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 paper, 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.

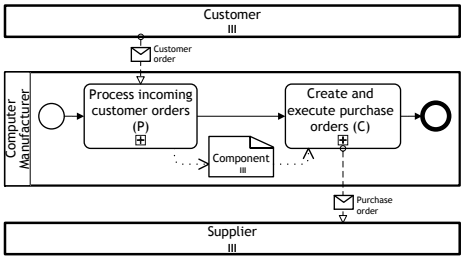


Fig. 1. Build-to-order process, where subprocess P collects multiple orders from several Customers in an internal loop and where C sends multiple Purchase orders to several Suppliers using a multi instance subprocess internally.

Widely accepted process modeling languages such as BPMN [17] 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, 24] 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, 15]: 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 [15]. 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 [17] with relational data modeling as known from relational databases [21]. 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 [17]. 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 and our implementation before we review related work in Section 5 and conclude in Section 6.

2 Data Modeling in BPMN

BPMN [17], 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 [17], 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*.

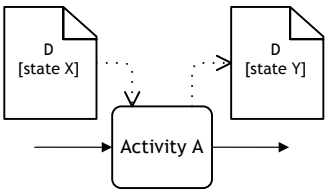


Fig. 2. Object life cycle of data object D with two representations.

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 [21]. 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 [21]).

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 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

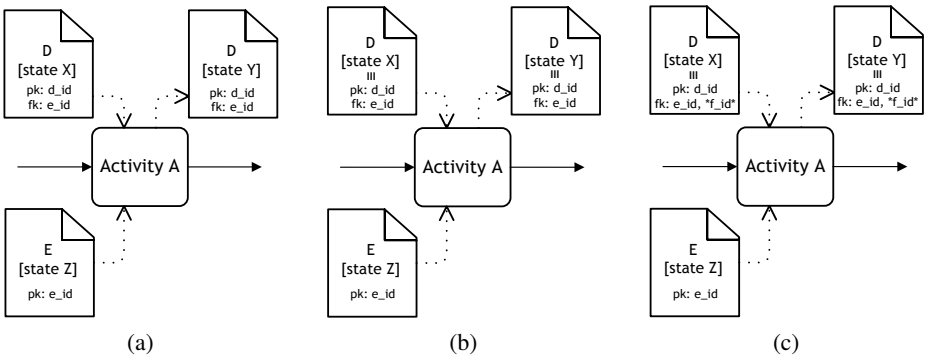


Fig. 3. Describing object interactions in (a) 1:1, (b) 1:n, and (c) m:n cardinality.

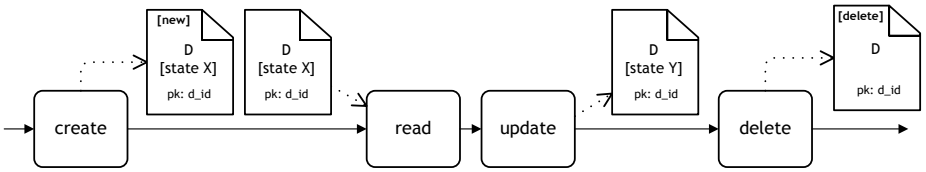


Fig. 4. Describing create, read, update, and delete of a data object.

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 [18], 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 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.

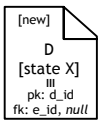


Fig. 5. Extended data object representation.

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. $\diamond$

$\bot$ 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 [16] 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 this paper, we assume that all non case objects are *directly* related to the case object; see our technical report [14] for the general case. We make data objects and case objects part of the process model as follows, utilizing a subset of BPMN [17].

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. $\diamond$

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.

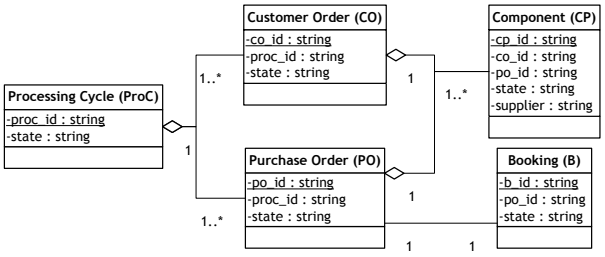


Fig. 6. Data model.

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 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

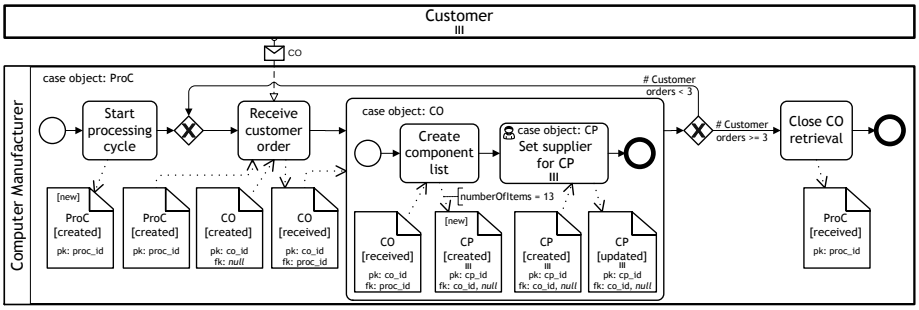


Fig. 7. Build-to-order scenario: customer order collection.

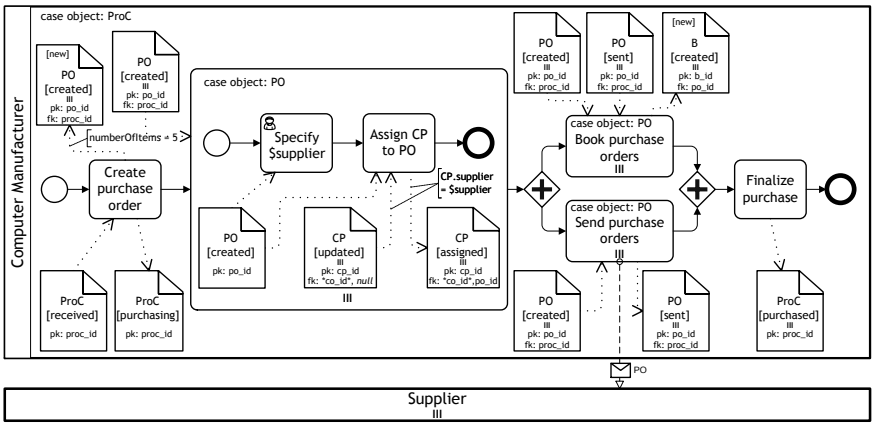


Fig. 8. Build-to-order scenario: purchase order arrangement.

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.

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.

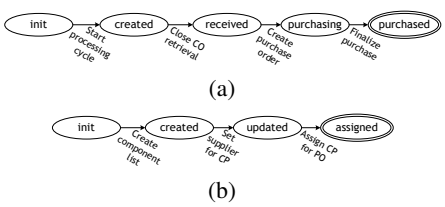


Fig. 9. Object life cycles of objects (a) *ProC* and (b) *CP* derived from the process model.

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 [17, 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 $(C', \mathcal{D}) \rightarrow (C'', \mathcal{D}')$, 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 [14]. 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; Tables 1 and 2 list the patterns and their formalization that we explain next. All 43 patterns and their formalization are given in our technical report [14].

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.

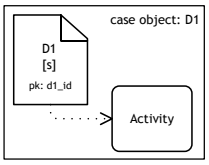
Read single object instance. Pattern 1 describes a read operation on a single data object D1 that is also the case object of the scope. The activity is only enabled when this case object is in the given state s. The guard shown below P1 in Table 1 operationalizes this behavior: it is *true* iff table 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 Table 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 [14] 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 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

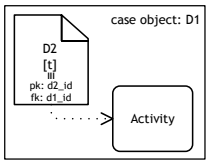
Table 1. SQL queries for patterns 1 to 3 for subprocess in Fig. 8.

P1



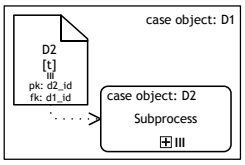

```
guard:
  (SELECT COUNT(d1.d1_id)
  FROM d1
  WHERE d1.d1_id = $ID
  AND d1.state='s') ≥ 1
```

P2




```
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)
```

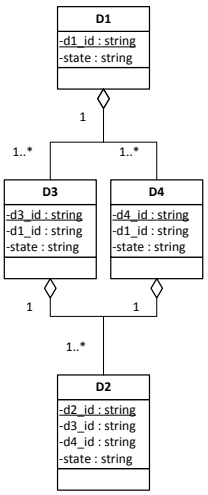
P3



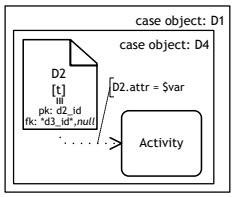

```
For each d2_id ∈ (
  SELECT d2.d2_id
  FROM d2
  WHERE d2.d1_id = $ID)
start subprocess
with id d2_id
```

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

Data model

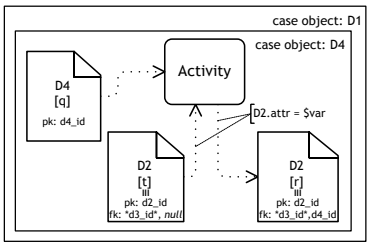


P4




```
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
```

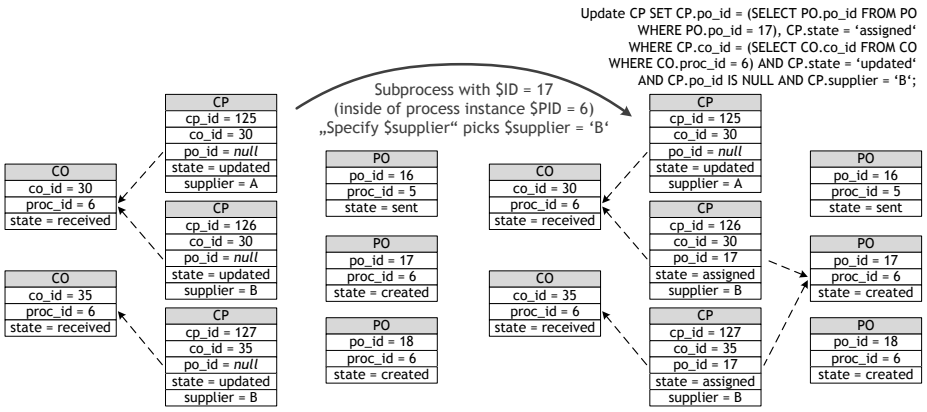
P5




```
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)
```

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 Table 1. For our example, where 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



(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.

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 Table 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 [14] for the general case. The guard shown below P4 in Table 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 Table 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.

4.3 Implementation

We evaluated our approach for enacting process models with complex data dependencies by implementation. In the spirit of building on existing standards and techniques, we made the existing BPMN process engine Activiti [4] data-aware by only few additions to its control structures. Activiti enacts process models given in the BPMN XML format and supports standard BPMN control flow constructs.

We extended the BPMN XML specification with our concepts introduced in Section 3.1 using *extension elements* explicitly supported by BPMN [17]. The BPMN parser of Activiti was supplemented correspondingly. Finally, we adapted the actual execution engine to check for availability of input data objects when an activity is enabled, and to make output data objects available when an activity completes – both through the SQL queries presented in Section 4.2. Our extensions required just over 1000 lines of code with around 600 lines being concerned with classifying data objects to patterns and generating and executing SQL queries; see [14] for details.

With the given implementation, a user can model data annotated processes in BPMN, and directly deploy the model to the extended Activiti engine, which then executes the process *including* all data dependencies. No further hard-coding is required as all information is derived from the process model. The extended engine, a graphical modeling tool, examples, and a complete setup in a virtual machine are available for download together with the source code and a screencast at `http://bpt.hpi.uni-potsdam.de/Public/BPMNData`.

5 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

Table 3. Comparison of data-aware process modeling techniques.

	requirement [in [10]]	Proclets [3]	CorePro [15]	OPM [8]	Obj.-Cent. [19]	PBWS [22]	Artifacts [6]	CH [2]	BPMN [17]	PH.FI. [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)

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 3 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* [21] 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, 23] and also BPMN [17]; some of these techniques allow flexible process execution [20].

Table 3 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 [22], process execution and object interaction are derived from a product data model. *CorePro* [15], the *Object-Process Methodology* [8], *Object-Centric Process Modeling* [19], 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 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.

6 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. Meyer, A., Pufahl, L., Fahland, D., Weske, M.: Modeling and Enacting Complex Data Dependencies in Business Processes. Tech. Rep. 74, HPI at the University of Potsdam (2013)
15. 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)
16. Nigam, A., Caswell, N.: Business artifacts: An Approach to Operational Specification. IBM Systems Journal 42(3), 428–445 (2003)
17. OMG: Business Process Model and Notation (BPMN), Version 2.0 (2011)
18. OMG: Unified Modeling Language (UML), Version 2.4.1 (2011)
19. 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)
20. Reichert, M., Rinderle-Ma, S., Dadam, P.: Flexibility in process-aware information systems. ToPNoC 5460, 115–135 (2009)
21. Silberschatz, A., Korth, H.F., Sudarshan, S.: Database System Concepts, 4th Edition. McGraw-Hill Book Company (2001)
22. Vanderfeesten, I.T.P., Reijers, H.A., van der Aalst, W.M.P.: Product-based workflow support. Inf. Syst. 36(2), 517–535 (2011)
23. Wang, J., Kumar, A.: A Framework for Document-Driven Workflow Systems. In: Business Process Management. pp. 285–301. Springer (2005)
24. 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)