soa solutions. J. Database Manag. 21(4), 14–34 (2010)
14. Müller, D., Reichert, M., Herbst, J.: Data-driven modeling and coordination of large process structures. In: OTM 2007. LNCS, vol. 4803, pp. 131–149. Springer (2007)
15. Nigam, A., Caswell, N.: Business artifacts: An Approach to Operational Specification. IBM Systems Journal 42(3), 428–445 (2003)
16. OMG: Business Process Model and Notation (BPMN), Version 2.0 (2011)
17. OMG: Unified Modeling Language (UML), Version 2.4.1 (2011)
18. Redding, G., Dumas, M., ter Hofstede, A.H.M., Iordachescu, A.: A flexible, object-centric approach for business process modelling. SOCA'10 4(3), 191–201 (2010)
19. Reichert, M., Rinderle-Ma, S., Dadam, P.: Flexibility in process-aware information systems. ToPNoC 5460, 115–135 (2009)
20. Silberschatz, A., Korth, H.F., Sudarshan, S.: Database System Concepts, 4th Edition. McGraw-Hill Book Company (2001)
21. Vanderfeesten, I.T.P., Reijers, H.A., van der Aalst, W.M.P.: Product-based workflow support. Inf. Syst. 36(2), 517–535 (2011)
22. Wang, J., Kumar, A.: A Framework for Document-Driven Workflow Systems. In: Business Process Management. pp. 285–301. Springer (2005)
23. Wang, Z., ter Hofstede, A.H.M., Ouyang, C., Wynn, M., Wang, J., Zhu, X.: How to Guarantee Compliance between Workflows and Product Lifecycles? Tech. rep., BPM Center Report BPM-11-10 (2011)

ISBN 978-3-86956-245-2
ISSN 1613-5652