PQL - A Descriptive Language for Querying, Abstracting and Changing Process Models

Klaus Kammerer, Jens Kolb, and Manfred Reichert

Institute of Databases and Information Systems
Ulm University, Germany
{klaus.kammerer,jens.kolb,manfred.reichert}@uni-ulm.de
http://www.uni-ulm.de/dbis

Abstract. The increasing adoption of process-aware information systems (PAISs) has resulted in large process repositories comprising large and complex process models. To enable context-specific perspectives on these process models and related data, a PAIS should provide techniques for the flexible creation and change of process model abstractions. However, existing approaches focus on the formal model transformations required in this context rather than on techniques for querying, abstracting and changing the process models in process repositories. This paper presents a domain-specific language for querying process models, describing abstractions on them, and defining process model changes in a generic way. Due to the generic approach taken, the definition of process model abstractions and changes on any graph-based process notation becomes possible. Overall, the presented language provides a key component for process model repositories.

1 Introduction

Process-aware information systems (PAISs) provide support for business processes at the operational level. In particular, a PAIS separates process logic from application code relying on explicit *process models*. This enables a separation of concerns, which is a well-established principle in computer science to increase maintainability and to reduce costs of change [1]. The increasing adoption of PAISs has resulted in large process model collections. Thereby, a process model may comprise dozens or hundreds of activities [2]. Furthermore, process models may refer to business objects, organizational units, user roles and other resources. Due to this high complexity, the various user groups need customized views on the processes [3]. For example, managers rather prefer an abstract process overview, whereas process participants need a detailed view of the process parts they are involved in.

Several approaches for creating process model abstractions have been proposed in literature [4, 5, 6]. However, current proposals focus on fundamental abstraction concepts for aggregating or reducing process model elements to derive a context-specific process view. Existing approaches neither provide concepts to specify process model abstractions independent from a particular process model (e.g., a particular user may be involved in several processes) nor to define them in a more descriptive way. Accordingly, for each relevant process model, users must create respective abstractions manually. In particular, the operations for abstracting a process model need to be specifically defined referring to the elements of this model; i.e., abstractions must be specified separately for each individual process model, which causes high efforts when being confronted with large process model collections (cf. Figure 1a). A possibility to lower efforts is to reduce the number of operations required to abstract process models, i.e., elementary operations may be composed to high-level ones [5]. However, the application of respective operations is still specific to a particular process model.

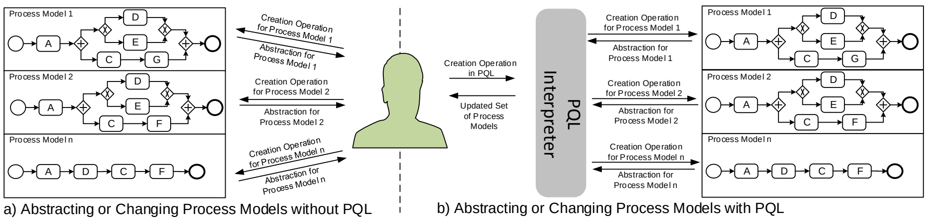

Process Model 1
Process Model 2
Process Model n
Creation Operation for Process Model 1
Abstraction for Process Model 1
Creation Operation for Process Model 2
Abstraction for Process Model 2
Creation Operation for Process Model n
Abstraction for Process Model n
Creation Operation in PQL
Updated Set of Process Models
PQL Interpreter
a) Abstracting or Changing Process Models without PQL
b) Abstracting or Changing Process Models with PQL

Fig. 1. Using the Process Query Language to Abstract or Change Process Models

In existing approaches, process changes refer to specific elements (e.g., nodes) of a process model rather than on generic process properties (e.g., process attributes). For example, it is usually not possible to replace a specific user role by another one in all process models stored in the repository [2], i.e., the change must be manually applied to each process model.

In other domains (e.g., database management), the use of domain-specific languages is common when facing large data sets. For example, *SQL* has been used to create, access and update data in relational databases [7]. However, to the best of our knowledge, no comparable approach exists for large process model repositories. To remedy this drawback, this paper introduces the *Process Query Language (PQL)*. PQL is a domain-specific language that allows defining process model abstractions in a declarative way as well as specifying changes on collections of (abstracted) process models from a process repository (cf. Figure 1b). In particular, such a declarative definition of a process can be automatically applied to multiple process models if required. Furthermore, users may use PQL to define personalized process views on their process, e.g., by abstracting process information not relevant for them. Additionally, process model collections can be easily changed based on such declarative descriptions. For example, same or similar process elements used in multiple models may be changed concurrently based on one PQL change description, e.g., if activities related to quality assurance shall be changed in all variants of a business process.

Section 2 introduces fundamentals on abstracting process models. Section 3 presents PQL and its syntax. Section 4 presents a proof-of-concept implementation. Section 5 discusses related work and Section 6 summarizes the paper.

2 Fundamentals on Process Model Abstractions

Section 2.1 defines basic notions. Section 2.2 then discusses how *process model abstractions* can be created and formally represented. It further describes how elementary operations can be composed to define high-level operations for abstracting process models.

2.1 Process Model

Basically, a process model comprises *process elements*, i.e., *process nodes* as well as the *control flow* between them (cf. Figure 2). The modeling of the latter is based on *gateways* and *control flow edges* (cf. Definition 1). Note that the data perspective is excluded in this paper to set a focus.

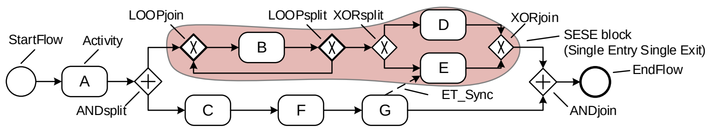


Fig. 2. Example of a Process Model

Definition 1 (Process Model). *A process model is defined as a tuple* $P = (N, NT, CE, EC, ET, attr, val)$ *where:*

- N *is a set of process nodes (i.e., activities, gateways, and start/end nodes).*
- $NT : N \rightarrow NodeType$ *with* $NodeType = \{StartFlow, EndFlow, Activity, ANDsplit, ANDjoin, XORsplit, XORjoin, LOOPsplit, LOOPjoin\}$ *is a function with* $NT(n)$ *returning the type of node* $n \in N$*.*
- $CE \subset N \times N$ *is a set of precedence relations (i.e., control edges):* $e = (n_{src}, n_{dest}) \in CE$ *with* $n_{src}, n_{dest} \in N \wedge n_{src} \neq n_{dest}$*.*
- $EC : CE \rightarrow Conds \cup \{\text{TRUE}\}$ *assigns to each control edge either a branching condition or* $TRUE$ *(i.e., the branching condition of the respective control edge always evaluates to true).*
- $ET : CE \rightarrow EdgeType$ *with* $EdgeType = \{ET\_Control, ET\_Sync, ET\_Loop\}$*.* $ET(e)$ *assigns a type to each control edge* $e \in CE$*.*
- $attr : N \,\dot\cup\, CE \rightarrow \mathcal{AS}$ *assigns to each process element a corresponding attribute set* $AS \subseteq \mathcal{AS}$*.*
- $val : (N \,\dot\cup\, CE) \times \mathcal{AS} \rightarrow valueDomain(\mathcal{AS})$ *assigns to any attribute* $x \in \mathcal{AS}$ *of a process element* $pe \in N \,\dot\cup\, CE$ *its value:*

$$val(pe, x) = \begin{cases} value(x)^1, & x \in attr(pe) \\ null^2, & x \notin attr(pe) \end{cases}$$

[1] $value(x)$ denotes the value of process attribute x
[2] attribute is undefined for the respective process element

Definition 1 can be used for representing the schemes of both process models and related process model abstractions. In particular, Definition 1 is not restricted to a specific activity-oriented modeling languages, but may be applied in the context of arbitrary graph-based process modeling languages. This paper uses a subset of BPMN elements as modeling notation. We further assume that a process model is *well-structured*, i.e., sequences, branchings (of different semantics) and loops are specified as blocks with well-defined start and end nodes having the same gateway type. These blocks—also known as SESE blocks (cf. Definition 2)—may be arbitrarily nested, but must not overlap (like blocks in BPEL). To increase expressiveness, *synchronization edges* allow for a *cross-block* synchronization of parallel activities (like BPEL links). In Figure 2, for example, activity E must not be enabled before G is completed. Additionally, process elements have associated attributes. For example, an activity has attributes like *ID*, *name* or *assignedUser* (cf. Figure 3).

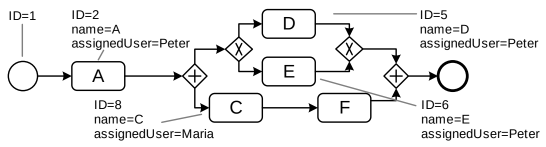


Fig. 3. Example of a Process Model with Attributes

Definition 2 (SESE). *Let* $P = (N, NT, CE, EC, ET, attr, val)$ *be a process model and* $X \subseteq N$ *be a subset of activities (i.e.,* $NT(n) = Activity\ \forall n \in X$*). Then: Subgraph* P' *induced by* X *is called* SESE *(Single Entry Single Exit) block iff* P' *is connected and has exactly one incoming and one outgoing edge connecting it with* P*. Further, let* $(n_s, n_e) \equiv MinimalSESE(P, X)$ *denote the start and end node of the minimum SESE comprising all activities from* $X \subseteq N$*.*

How to determine SESE blocks is described in [8]. Since we presume a well-structured process model, a minimum SESE can be always determined.

2.2 Process Model Abstractions

In order to abstract a given process model, the schema of the latter needs to be simplified. For this purpose, *elementary operations* are provided (cf. Table 1) that may be further combined to realize *high-level abstraction operations* (e.g., show all activities a particular actor is involved in and their precedence relations) [9, 5]. At the elementary level, two categories of operations are provided: *reduction* and *aggregation*. An elementary *reduction* operation hides an activity of a process model. For example, $RedActivity(P, n)$ removes activity n and its incoming and outgoing edges and re-inserts a new edge linking the predecessor of n with its successor in process model P (cf. Figure 4a). An *aggregation* operation, in turn, takes a set of activities as input and combines them to an abstracted node. For example, $AggrSESE(P, N')$ removes all nodes of the

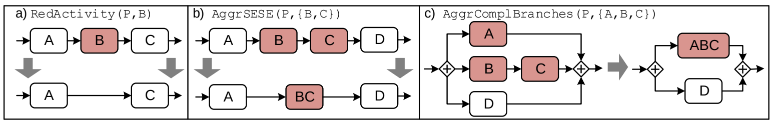


Fig. 4. Examples of Elementary Abstraction Operations

SESE block induced by node set N' and re-inserts an abstract activity instead (cf. Figure 4b). Furthermore, operation $AggrComplBranches(P, N')$ aggregates multiple branches of an XOR/AND branching to a single branch with one abstracted node (cf. Figure 4c). An abstraction of a process model can be created through the consecutive application of elementary operations on a given process model (cf. Definition 3) [5, 10]. Note that there exist other elementary operations, which address process perspectives other than control flow as well (e.g., data flow); we omit details here.

Definition 3 (Process Model Abstraction). *Let P be a process model. A process model abstraction V(P) is described through a creation set* $CS_{V(P)} = (P, Op)$ *with*

- $P = (N, NT, CE, EC, ET, attr, val)$ *being the original process model,*
- $Op = \langle Op_1, \ldots, Op_n \rangle$ *being a sequence of elementary operations applied to P:* $Op_i \subseteq \mathcal{O} \equiv \{RedActivity, AggrSESE, AggrComplBranches\}$

A node n of the abstracted process model $V(P)$ either directly corresponds to a node $n \in N$ of the original process model P or it abstracts a set of nodes from P. $PMNode(V(P), n)$ reflects this correspondence by returning either n or node set N_n aggregated in V(P), depending on creation set $CS_{V(P)}$.

$$PMNode(V(P), n) = \begin{cases} n & n \in N \\ N_n & \exists Op_i \in Op : N_n \xrightarrow{Op_i} n \end{cases}$$

Operation	Description
$RedActivity(P, n)$	Activity n and its incoming and outgoing edges are removed in P, and a new edge linking the predecessor of n with its successor is inserted (cf. Figure 4a).
$AggrSESE(P, N')$	All nodes of the SESE block defined by N' are removed in P and an abstract activity is re-inserted instead (cf. Figure 4b).
$AggrComplBranches(P, N')$	Complete branches of an XOR/AND branching are aggregated to a branch with one abstracted node in P. N' must contain the activities of the branches (i.e., activities between split and corresponding join gateway) that shall be replaced by a single branch consisting of one aggregated node (cf. Figure 4c).

Table 1. Examples of Elementary Abstraction Operations

When abstracting a process model, unnecessarily complex control flow structures could result due to the generic nature of the operations applied. For example, single branches of a parallel branching might become "empty" or a parallel branching might only have one branch left after applying reductions. In such cases, unnecessary gateways should be removed to obtain a more comprehensible schema of the abstracted model. Therefore, refactoring operations are provided. In particular, this does not affect the control flow dependencies of activities and, hence, does not change behavioral semantics of the refactored model [2].

To abstract multiple aspects of a process model several elementary operations may be applied in combination [5]. For example, *AggrSESE* and *AggrComplBranch* may be combined to high-level operation *AggregateControlFlow* (cf. Figure 5). Obviously, abstracting large process models becomes easier, when providing high-level operations in addition to elementary ones. In particular, the selection of the nodes to be abstracted should be more convenient. Current abstraction approaches, however, require the explicit specification of these nodes in relation to a particular process model. A declarative definition of these nodes (i.e., select all activities, a user is involved in), therefore, would enable users to abstract nodes in a more convenient and flexible way.

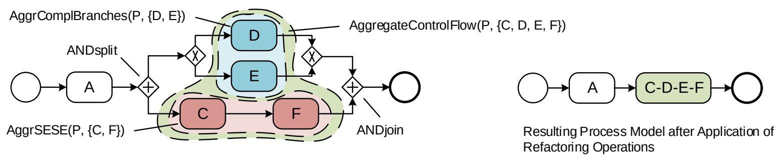

AggrComplBranches(P, {D, E})
ANDsplit
AggregateControlFlow(P, {C, D, E, F})
A
D
E
C
F
AggrSESE(P, {C, F})
ANDjoin
A
C-D-E-F
Resulting Process Model after Application of Refactoring Operations

Fig. 5. Composition of Elementary Operations

3 The PQL Language

The presented operations for abstracting process models (i.e., process views) always refer to a process model they shall be applied to. Thus, their effects cannot be described independently from a particular process model, which causes high efforts in case an abstraction shall be introduced to multiple process models. To remedy this drawback, we introduce *Process Query Language (PQL)* that allows specifying process abstractions independent from a specific process model. PQL allows defining changes on a selected collection of process models as well.

3.1 Overview

PQL allows describing process model abstractions as well as process model changes in a declarative way. More precisely, respective descriptions may not only be applied to a single process model, but to a collection of selected process models as well. In the following, we denote such a declarative description of abstractions or changes on a collection of process models as *PQL request.* In general, a PQL request consists of two parts: First, *selection section* defines the process models concerned by the PQL request; Second, *modification section* defines the abstractions and changes respectively to be applied to the selected process models.

Figure 6 illustrates how a PQL request is processed: First, a user sends a *PQL request* to the *PQL interpreter* (Step ①). Then, those process models are selected from the process repository that match the predicates specified in selection section of the PQL request (Step ②). If applicable, changes of the modification section of the PQL request are applied to the selected process models (Step ③). Following this, the abstractions defined in the modification section are applied to the selected process models (Step ④). Finally, the selected, changed and abstracted process models are presented to the user (Step ⑤). Note that Steps 3+4 are optional depending on the modification section of the PQL request.

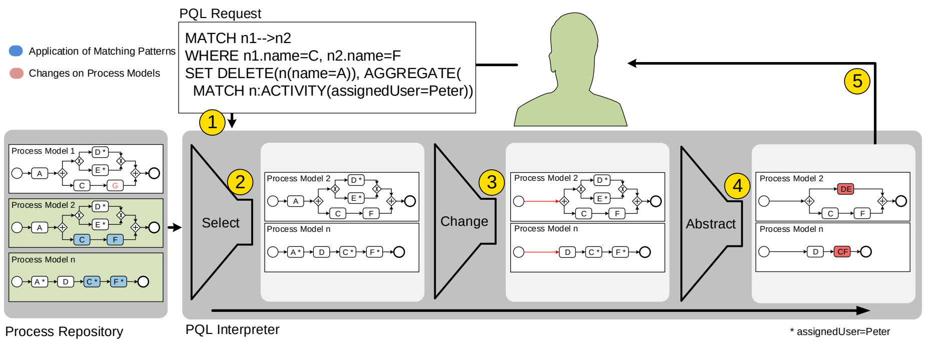


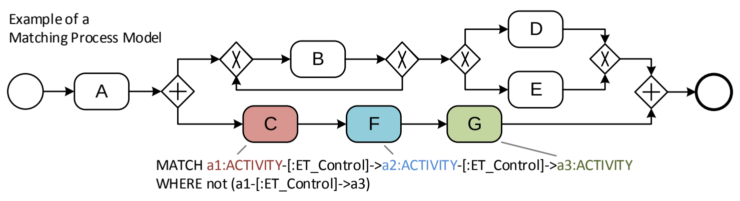
Fig. 6. Processing a PQL Request

The specification of PQL requests relies on the *Cypher Query Language*, which we *adopt* to meet the specific requirements of process modeling [11]. Cypher is a declarative graph query language known from the Neo4J graph database. In particular, it allows querying and changing the graphs from a graph database [12]. Furthermore, Cypher has been designed with the goal to be efficient, expressive and human-readable. Thus, it is well suited as basis for PQL. An example of a PQL request expressed with Cypher is shown in Listing 1.

```
1 MATCH   a1:ACTIVITY-[:ET_Control]->a2:ACTIVITY-[:ET_Control]->a3:ACTIVITY
2 WHERE   not (a1-[:ET_Control]->a3)
3 RETURN  a3
```

Listing 1. Example of a PQL Request

In the PQL request from Listing 1, Line 1 refers to process model that contain a path (i.e., a sequence of edges with type *ET_Control*) linking activities $a1$, $a2$ and $a3$. Note that $a1, a2$ and $a3$ constitute variables. To be more precise, the PQL request searches for process models comprising any sequence consisting of three activities. As a constraint (cf. Line 2), only directly adjacent nodes of $a2$ are returned. Additionally, the nodes must not be directly adjacent to $a1$. An application on the process models depicted in Figure 2 returns process activities C, F and G as the only possible match (cf. Figure 7).
Listing 2 presents the general syntax of a PQL request in BNF grammar notation [13]. Other relevant PQL syntax elements will be introduced step-by-step.



```
MATCH a1:ACTIVITY-[:ET_Control]->a2:ACTIVITY-[:ET_Control]->a3:ACTIVITY
WHERE not (a1-[:ET_Control]->a3)
```

Fig. 7. PQL Request Determining a Sequence of Three Activities

```
PQLrequest ::= match where? set?
```

Listing 2. BNF for a PQL Request

3.2 Selecting Process Models and Process Elements

PQL allows for a predicate-based selection of process models and process elements respectively. First, a search predicate may describe structural properties of the process models to be queried, e.g., to select all process models comprising two sequential nodes $n1$ and $n2$ (cf. Step ② in Figure 6). Second, process models and process elements can be selected by a predicate-based search on specific process element attributes. The latter are usually defined for each process model in a process repository, e.g., a user role designated to execute certain activities [5]. A *predicate* serves to assign properties (i.e., attributes) to process elements. For example, for predicate $PR \equiv \{x | x < 4, x \in \mathbb{N}\}$, we obtain $x \in \{1, 2, 3\}$. In general, a predicate PR may be described as boolean-valued function:

Definition 4 (Predicate). *Let $P = (N, NT, CE, EC, ET, attr, val)$ be a process model, $x \in N \dot\cup CE$ be a process element, and PR be a predicate. Then:*

$$PR(x) : x \longrightarrow \{true, false\}$$

A predicate is used to compare attributes of process models and process elements respectively. In this context, *ordering functions* for numerical values (i.e., $\neq, <, \leq, =, \geq, >$) may be used. For example, string values may be compared on either equality against fixed values or based on edit distance to determine their similarity [14]. Two or more predicates may be concatenated using Boolean operations (i.e., AND, OR, NOT).

As aforementioned, PQL offers *structural* as well as *attributional matching patterns* to determine whether a specific process fragment is present in a particular process model. More precisely, *structural matching patterns* consider the control flow of a process model; i.e., they define the process fragments that need to be present in selected process models. In turn, *attributional matching patterns* allow selecting process models and process elements, respectively, based on process element attributes.

Structural matching patterns define constraints on process fragments to be matched against existing process models in a process repository. In PQL, structural matching patterns are initiated by a `MATCH` keyword (cf. Line 1, Listing 3) followed by a respective matching pattern, which describes the respective process fragment (cf. Line 2).

```
 1 match        ::= "MATCH" match_pat ((","  match_pat)+)?
 2 match_pat    ::= (PQL_PATHID "=")? (MATCH_FUNCTION "(" path ")" | path)
 3
 4 path         ::= node ((edge) node)+)?
 5
 6 node         ::= PQL_NODEID (":" NODETYPE)? ("(" NODEID ")")?
 7
 8 edge         ::= cond_edge | uncond_edge
 9 uncond_edge  ::= ("--" | "-->")
10 cond_edge    ::= (("-" edge_attrib "-") | ("-" edge_attrib "->"))
11 edge_attrib  ::= "[" PQL_EDGEID? (":"
12                      ((EDGETYPE ("|" EDGETYPE)* )? | edge_quant)?
13                  "]"
14 edge_quant   ::= "*" (EXACT_QUANTIFIER |
15                      (MIN_QUANTIFIER ".." MAX_QUANTIFIER)?)?
```

Listing 3. BNF for Structural Matching in a PQL Request

Structural matching patterns are further categorized into *dedicated* and *abstract* patterns. While *dedicated* patterns (cf. Lines 8-11 in Listing 3) are able to describe SESE blocks of a process model, an *abstract* pattern (cf. Lines 12-15) offers an additional edge attribute. The latter defines control flow adjacencies between nodes, i.e., the proximity of a pair of nodes. For example, to specify the selection of all succeeding nodes of activity A in Figure 3 requires abstract structural patterns. Table 2 summarizes basic PQL structural matching patterns.

Pattern	**Description**	**Type**
`MATCH a-->b`	Pattern describing the existence of an edge of any type between nodes a and b.	dedicated
`MATCH a(2)-[:EDGE_TYPE]->b`	Pattern describing a process fragment whose nodes a and b are connected by an edge with type `EDGE_TYPE`; furthermore, a has attribute ID with value 2	dedicated
`MATCH a-[*1..5]->b`	Pattern describing a process fragment with nodes a and b that do not directly succeed, but are separated by at least one and at most five nodes.	abstract
`MATCH a-[*]->b`	Pattern describing an arbitrary number of nodes between nodes a and b.	abstract
`MATCH p = shortestPath( a-[:ET_Control*3]->c)`	Pattern describing a minimum SESE block with a maximum of three control edges between nodes a and c.	abstract

Table 2. Examples of Structural Matching Patterns

Attributional matching patterns allow for an additional filtering of process fragments selected through a structural matching. For this purpose, predicates referring to process element attributes may be defined (cf. Listing 4). Attributional matching is initiated by a `WHERE` keyword, which may follow a `MATCH` keyword (cf. Table 3). Note that attributional matching patterns refer to process elements pre-selected through a structural matching pattern. For example, nodes a and b selected by pattern `MATCH a-->b` can be further filtered with attributional matching patterns. If the attributional matching $a.ID = 5$ shall be applied to all activities of a process model, the `MATCH` keyword needs to be defined as follows: `MATCH a:ACTIVITY(*) WHERE a.ID=5`.

```
where            ::= "WHERE" predicate ((BOOL_OPERATOR predicate)+)?
predicate        ::= comparison_pred | regex_pred

comparison_pred  ::= PROPERTY_ID COMPARISON_OPERATOR any_val
regex_pred       ::= PROPERTY_ID "=~" REGEX_EXPRESSION
```

Listing 4. BNF for Attributional Matching in a PQL Request

Attributional matching patterns may be combined with structural ones. For example, PQL request `MATCH a:ACTIVITY-->b` matches with node attribute $NodeType = ACTIVITY$ for node a and any sequence of nodes a and b.

Pattern	Description
`MATCH (a) WHERE (a.NAME="Sell Item")`	Select all nodes with name "Sell Item".
`MATCH (a) WHERE HAS (a.attrib)`	Select all nodes for which an attribute with name *attrib* is present.
`MATCH (a) WHERE a:ACTIVITY`	Select all nodes with node type $ACTIVITY$.
`MATCH (a-[*]->b) WHERE a:ACTIVITY(1), b(2)`	Select (1) activity a with ID=1 and node type $ACTIVITY$ and (2) node b with ID=2.

Table 3. Examples of Attributional Matching Patterns

Figure 8 illustrates the application of a matching pattern to the process model from Figure 3. Figure 8a matches a sequence of nodes a, b and c, with node a having attribute $ID = 2$, node b being an arbitrary node, and node c being of type $ACTIVITY$. Figure 8b matches for a node a with $a.ID = 8$ and arbitrary nodes succeeding a, having a maximum distance of 2 to a (i.e., nodes G and $ANDjoin$ match in the process model). In turn, Figure 8c matches all nodes having assigned attribute $assignedUser$. Finally, Figure 8d matches nodes whose ID either is 2 or 5 (i.e., nodes A and D match in the process model).

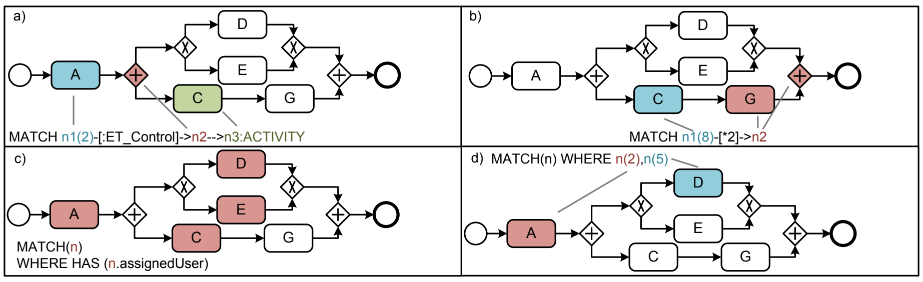

a)
A
D
E
C
G
MATCH n1(2)-[:ET_Control]->n2-->n3:ACTIVITY
b)
A
D
E
C
G
MATCH n1(8)-[*2]->n2
c)
A
D
E
C
G
MATCH(n)
WHERE HAS (n.assignedUser)
d) MATCH(n) WHERE n(2),n(5)
A
D
E
C
G

Fig. 8. Overview on PQL Matching Patterns

3.3 Abstracting Process Models

This section shows how to define abstractions on the process models referenced by the selection section of a PQL request.

Based on the matching patterns, PQL allows defining abstractions independent of a particular process model. As opposed to the elementary abstraction operations introduced in Section 2, two high-level abstraction operations

AGGREGATE and REDUCE are introduced. The latter allow abstracting a set of arbitrary process elements (including data elements [15]). Thereby, the process elements to be abstracted are categorized into process element sets based on their type, e.g., node type. If an aggregation shall be applied to a set of process nodes, a minimum SESE block is determined to aggregate adjacent process nodes to an abstracted node. Hence, both well-structuredness and behavioral semantics of the respective process model are preserved.

In PQL, abstractions of process models are initiated by keyword SET. In turn, keywords AGGREGATE and REDUCE indicate the elements to be aggregated or reduced (cf. Listing 5 and Figure 8).

```
set         ::= "SET" operation (("," operation)+)?
operation   ::= abstraction | change_operation
abstraction ::= ("AGGREGATE" | "REDUCE")+ "(" PQLrequest ")"
```

Listing 5. BNF for Structural Matching in a PQL Request

The PQL request depicted in Listing 6 selects all process models that contain any process fragment consisting of a sequence of two activities, i.e., variables a and b. Process elements selected by the first MATCH are then aggregated if their type is $ACTIVITY$ and $val(pe, assignedUser) = Peter$ holds (cf. Figure 8).

```
MATCH  a:ACTIVITY(NAME=C)-->b:ACTIVITY(NAME=F)
SET    AGGREGATE(
         MATCH n:ACTIVITY(*)
         WHERE n.assignedUser=Peter)
```

Listing 6. PQL Request to Aggregate Nodes

Listing 7 shows a PQL request reducing neighboring nodes C and F as described by PQL variables a and b. Note that a second PQL request is nested (cf. Line 3) utilizing the same variables as the parent PQL request does.

```
1 MATCH  a:ACTIVITY(NAME=C)-->b:ACTIVITY(NAME=F)
2 SET    REDUCE(
3          MATCH a-->b)
```

Listing 7. PQL Request to Reduce Nodes

3.4 Changing Process Models

In contemporary process repositories, changes related to multiple process models usually need to be performed on each process model separately. This not only causes high efforts for process designers, but also constitutes an error-prone task. To remedy this drawback, PQL allows changing all process models defined by the selection section of a PQL request at once, i.e., by one and the same change transaction. For example, structural matching patterns can be applied to select the process models to be changed.
Table 4 shows elementary change operations supported by PQL. These may be encapsulated as *high-level change operations*, e.g., inserting a complete process

Operation	**Description**
$InsertNode(P, n_{pred}, n_{succ}, n)$	Node n is inserted between preceding node n_{pred} and succeeding node n_{succ} in process model P. Control edges between n_{pred} and n as well as between n and n_{succ} are inserted to ensure compoundness of the nodes.
$DeleteNode(P, N')$	A set of nodes N' is removed from process model P.
$MoveNode(P, n, n_{pred}, n_{succ})$	Node n is moved from its current position to the one between n_{pred} and n_{succ}, control edges are adjusted accordingly.

Table 4. Examples of Change Operations Supported by PQL

fragment through the application of a set of *InsertNote* operations.

Change operation $InsertNode(P, n_{pred}, n_{succ}, n)$, for example, inserts node n between nodes n_{pred} and n_{succ} in process model P. Thereby, the control edge between n_{pred} and n_{succ} is adjusted and another control edge is inserted to prevent unconnected nodes. Due to lack of space, we omit a discussion of other change operations here and refer interested readers to [16] instead.

Listing 8 shows how to insert a node with type $ACTIVITY$ and name 'New Node' between nodes C and F (cf. Figure 3). Note that the insertion will be applied to any process model in which nodes C and F (cf. Line 2) are present; i.e., the insertion may be applied to a set of process models. Finally, abstractions and changes on process models may be defined in a single PQL request; in this case, changes on process models are applied first.

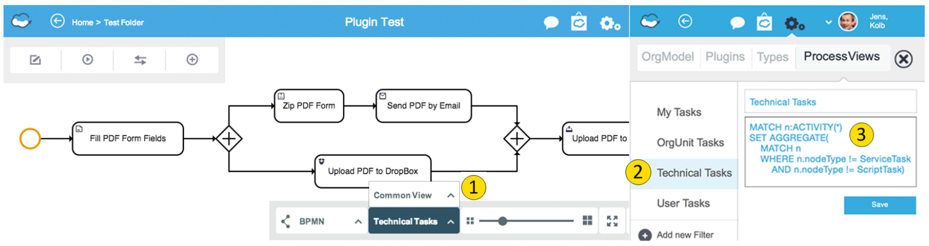
```
1 MATCH a-->b
2 WHERE a.NAME=C, b.NAME=F
3 SET   INSERTNODE(a, b, ACTIVITY, 'New Node')
```

Listing 8. PQL Request to Insert a Node

4 Proof-of-Concept Prototype

In order to demonstrate the applicability of PQL we developed a web-based PAIS called *Clavii BPM Platform*[1]. This platform implements a software architecture supporting the predicate-based definition and creation of process abstractions as well as predicate-based process model changes utilizing PQL [17]. Figure 9 illustrates the creation of a process model abstraction. Drop-down menu ① shows a selection of pre-specified PQL requests directly applicable to a process model. In turn, Figure 9b depicts a screenshot of Clavii's configuration window, where a stored PQL request may be altered. In this case, PQL request 'Technical Tasks' is outlined in ②. The latter aggregates all nodes neither being service tasks nor script tasks (cf. ③). Future research will address the applicability of the prototype and PQL, respectively, in practical settings.

[1] http://www.clavii.com/



Technical Tasks

```
MATCH n:ACTIVITY(*)
SET AGGREGATE(
    MATCH n
    WHERE n.nodeType != ServiceTask
        AND n.nodeType != ScriptTask)
```

a) Process Model Abstraction

b) Stored PQL Request Preserving Technical Tasks

Fig. 9. Process Model Abstractions in the Clavii BPM Platform

5 Related Work

BPMN-Q allows querying process models in [18]. Queries are defined visually using a BPMN-like notation extended with additional attributes. As opposed to PQL requests, BPMN-Q queries are first defined visually and then converted into semantically expanded queries. Model selection is based on the comparison of process element attributes with predicate values specified in the BPMN-Q query, i.e., by measuring the edit-distance. However, BPMN-Q does not support abstractions and changes of the process models selected.

BP-QL is another language for querying process models, which is based on *statecharts* [19]. In particular, a query can be defined in terms of state chart patterns. BP-QL uses pattern matching in respect to node attributes as well as control flow structures when selecting process models. Like BPMN-Q, BP-QL does not allow for changes or abstractions of the selected process models.

A technique for searching and retrieving process variants based on similarity metrics is presented in [20]. More precisely, process variants may be compared by queries comprising structural, behavioral and contextual constraints.

None of the discussed approaches deals with abstractions or changes of the process models selected by a query. Furthermore, the various approaches are based on rather rigid constraints not taking practical issues into account, e.g., regarding the evolution of process models over time.

Several approaches for defining and managing process model abstractions exist. The approach presented in [21] measures semantic similarity between process activities by analyzing the structure of the respective process model. The discovered similarity is then used to abstract the process model. However, this approach neither distinguishes between different user perspectives nor does it provide concepts for flexibly describing process model abstractions. In turn, [22] applies graph reduction techniques to verify structural properties of process models similar to high-level abstraction operations `AGGREGATE` and `REDUCE`. However, no support for querying process models exists.

For defining and changing process models, various approaches exist. In [16], an overview of evidenced change patterns is presented. Furthermore, [23] summarizes approaches enabling flexibility in PAISs. In particular, [24] presents

adaptions of well-structured process models, while preserving their correctness. In turn, [25] presents concepts for evolving process models over time. Finally, changes based on abstracted process models are described in [26, 27, 28].
The *Cypher* query language allows querying and modifying graphs stored in a graph database [11]. By contrast, the *Gremlin* graph query language is a Turing complete programming language realizing queries as chain of operations or functions. Hence, it is suited to construct complex queries for programming languages [29]. The *Structured Query Language (SQL)* offers techniques to manage sets of data in relational databases [7].

6 Summary

We introduced the *Process Query Language (PQL)*, which enables users to automatically select, abstract and change process models in large process model collections. Due to its generic approach, the definition of process model abstractions and changes on any graph-based process notation becomes possible. For this purpose, *structural* and *attributional matching pattern*s are used, which declaratively select process elements either based on the control flow structure of a process model or on element attributes. *PQL* has been implemented in a proof-of-concept prototype demonstrating its applicability. Altogether, process querying languages will be a key part of process repositories to allow for convenient management of process abstractions and changes on large sets of process models.

References

1. Weber, B., Sadiq, S., Reichert, M.: Beyond Rigidity - Dynamic Process Lifecycle Support: A Survey on Dynamic Changes in Process-Aware Information Systems. Computer Science - Research and Development **23**(2) (2009) 47–65
2. Weber, B., Reichert, M., Mendling, J., Reijers, H.A.: Refactoring Large Process Model Repositories. Computers in Industry **62**(5) (2011) 467–486
3. Streit, A., Pham, B., Brown, R.: Visualization Support for Managing Large Business Process Specifications. In: Proc BPM'05. (2005) 205–219
4. Tran, H.: View-Based and Model-Driven Approach for Process-Driven, Service-Oriented Architectures. TU Wien, PhD thesis (2009)
5. Reichert, M., Kolb, J., Bobrik, R., Bauer, T.: Enabling Personalized Visualization of Large Business Processes through Parameterizable Views. In: Proc 27th ACM Symposium On Applied Computing (SAC'12), Riva del Garda, Italy (2012)
6. Chiu, D.K., Cheung, S., Till, S., Karlapalem, K., Li, Q., Kafeza, E.: Workflow View Driven Cross-Organizational Interoperability in a Web Service Environment. Information Technology and Management **5**(3/4) (2004) 221–250
7. Information Technology – Database Languages – SQL – Part 11: Information and Definition Schemas (SQL/Schemata). Norm ISO 9075:2011 (2011)
8. Johnson, R., Pearson, D., Pingali, K.: Finding Regions Fast: Single Entry Single Exit and Control Regions in Linear Time. In: Proc ACM SIGPLAN'93. (1993)

9. Kolb, J., Reichert, M.: A Flexible Approach for Abstracting and Personalizing Large Business Process Models. ACM Applied Comp. Review **13**(1) (2013) 6–17
10. Bobrik, R., Reichert, M., Bauer, T.: View-Based Process Visualization. In: Proc 5th Int'l Conf. on Business Process Management (BPM'07), Brisbane, Australia (2007) 88–95
11. Panzarino, O.: Learning Cypher. Packt Publishing (2014)
12. Robinson, I., Webber, J., Eifrem, E.: Graph Databases. O'Reilly (2013)
13. Backus, J.: Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. Comm ACM **21**(8) (1978) 613–641
14. Wagner, R.A., Fischer, M.J.: The String-to-String Correction Problem. Journal of the ACM **21**(1) (1974) 168–173
15. Kolb, J., Reichert, M.: Data Flow Abstractions and Adaptations through Updatable Process Views. In: Proc 27th Symposium on Applied Computing (SAC'13), Coimbra, Portugal, ACM (2013) 1447–1453
16. Weber, B., Reichert, M., Rinderle, S.: Change Patterns and Change Support Features - Enhancing Flexibility in Process-Aware Information Systems. Data & Knowledge Engineering **66**(3) (2008) 438–466
17. Kammerer, K.: Enabling Personalized Business Process Modeling: The Clavii BPM Platform. Master's thesis, Ulm University (2014)
18. Sakr, S., Awad, A.: A Framework for Querying Graph-Based Business Process Models. In: Proc ACM WWW'10. (2010) 1297–1300
19. Beeri, C., Eyal, A., Kamenkovich, S., Milo, T.: Querying Business Processes. In: Proc VLDB'06. (2006) 343–354
20. Lu, R., Sadiq, S., Governatori, G.: On managing business processes variants. Data & Knowledge Engineering **68**(7) (2009) 642–664
21. Smirnov, S., Reijers, H.A., Weske, M.: A Semantic Approach for Business Process Model Abstraction. In: Proc CAiSE'11, Springer (2011) 497–511
22. Sadiq, W., Orlowska, M.E.: Analyzing Process Models Using Graph Reduction Techniques. Information Systems **25**(2) (2000) 117–134
23. Reichert, M., Weber, B.: Enabling Flexibility in Process-aware Information Systems - Challenges, Methods, Technologies. Springer (2012)
24. Reichert, M., Dadam, P.: ADEPTflex - Supporting Dynamic Changes of Workflows Without Losing Control. Journal of Intelligent Inf. Sys. **10**(2) (1998) 93–129
25. Rinderle, S., Reichert, M., Dadam, P.: Flexible Support of Team Processes by Adaptive Workflow Systems. Distributed and Par. Databases **16**(1) (2004) 91–116
26. Kolb, J., Kammerer, K., Reichert, M.: Updatable Process Views for User-centered Adaption of Large Process Models. In: Proc 10th Int'l Conf. on Service Oriented Computing (ICSOC'12). (2012) 484–498
27. Kolb, J., Kammerer, K., Reichert, M.: Updatable Process Views for Adapting Large Process Models: The proView Demonstrator. In: Demo Track of the 10th Int'l Conf on Business Process Management (BPM'12). (2012) 6–11
28. Kolb, J., Reichert, M.: Supporting Business and IT through Updatable Process Views: The proView Demonstrator. In: Proc 10th Int'l Conf. on Service Oriented Computing (ICSOC'12), Demonstration Track, Shanghai (2013) 460–464
29. TinkerPop: Gremlin, http://gremlin.tinkerpop.com, last visited: 11-14-2014