Process Query Language

Artem Polyvyanyy

Abstract A *process* is a collection of actions that were already, are currently being, or must be taken in order to achieve a goal, where an *action* is an atomic unit of work, for instance, a business activity or an instruction of a computer program. A *process repository* is an organized collection of models that describe processes, for example, a business process repository and a software repository. Process repositories without facilities for *process querying* and *process manipulation* are like databases without Structured Query Language, that is, collections of elements without effective means for deriving value from them. Process Query Language (PQL) is a domain-specific programming language for managing processes described in models stored in process repositories. PQL can be used to query and manipulate process models based on possibly infinite collections of processes that they represent, including processes that support concurrent execution of actions. This chapter presents PQL, its current features, publicly available implementation, planned design and implementation activities, and open research problems associated with the design of the language.

Keywords: Process querying, process manipulation, Process Query Language, PQL

1 Introduction

Computing revolutionizes many aspects of our lives by innovating how data is collected and processed. The innovations often stem from the ability to design, manage, and automatically learn semantically rich artifacts from the data, for example, using machine learning, statistical analysis, and data and process mining techniques.

Artem Polyvyanyy
School of Computing and Information Systems
Faculty of Engineering and Information Technology
The University of Melbourne
Victoria 3010, Australia
e-mail: artem.polyvyanyy@unimelb.edu.au

Such semantically rich artifacts reflect different types of patterns present in the data, calling for dedicated methods for querying and manipulating them to allow systematic derivation of value. One such type of patterns concerns temporal aspects of the data, capturing how work is carried out in processes.

A *process* is a collection of actions that, when executed, lead to the accomplishment of a goal. An *action* is an atomic unit of work. For example, an action can represent a business activity or a computer program instruction. Execution of an action in a process leads to a change in the state of the process. A process can contain already executed actions, actions that are currently being executed, and actions that yet are awaiting their execution. A process solely composed of already executed actions represents a historical process that was observed in the real world. In turn, a process comprising only designed but not executed actions is an envisioned process that may be observed in the future. A *process model* is a model that describes a collection of processes that encode different ways to accomplish the same goal. Note that a process model often describes an infinite collection of processes to address the need to iterate certain actions an initially unknown number of times to achieve the desired process state. Finally, a *process repository* is an organized collection of process models. For example, models can be organized into folders to impose their logical grouping. Examples of process repositories include business process repositories and software repositories.

Process repositories without *process querying* and *process manipulation* capabilities are of low practical utility, as their manual processing is often infeasible. Process Query Language (PQL) is a domain-specific programming language for querying and manipulating process models based on the processes these models describe. It is a declarative language with SQL-like syntax. PQL programs are also called *queries*.

To support process querying, PQL implements two classes of predicates. The first class comprises the *4C behavioral predicates*, a collection of constraints that systematically explore the fundamental behavioral relations of co-occurrence, conflict, causality, and concurrency in processes [22, 28]. These predicates, for instance, can be used to retrieve models that describe processes in which a given action always occurs or in which a given pair of actions can be executed concurrently. The second class is composed of *process scenarios*, sequences of actions with wildcards [25]. Despite being declarative, process scenarios allow checking whether a model describes processes that contain requested sequences of actions. Hence, process scenarios can be used to retrieve models that describe processes that obey the requested imperative constraints.

PQL supports statements for process manipulation. Concretely, one can use PQL to specify and execute instructions for manipulating models to insert, delete, and update processes in the collections of processes these models describe. The process insertion capabilities of PQL are implemented as a solution to the *process repair* problem [20, 25]. The delete and update process manipulations are not implemented in the current version of PQL. Still, they are demonstrated here for the completeness of the discussion of the intended scope for the language.

The next section presents several motivating examples of PQL programs for querying and manipulating processes. Section 3 gives an overview of the features

currently supported by PQL. To facilitate the comparison of PQL with another process querying methods, Section 4 positions PQL within the Process Querying Framework [24]. Then, Section 5 discusses our open-source implementation of a process repository that supports PQL. Section 6 surveys open research problems triggered by the design of PQL and lists planned efforts that aim to shape the language. Finally, Section 7 closes the chapter with conclusions.

2 Motivating Examples

In this section, we present several motivating examples of PQL programs for querying and manipulating process models. To this end, we use an example process repository composed of six process models shown in Fig. 1. The models are captured in Business Process Model and Notation (BPMN). In BPMN, rectangles with rounded corners denote actions. Gateways are visualized as diamonds. Exclusive gateways use the "×" marker inside the diamond shape, whereas parallel gateways use the "+" marker. Directed arcs encode control flow dependencies. For simplicity, the models in the example repository use abstract action labels; see labels `A` through `G` in the figure. In general, an action label specifies the meaning of the action, for example, "assess claim" or "archive case." Models can be further supplied with attributes, for instance, unique identifier, version, creation date, and author. Models can be grouped into collections in a repository by putting them into folders, which, similar to folders of a file system, can form a folder hierarchy.

Models in a repository can be queried using PQL `SELECT` statements. For example, PQL queries Q1 and Q2 listed below implement process querying using the 4C predicates, while PQL queries Q3 and Q4 use process scenarios.

```
Q1. SELECT * FROM *
    WHERE AlwaysOccurs("C") AND
      Cooccur("B","C");
Q2. SELECT "Author", "Version" FROM "/examples"
    WHERE (CanOccur("G") AND
       (NOT Conflict("E","G"))) OR
       (TotalConcurrent("C",{"B","D"},ANY) AND
      AlwaysOccurs("C"));
```

Query Q1 requests to retrieve every model and all its attributes (see "`SELECT *`") from every folder of the repository ("`FROM *`") that describes ("`WHERE`") a collection of processes in which every process contains at least one occurrence of action `C` ("`AlwaysOccurs("C")`") and actions `B` and `C` cooccur in the processes ("`Cooccur("B","C")`"), that is, `B` cannot occur without `C` in a process, `C` cannot occur without `B` in a process, and there exists at least one process in the collection in which both actions `B` and `C` appear. Model 1 in Fig. 1 matches query Q1 and, thus, should be retrieved if Q1 is executed over the repository. Indeed, model 1 describes four processes: ⟨A,B,C,D,E,F⟩, ⟨A,C,B,D,E,F⟩, ⟨A,B,C,D,B⟩, and ⟨A,C,B,D,B⟩; we map BPMN models to Petri nets to interpret them as collections of processes [7].

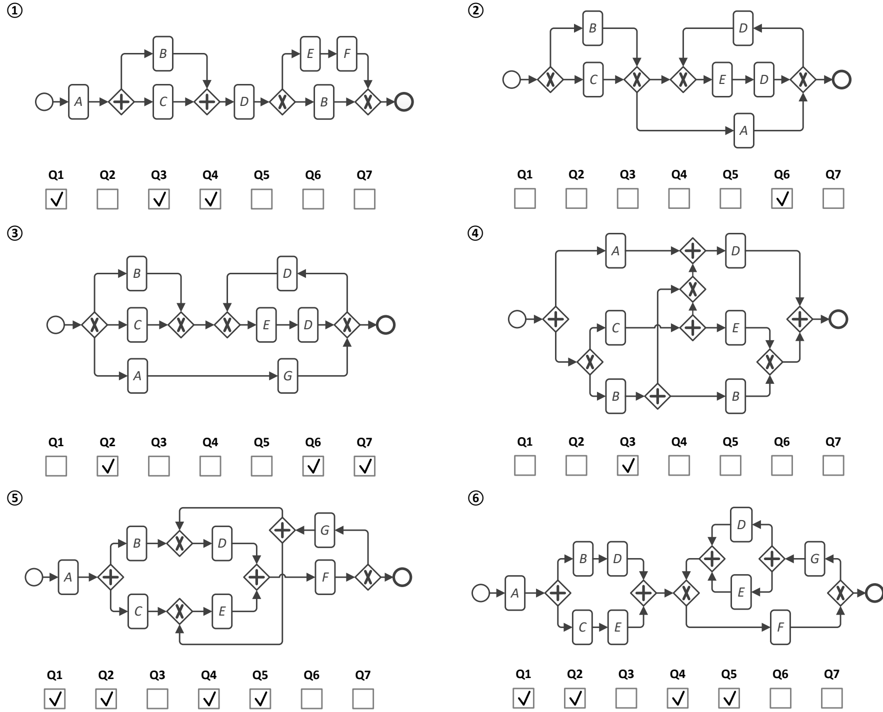

Fig. 1: An example process repository.

Note that action C occurs in every process, while actions B and C cooccur in the processes of the model. Models 5 and 6 also match query Q1. It is easy to verify that both actions B and C occur in all processes these two models describe, as every process starts with one of these two prefixes: ⟨A, B, C⟩ or ⟨A, C, B⟩. To denote which models match which queries, in Fig. 1, under each model, we mark corresponding checkboxes. Hence, models 2, 3, and 4 do not match query Q1. For instance, the process ⟨B, A⟩ described by model 2 confirms that neither C always occurs nor B and C cooccur in the processes of model 2.

Query Q2 requests to retrieve process models and their attributes `Author` and `Version` (“`SELECT "Author", "Version"`”) located in the “`/examples`” folder of the repository (“`FROM "/examples"`”) that satisfy at least one of the two following conditions. First, the model should describe at least one process in which action G occurs at least once (“`CanOccur("G")`”) and actions E and G do not conflict (“`NOT Conflict("E","G")`”), where actions E and G conflict if the model describes at least one process in which E occurs but G does not occur, at least one process in which G occurs but E does not occur, and the model does not describe a process in which both E and G occur. Second, in every process of the model, action C occurs (“`AlwaysOccurs("C")`”), and all occurrences of action C are either concurrent with all occurrences of B or with all occurrences of

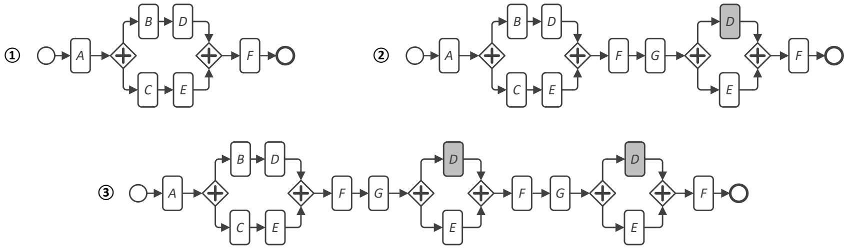

Fig. 2: Three concurrent processes of model 5 from Fig. 1.

D (“TotalConcurrent("C",{"B","D"},ANY)”). In general, two actions A and B are in the total concurrent relation if in every process in which both A and B occur, every occurrence of action A is concurrent with every occurrence of action B; refer to Section 3 for details.

Assuming that all models in Fig. 1 are stored in the “/examples” folder, models 3, 5, and 6 match query Q2. In model 3, action G can occur, consider, for example, the process ⟨A,G⟩ of the model, and actions E and G do not conflict, as evidenced, for instance, by the process ⟨A,G,D,E,D⟩ of the model. In models 5 and 6, in turn, action C always occurs and actions B and C are in the total concurrent relation. Fig. 2 shows three, out of infinitely many, concurrent processes described by model 5. In these three processes, actions B and C occur once and are concurrent; there is no directed path between these actions; for details, again, see Section 3. The same phenomenon can be observed for all the other processes of model 5. Note that the only occurrence of action C is concurrent with the only occurrence of action D in process 1. However, in processes 2 and 3, there are occurrences of action D that are not concurrent with the occurrence of action C. These occurrences are highlighted with gray background in the figure.

PQL query Q3 below requests to retrieve all process models in the repository that support a process that commences with zero or more actions before action B occurs, then eventually action D occurs in the process, followed eventually by another occurrence of action B, and then the process completes via zero or more occurrence of any other actions. Models 1 and 4 from the repository in Fig. 1 match query Q3. This fact is evidenced by processes ⟨A,B,C,D,B⟩ and ⟨A,B,D,B⟩ described by models 1 and 4, respectively. Query Q4, also shown below, requests to retrieve models that describe the process ⟨A,B,C,D,E,F⟩ and does not describe processes with two consecutive occurrences of action B. Models that match this query are models 1, 5, and 6.

```
Q3. SELECT * FROM *
    WHERE Executes(<*,"B",*,"D",*,"B",*>);
Q4. SELECT * FROM *
    WHERE Executes(<"A","B","C","D","E","F">) AND
      NOT Executes(<*,"B","B",*>);
```

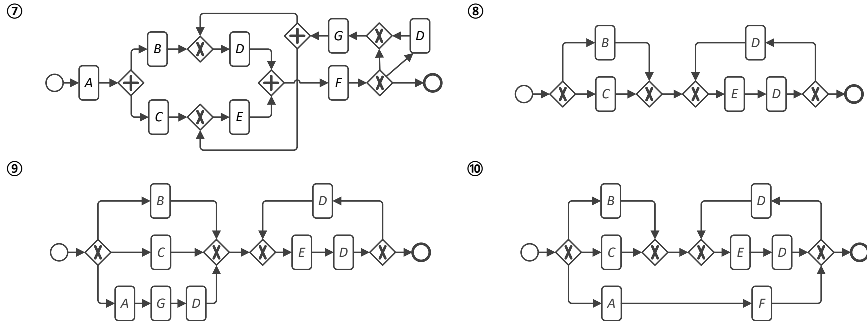

Fig. 3: Manipulated process models.

The attentive reader has noticed that models 5 and 6 from the repository describe the same processes. Thus, these two models, besides being structurally different, are behaviorally equivalent. The result of a PQL query depends on the processes the models describe and is independent of the particular way the models are structured. Consequently, models 5 and 6 either both match or both do not match a given PQL query; refer to the checkboxes next to these two models in Fig. 1.

PQL queries Q5–Q7 below capture instructions for manipulating process models.

```
Q5. INSERT <*,"F","D","G",*> INTO *
    WHERE Executes(<*,"F","G",*>);
Q6. DELETE <"A","G"> FROM *
    WHERE GetTasksAlwaysOccurs(GetTasks())
    EQUALS {};
Q7. UPDATE <"A","G",*>
    SET <"A","F",*>
    FOR *;
```

Query Q5 ensures that each model from every folder of the repository (“`INTO *`”) that describes a process in which an occurrence of action `G` immediately follows an occurrence of action `F` (“`WHERE Executes(<*,"F","G",*>)`”) also describes a process in which an occurrence of `F` is immediately followed by an occurrence of `D` that, in turn, is immediately followed by an occurrence of `G` (“`INSERT <*,"F","D","G",*>`”). If a model that describes the former process also describes the latter requested process, the model is not manipulated. Otherwise, the model is manipulated to obtain an extended version of the model that also describes the requested latter process. Models 5 and 6 in the repository describe processes in which `F` is immediately followed by `G` and, hence, must be manipulated. Model 7 in Fig. 3 is a model that can be created based on model 5 as a result of executing PQL query Q5. Note that model 7 describes the requested process. Note also that the requested manipulation can be implemented in several ways, which raises the question of the quality of the resulting model. This aspect is a subject of ongoing research and is discussed in Section 6.

Query Q6 captures a request to manipulate every process model in the repository (“`FROM *`”) that does not contain an action that occurs in each of its processes (“`WHERE GetTasksAlwaysOccurs(GetTasks()) EQUALS {}`”) and describes the process that starts with an occurrence of action `A` and then immediately completes with an occurrence of action `G` so that the resulting model does not describe that process (“`DELETE <"A","G">`”). Models 2 and 3 from the repository match the condition in the `WHERE` clause. However, only model 3 describes process $\langle \texttt{A}, \texttt{G} \rangle$, and, thus, should be manipulated. The resulting, manipulated by PQL, model is added as a fresh model to the repository. Similar as for the `INSERT` statement, several valid resulting models can be considered. For example, models 8 and 9 in Fig. 3 can be accepted as models that result from executing query Q6 over model 3. While model 8 does not describe all the processes with prefix $\langle \texttt{A}, \texttt{G} \rangle$ described by model 3, including the requested process $\langle \texttt{A}, \texttt{G} \rangle$, the processes described by model 9 differ from those described by model 3 by exactly one process $\langle \texttt{A}, \texttt{G} \rangle$. Note that the implementation of the `DELETE` statement can vary between versions of PQL.

Finally, query Q7 requests to update all models in the repository (“`FOR *`”) by updating processes that start with the prefix $\langle \texttt{A}, \texttt{G} \rangle$ (“`UPDATE <"A","G",*>`”) to start with the prefix $\langle \texttt{A}, \texttt{F} \rangle$ (“`SET <"A","F",*>`”). Again, multiple implementations of the `UPDATE` statement can be envisaged, and model 10 in Fig. 3 is a possible result of executing query Q7 over model 3, which is also the only model in Fig. 1 that must be manipulated according to query Q7.

3 Process Query Language

This section reviews the core features of PQL. First, Section 3.1 discusses the main primitives of PQL for querying process models. Then, Section 3.2 presents the currently implemented PQL mechanisms for manipulating process models.

3.1 Process Querying

For the purpose of process querying, PQL interprets a process model as a collection of *concurrent processes*. A *concurrent process* is a collection of actions such that for some pairs of actions in the collection, it is specified that one of the actions *causally precedes* the other in the executions of the process. The control flow arcs and the transitive dependencies that these arcs induce in Fig. 2 define the causal precedence relations of the corresponding concurrent processes. In concurrent process 1 in Fig. 2, for example, action `A` causally precedes action `E`, which, in turn, causally precedes action `F`. In contrast, for the pairs of actions that are not in the causal precedence relation, it is accepted that they are independent, or *concurrent*, and, thus, can be performed simultaneously in the executions of the process. For instance, actions `B`

and E are concurrent in process 1 in Fig. 2. As already explained in Section 2, a process model can describe infinitely many concurrent processes.

Every concurrent process describes a collection of (sequential) processes. These are processes that do not violate the causal precedence constraints of the concurrent process. For example, concurrent process 2 in Fig. 2 describes twelve sequential processes, induced by all the interleavings of the concurrent actions; these twelve processes include, for instance, processes $\langle \mathtt{A,B,C,E,D,F,G,D,E,F} \rangle$ and $\langle \mathtt{A,C,B,E,D,F,G,E,D,F} \rangle$. Every concurrent process describes a finite collection of sequential processes. But, needless to say, a model that describes infinitely many concurrent processes also describes infinitely many sequential processes.

To perform process querying using PQL, the user can specify a query that requests to retrieve models that fulfill a condition verified over all the processes of the models. One way to specify a condition is by using behavioral predicates, as detailed in Section 3.1.1, or scenarios, as discussed in Section 3.1.2.

3.1.1 Behavioral Predicates

Process models describe processes composed of actions that can be executed and, thus, observed in the real world. One way to convey how many occurrences of an action, or pairs of actions in a specific behavioral relationship, can be observed in the executions of processes described by the model is by using predicates with quantifiers.[1] When studying process models, the user may, for instance, be interested in how often certain actions can occur, how often certain actions can cause occurrences of other actions, or how often actions can be executed simultaneously.

The 4C spectrum is a systematically organized repertoire of predicates that assess in how many processes that a model describes how many occurrences of one action are in a specific behavioral relation with how many occurrences of another action [28]. The predicates of the spectrum explore the fundamental *behavioral relations* of co-occurrence, conflict, causality, and concurrency of action occurrences in processes. Hence, we refer to these predicates as *behavioral predicates*.

A PQL query can use predicates of the 4C spectrum as atomic propositions in the propositional logic formula of its WHERE clause. When a model is matched to a query, the value of each predicate is established based on the processes that the model describes. If the formula in the WHERE clause of a SELECT statement evaluates to *true* for a particular model, then the model is included in the result of the query. Additional checks may need to be applied for other PQL statement types to confirm that the model indeed must be manipulated.

To accompany the 4C predicates, all of which are binary predicates, that is, they take two actions as input, PQL supports two unary predicates listed in Table 1. As suggested by their definitions, these predicates allow verifying the frequencies of individual action occurrences, for example, before applying the 4C predicates, which then can explain how these occurrences relate to each other.

[1] A *predicate* is a function that evaluates to either *true* or *false* truth value, while a *quantifier* is an operator that specifies how many elements from the given collection should satisfy an open formula.

Predicate	Definition
`CanOccur(A)`	The predicate evaluates to *true* if the model describes *at least one* process with *at least one* occurrence of action `A`; otherwise, it evaluates to *false*.
`AlwaysOccurs(A)`	The predicate evaluates to *true* if *every* process the model describes has *at least one* occurrence of action `A`; otherwise, it evaluates to *false*.

Table 1: Occurrence predicates; the predicates are evaluated in the context of a process model.

Table 2 lists six 4C predicates grounded in the conflict and co-occurrence behavioral relations. Note that the `CanConflict` and `CanCooccur` predicates are seminal as the remaining four predicates from the table can be expressed as propositional logic formulas over them. Hence, these four predicates can be seen as macros that can simplify the conditions the user may want to express in the `WHERE` clause of a PQL query. The `CanConflict` and `CanCooccur` predicates can be combined into logic formulas to express other conditions that explore conflict and co-occurrence behavioral relations. According to one classification, 63 conflict and 15 co-occurrence properties can be expressed this way [28].

Predicate	Definition
`CanConflict(A,B)`	The predicate evaluates to *true* if the model describes *at least one* process with *at least one* occurrence of action `A` and *no* occurrences of action `B`; otherwise, it evaluates to *false*.
`CanCooccur(A,B)`	The predicate evaluates to *true* if the model describes *at least one* process with *at least one* occurrence of action `A` and *at least one* occurrence of action `B`; otherwise, it evaluates to *false*.
`Conflict(A,B)`	The predicate evaluates to *true* if the model describes *no* process with *at least one* occurrence of action `A` and *at least one* occurrence of action `B`; otherwise, it evaluates to *false*.
`Cooccur(A,B)`	The predicate evaluates to *true* if *every* process the model describes that has *at least one* occurrence of action `A` also has *at least one* occurrence of action `B`, and *vice versa*; otherwise, it evaluates to *false*.
`Requires(A,B)`	The predicate evaluates to *true* if the model describes *no* process with *at least one* occurrence of action `A` and *no* occurrences of action `B`, *at least one* process with *at least one* occurrence of action `B` and *no* occurrences of action `A`, and *at least one* process with *at least one* occurrence of action `A` and *at least one* occurrence of action `B`; otherwise, it evaluates to *false*.
`Independent(A,B)`	The predicate evaluates to *true* if the model describes *at least one* process with *at least one* occurrence of action `A` and *no* occurrences of action `B`, *at least one* process with *at least one* occurrence of action `B` and *no* occurrences of action `A`, and *at least one* process with *at least one* occurrence of action `A` and *at least one* occurrence of action `B`; otherwise, it evaluates to *false*.

Table 2: Co-occurrence and conflict predicates; the predicates are evaluated in the context of a process model.

Table 3 lists all the 4C predicates grounded in the causal precedence and concurrency behavioral relations. Given actions A and B, the predicates emerge through universal or existential quantification over three domains, namely the collection of all concurrent processes that the model describes (see column "Pr." in the table), the collection of all occurrences of action A in a concurrent process the model describes (column "A"), and the collection of all occurrences of action B in the same concurrent process, and the choice of the behavioral relation between the occurrences of actions A and B (column "Rel."), either causal precedence ("Causal.") or concurrency ("Concur."). These configurations lead to eight causality and eight concurrency predicates. The syntax of the behavioral predicates in PQL and their names are provided in columns "Syntax" and "Name" of Table 3, respectively.

Pr.	A	B	**Rel.**	**Syntax**	**Name**
∀	∀	∀	Causal.	`TotalCausal(A,B)`	Total (mutual) causal
∀	∀	∀	Concur.	`TotalConcurrent(A,B)`	Total (mutual) concurrent
∀	∀	∃	Causal.	`TotalFunctionalCausal(A,B)`	Total functional causal
∀	∀	∃	Concur.	`TotalFunctionalConcurrent(A,B)`	Total functional concurrent
∀	∃	∀	Causal.	`TotalDominantCausal(A,B)`	Total dominant causal
∀	∃	∀	Concur.	`TotalDominantConcurrent(A,B)`	Total dominant concurrent
∀	∃	∃	Causal.	`TotalExistCausal(A,B)`	Total existential causal
∀	∃	∃	Concur.	`TotalExistConcurrent(A,B)`	Total existential concurrent
∃	∀	∀	Causal.	`ExistTotalCausal(A,B)`	Existential total causal
∃	∀	∀	Concur.	`ExistTotalConcurrent(A,B)`	Existential total concurrent
∃	∀	∃	Causal.	`ExistFunctionalCausal(A,B)`	Existential functional causal
∃	∀	∃	Concur.	`ExistFunctionalConcurrent(A,B)`	Existential functional concurrent
∃	∃	∀	Causal.	`ExistDominantCausal(A,B)`	Existential dominant causal
∃	∃	∀	Concur.	`ExistDominantConcurrent(A,B)`	Existential dominant concurrent
∃	∃	∃	Causal.	`ExistCausal(A,B)`	Existential (mutual) causal
∃	∃	∃	Concur.	`ExistConcurrent(A,B)`	Existential (mutual) concurrent

Table 3: Concurrency and causality predicates; "Pr." – concurrent processes of the model, "A" – occurrences of action A in the concurrent process, "B" – occurrences of action B in the concurrent process, "∀" – every concurrent process of the model/every occurrence of the action in the concurrent process, "∃" – exists a concurrent process of the model/exists an occurrence of the action in the concurrent process, "Rel." – behavioral relation, "Syntax" – PQL syntax for expressing the predicate, and "Name" – the name of the predicate. The predicates are evaluated in the context of a process model.

For example, the total concurrent predicate evaluates to *true* for input actions A and B, if in every ("∀") concurrent process the model describes that has at least one occurrence of action A and at least one occurrence of action B, it holds that every ("∀") occurrence of action A is concurrent ("Concur.") with every ("∀") occurrence of action B; otherwise, the total concurrent predicate evaluates to *false* for that input. Thus, `TotalConcurrent(B,C)` evaluates to *true* for model 5 in Fig. 1. Indeed, every concurrent process of model 5 contains exactly one occurrence of action B, exactly one occurrence of action C, and these occurrences are concurrent; see three out of infinitely many concurrent processes model 5 describes

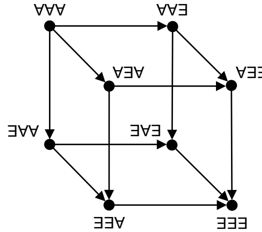

Fig. 4: The 4C spectrum causality/concurrency lattice [28].

in Fig. 2. In contrast, `TotalConcurrent(D,E)` evaluates to *false* for model 5 and processes 2 and 3 in Fig. 2 evidence this, as they contain occurrences of `D` and `E` that are in the causal precedence relation. However, process 1 in Fig. 2 justifies the fact that `ExistTotalConcurrent ̃(D,E)` holds *true*. This predicate verifies whether there exists a concurrent process described by the model in which all occurrences of actions are concurrent. In process 1, there is exactly one occurrence of action `D`, exactly one occurrence of action `E`, and these two occurrences are concurrent. Note, however, that "stronger" concurrency relations also hold between actions `D` and `E` in model 5, for instance, `TotalFunctionalConcur(D,E)` and `TotalFunctionalConcur(E,D)`. Indeed, in every ("∀") concurrent process of model 5, for every ("∀") occurrence of action `D` in the process, there exists ("∃") an occurrence of action `E` that is concurrent with that occurrence of `D`, and vice versa.

As examples of the causality predicates, note that `TotalCausal(B,D)` holds, but `TotalCausal(C,D)` does not hold for model 5 from Fig. 1. In every concurrent process in Fig. 2 it holds that the only occurrence of action `C` is concurrent to one occurrence of action `D`, invalidating the total causal relation between the actions. In contrast, the only occurrence of action `B` is in the causal precedence relation with every occurrence of action `D` in every concurrent process of model 5.

Table 4 lists definitions of all the eight 4C causality predicates. The definitions of the eight concurrency predicates can be obtained by replacing the causal precedence relations with the concurrency relations. Furthermore, Polyvyanyy et al. [28] formalize all the predicates using mathematical notation.

As already mentioned, causality and concurrency predicates can be distinguished based on their "strength." Fig. 4 summarizes implications between the pairs of causality (or concurrency) predicates from the 4C spectrum; the transitive implications are not shown. The vertices represent causality (or concurrency) predicates, while the labels encode the quantifiers from the first three columns in Table 3. Hence, for example, the fact that the `ExistTotalCausal` predicate holds for a given pair of actions (see the "∃∀∀" label in Fig. 4) implies that both `ExistFunctionalCausal` ("∃∀∃") and `ExistDominantCausal` ("∃∃∀") predicates hold and, transitively, `ExistCausal` ("∃∃∃") holds for the same pair of actions; note that the converse implications, in general, do not hold. Consequently, we say that `ExistTotalCausal` is stronger than the other existential causality predicates.

Predicate	Definition
`ExistCausal(A,B)`	The predicate evaluates to *true* if the model describes *at least one* concurrent process in which *at least one* occurrence of action `A` causally precedes *at least one* occurrence of action `B`; otherwise, it evaluates to *false*.
`ExistDominantCausal(A,B)`	The predicate evaluates to *true* if the model describes *at least one* concurrent process with *at least one* occurrence of action `B` and, in that concurrent process, there is *one* occurrence of action `A` that causally precedes *every* occurrence of action `B`; otherwise, it evaluates to *false*.
`ExistFunctionalCausal(A,B)`	The predicate evaluates to *true* if the model describes *at least one* concurrent process with *at least one* occurrence of action `A` and, in that concurrent process, there is *one* occurrence of action `B` such that *every* occurrence of action `A` causally precedes that occurrence of action `B`; otherwise, it evaluates to *false*.
`ExistTotalCausal(A,B)`	The predicate evaluates to *true* if the model describes *at least one* concurrent process with *at least one* occurrence of action `A`, *at least one* occurrence of action `B`, and, in that concurrent process, *every* occurrence of action `A` causally precedes *every* occurrence of action `B`; otherwise, it evaluates to *false*.
`TotalExistCausal(A,B)`	The predicate evaluates to *true* if in *every* concurrent process the model describes that has *at least one* occurrence of action `A` and *at least one* occurrence of action `B`, *at least one* occurrence of action `A` causally precedes *at least one* occurrence of action `B`; otherwise, it evaluates to *false*.
`TotalDominantCausal(A,B)`	The predicate evaluates to *true* if in *every* concurrent process the model describes that has *at least one* occurrence of action `A` and *at least one* occurrence of action `B`, there is *one* occurrence of action `A` that causally precedes *every* occurrence of action `B`; otherwise, it evaluates to *false*.
`TotalFunctionalCausal(A,B)`	The predicate evaluates to *true* if in *every* concurrent process the model describes that has *at least one* occurrence of action `A` and *at least one* occurrence of action `B`, there is *one* occurrence of action `B` such that *every* occurrence of action `A` causally precedes that occurrence of action `B`; otherwise, it evaluates to *false*.
`TotalCausal(A,B)`	The predicate evaluates to *true* if in *every* concurrent process the model describes that has *at least one* occurrence of action `A` and *at least one* occurrence of action `B`, *every* occurrence of action `A` causally precedes *every* occurrence of action `B`; otherwise, it evaluates to *false*.

Table 4: Causality predicates. The predicates are evaluated in the context of a process model.

In a study with the prospective stakeholders of PQL, all twelve preselected 4C predicates were recognized as suitable for process querying. The `CanOccur`, `AlwaysOccurs`, `Cooccur`, `Conflict`, `TotalCausal`, and `TotalConcurrent` predicates were, in addition, identified as most useful and such that are most likely to be used for solving practical problems [22].

3.1.2 Scenarios

Any finite repertoire of behavioral predicates is limited in its expressive power, as it can only express a finite number of conditions over a fixed collection of actions, while the number of process collections that process models can express over the same actions is unbounded [21]. Therefore, in addition to querying based on the 4C predicates, PQL supports scenario-based querying [25].

The concept central to scenario-based querying is the notion of a *trace with wildcards*. A trace with wildcards is a finite sequence in which every element is either a special wildcard element '$*$' or a pair composed of an action and a number between zero and one. For example, $\omega = \langle *, (\mathtt{A}, 1.0), (\mathtt{B}, 0.8), *, (\mathtt{A}, 1.0) \rangle$ is a trace with wildcards composed of five elements.

A trace with wildcards defines a collection of processes. These processes result from the concatenation of collections of sequences defined by the elements of the trace. The concatenation is performed in the order the corresponding elements appear in the trace. The special '$*$' element defines the collection of all finite sequences over all possible actions. In turn, an element that is a pair of an action x and a number y defines the collection of all sequences composed of one action, where the actions are taken from the set of all actions that are similar with x to the level of at least y; the similarity should be established based on some given similarity function that maps pairs of actions to their similarity scores between zero and one. Different similarity functions can be used. For instance, one such similarity function can be established based on the similarity of action names or labels. Thus, ω defines the collection that includes every process in which action `A` eventually occurs, that occurrence is immediately followed by an occurrence of action `B`, or an occurrence of some similar with `B` action, and then some other actions can occur before the process ends with yet another occurrence of action `A`.

The `Executes` predicate takes as input a trace with wildcards and verifies, in the context of a given process model, whether the model describes at least one process that is also included in the collection of processes defined by the trace. In other words, it verifies whether the model can execute actions according to the pattern captured by the trace. If so, the predicate returns *true*; otherwise, it returns *false*. The concrete syntax of the `Executes` predicate for the input trace with wildcards ω is `Executes(<*,"A","B"[0.8],*,"A">)`, or `Executes(<*,"A",~"B",*,"A">)` if the process querying tool is configured to use 0.8 as the default action similarity threshold.

The `Executes` predicates can be used, together with the 4C predicates, as atomic propositions in the propositional logic formula of the `WHERE` clause of a PQL query, thus enriching the expressive power of the language. Indeed, by combining `Executes` predicates, one can, for instance, express a condition to check whether a given model describes, or does not describe, some finite collection of processes of interest. Note that, in general, the number of such conditions is unbounded. For more information on the scenario-based querying support in PQL, refer to [25].

3.2 Process Manipulation

Process manipulations in PQL are implemented using the concept of an *optimal alignment* between a process and a process model [3,4]. An alignment is composed of moves. A *synchronous move* is a pair in which both elements are the same action, for example (A, A). In contrast, an *asynchronous move* is a pair in which one element is an action, and the other element is a special “no move” element, denoted by ‘≫’. An *alignment* is a sequence of synchronous and asynchronous moves for which two conditions hold. First, the first elements from the moves, when positioned in the order the corresponding moves appear in the alignment and all the “no move” elements are skipped, form the process. Second, the second elements from the moves, again positioned as in the alignment and without the “no move” elements, form a process described by the model. Finally, an optimal alignment between a process and model is an alignment between the process and model such that every other alignment between them has more asynchronous moves than an optimal alignment.

An alignment is often summarized as a table. For instance, Table 5 shows an optimal alignment between process ⟨F,D,G⟩ and process model 5 from Fig. 1. It is a sequence of thirteen moves. In the table, moves are encoded as columns, such that two successive columns refer to two successive moves in the alignment. Each column has two rows. The top row of each column specifies the first element in the corresponding move, while the bottom row specifies the second element in the move. Hence, the optimal alignment in Table 5 consists of two synchronous and eleven asynchronous moves.

≫	≫	≫	≫	≫	F	D	G	≫	≫	≫
A	C	B	E	D	F	≫	G	D	E	F

Table 5: An optimal alignment between process ⟨F,D,G⟩ and process model 5 from Fig. 1.

For instance, PQL relies on the alignment from Table 5 to implement query Q5 discussed in Section 2 on model 5 from Fig. 1. Indeed, the alignment demonstrates that the process fragment ⟨F,D,G⟩ requested to be inserted into the model, see “INSERT <∗,"F","D","G",∗>” in the query, has a “gap” captured by the asynchronous move (D, ≫) in the processes described by the model, see the move highlighted with gray background in the alignment. This asynchronous move determines the place in the model at which action D can be inserted; after process-prefix ⟨A,C,B,E,D,F⟩ and before process-suffix⟨G,D,E,F⟩. The concrete modifications on the model are then implemented using *process repair* techniques [10,20] from the field of process mining [2]. Recall that model 7 in Fig. 3 is a model that results from executing query Q5 on model 5.

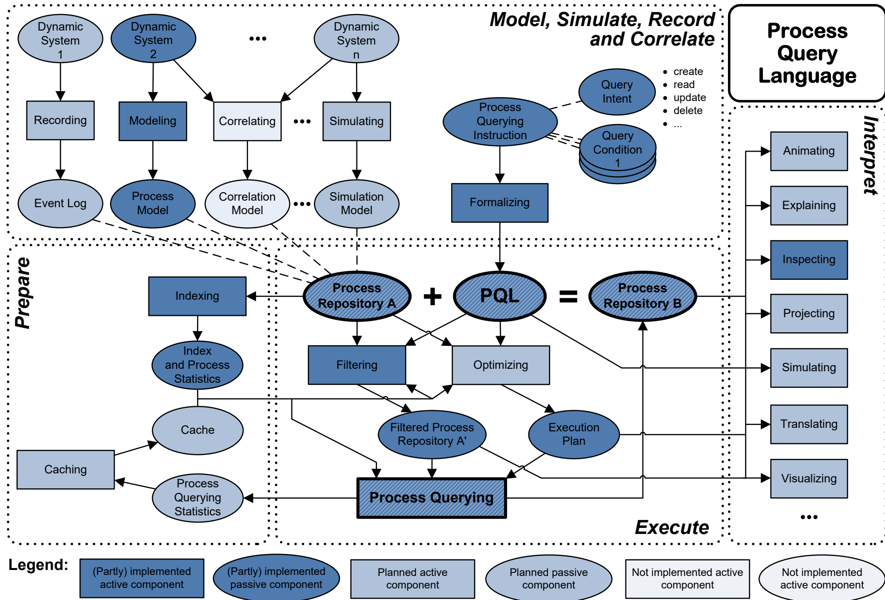

Fig. 5: A schematic view of the components of the Process Querying Framework implemented in PQL; adapted from [24].

4 Process Querying Framework

The *Process Querying Framework* (PQF) is an abstract system of components that can be selectively replaced to result in a new process querying method [24]. In this section, we identify which active and passive components of the framework are supported in PQL. The aim of this exercise is threefold: Tracking the status of the PQL implementation, planning the next design and implementation activities, and preparation of PQL for comparison with other process querying methods positioned within the framework.

Fig. 5 shows a schematic view of the framework. In the figure, rectangles and ovals denote active and passive components, respectively. The arcs denote input and output passive components of active components. That is, the passive components are consumed and produced by the active components. Dashed lines encode the aggregation relationships between the passive components. Finally, we use different backgrounds to reflect the different implementation statuses of the components; refer to the legend in the figure. The framework consists of four parts, each responsible for one dedicated function, including managing processes and queries, preparing and executing queries, and supporting the interpretation of querying results. In Fig. 5, each part is enclosed in an area with a dotted border.

The "Model, Simulate, Record, and Correlate" part of the framework is responsible for the management of the process repository and process queries. In general, the

repository can comprise different types of models of processes. PQL was initially introduced to address querying of process models, that is, conceptual models that describe collections of processes. Examples of process models are Petri nets, BPMN diagrams, Event-driven Process Chains (EPCs), and UML Activity Diagrams. The current implementation of PQL works with process models formalized as Petri nets. Note that for many process modeling notations, the corresponding mappings to Petri nets have been devised. Being able to query process models, PQL can be adapted for querying their recorded executions, also known as *event logs* in process mining [2], and, consequently, to *simulation models*, as combinations of models and their executions. The extension of PQL to support querying over event logs and simulation models is future work. Other models that describe processes, for instance, correlation models that specify relationships between multiple processes, are not currently supported by PQL. A process querying instruction specifies an intent to query or manipulate a process repository utilizing various query conditions. PQL is a language for formalizing process querying instructions. It supports process querying by means of the *read* intent implemented using `SELECT` statements. In the current version of PQL, process manipulation is implemented using `INSERT` statements that address the *create* and *update* process querying intents. In the future, the support of the *update* intent will be supplemented by `UPDATE` and `DELETE` PQL statements.

The "Prepare" part of the framework, as its name suggests, is responsible for preparing the process repository for efficient querying. The framework offers two types of preparations: indexing and caching. The *Indexing* component takes a process repository as input and constructs its alternative representation, called an *index*, which is then used to optimize computations during the execution of process queries. PQL implements indexing of the 4C behavioral predicates for all the process models in the repository. At runtime, when computing PQL programs, the precomputed behavioral relations are accessed in the index in close to real-time and reused. We plan to implement an additional index based on the special data structures, called *untanglings* of process models [23]. Untanglings can be used to efficiently identify groups of actions that can be executed together in some process. The *Caching* component stores data computed in the previous executions of PQL programs that then gets reused in computations of the future PQL programs. We plan to implement caching in PQL based on the statistics of the past PQL program executions.

The "Execute" part is responsible for executing process queries and comprises components for filtering process repositories and optimizing and executing process queries. For efficiency considerations, before a PQL program is executed, models that clearly should not be included in the result of the program are filtered away. The *Filtering* component of PQL checks whether actions that, according to the PQL program, must or must not be present in the result of the program are indeed described or not described, respectively, by the input model. We will extend this capability with filtering based on the untanglings to detect if combinations of actions can or cannot occur in an execution of the candidate model or process. Design and implementation of comprehensive query optimization mechanisms in PQL is future work. In the current implementation of the language, the execution plan of a PQL program is guided by its parse tree. Basic execution optimizations are supported. For

example, when the result of a propositional logic formula is known based on a subset of its propositions, the other propositions are not computed. Finally, the *Process Querying* component of the PQL method implements the formal semantics of the language; see [22, 25] for details. When a PQL program is executed, it takes as input a process repository and produces another repository consisting of the retrieved and manipulated, as requested by the PQL program, models.

The "Interpret" part of the framework is responsible for communicating the querying results to the user. All the components of this part aim to improve the comprehension of the results. The components are inspired by the various means for improving comprehension of conceptual models [16]. PQL results are encoded as process models or processes. The user can foster their understanding by inspecting, or reading, them. Future work will address the design, implementation, and evaluation of other techniques for explaining, projecting, translating, visualizing, animating, and simulating results of PQL programs for their better comprehension.

5 Implementation

The PQL querying method has been implemented in an open-source process repository.[2] Users interact with the repository via command-line interfaces (CLIs) of two utilities: the PQL bot and the PQL tool. The PQL bot prepares models for querying, while the PQL tool executes PQL programs over the models stored in the repository.

PQL programs process only indexed models. The PQL bot systematically indexes models in the repository. One can start multiple bot instances simultaneously to index multiple models in parallel. To construct an index, a bot instance computes all the 4C behavioral predicates over all the actions of the model using three types of analysis over the reachable states described by the model: the *reachability analysis* [12], the *coverability analysis* [29], and the *structural analysis over a complete prefix* [9, 18] of the *unfolding* [19] of the model. PQL bots use the solutions to the reachability and covering problems implemented in the `LoLA` tool version 2.0 [31]. The implementation of the algorithm by Esparza et al. [9], available as part of the `jBPT` library [27], is used to construct finite complete prefixes of unfoldings.

Process models stored in the repository are Petri nets described using the Petri Net Markup Language (PNML) [5]. Many high-level process modeling languages, such as BPMN and EPC, can be translated to Petri nets [1, 7]. As a result, PQL can be used to query and manipulate models developed using a wide range of notations.

The listing below shows a sample output of a PQL bot instance. One can configure a bot instance by specifying its name (option `-n`), time to sleep (i.e., stay idle) between indexing two models (option `-s`), and maximal time to attempt indexing a model (option `-i`). Once started, a bot instance alternates sleeping and indexing phases and sends periodic alive messages to the repository. Before indexing, models are checked for semantic correctness.

2 https://github.com/processquerying/PQL.git

```
>> java -jar PQL.BOT-1.0.jar -n=Brisbane -s=60 -i=3600
>> ======================================================================
>> Process Query Language (PQL) Bot ver. 1.0
>> ======================================================================
>> Name:                Brisbane
>> Sleep time:          60s
>> Max. index time:     3600s
>> ======================================================================
>> 10:45:18.487 Brisbane - There are no pending jobs
>> 10:45:18.487 Brisbane - Sent an alive message
>> 10:45:18.497 Brisbane - Going to sleep for 60 seconds
>> 10:46:18.505 Brisbane - Woke up
>> 10:46:18.525 Brisbane - Retrieved indexing job for the model with ID 1
>> 10:46:18.575 Brisbane - Start checking model with ID 1
>> 10:46:23.506 Brisbane - Finished checking model with ID 1
>> 10:46:23.506 Brisbane - Start indexing model with ID 1
>> 10:47:03.608 Brisbane - Finished indexing model with ID 1
>> 10:47:03.608 Brisbane - Going to sleep for 60 seconds
>> 10:48:03.613 Brisbane - Woke up
>> 10:48:03.623 Brisbane - Retrieved indexing job for the model with ID 2
>> 10:48:03.673 Brisbane - Start checking model with ID 2
>> 10:48:13.248 Brisbane - Finished checking model with ID 2
>> 10:48:13.249 Brisbane - Start indexing model with ID 2
>> 10:49:52.679 Brisbane - Finished indexing model with ID 2
>> 10:49:52.679 Brisbane - Going to sleep for 60 seconds
>> 10:50:52.704 Brisbane - Woke up
>> 10:50:52.704 Brisbane - There are no pending jobs
>> ...
```

Table 6 lists several CLI options of the PQL tool. For example, the PQL tool can be used to store (option `-s`), check (option `-c`), index (option `-i`), and delete (option `-d`) a process model, visualize the parse tree of a PQL program (option `-p`), execute a PQL program (options `-q`), and to reset the repository (option `-r`).

Option name	**Short name**	**Parameter**	**Description**	**Required option**
`-check`	`-c`		Check if model can be indexed	`-id`
`-delete`	`-d`		Delete model (and its index)	`-id`
`-index`	`-i`		Index model	`-id`
`-identifier`	`-id`	`<string>`	Model identifier	
`-parse`	`-p`		Show PQL program parse tree	`-pql`
`-pnmlPath`	`-pnml`	`<path>`	Path to a PNML file	
`-pqlPath`	`-pql`	`<path>`	Path to a PQL file	
`-query`	`-q`		Execute PQL program	`-pql`
`-reset`	`-r`		Reset repository	
`-store`	`-s`		Store model in the repository	`-pnml` (`-id`)

Table 6: CLI options of the PQL tool.

To store models in the repository, the CLI option `-s` of the PQL tool must be accompanied by the `-pnml` option that specifies a path to a single PNML file or to a directory that contains many PNML files. If a path to a single PNML file is specified, the call must include option `-id` to specify a unique identifier to associate with the model; otherwise, the models are attempted to be stored using their file names as unique identifiers. A stored model can be indexed by a PQL bot instance or by the PQL tool using the CLI option `-i` accompanied by option `-id` that specifies the

unique identifier of the model that should be indexed. When indexing a model, the PQL tool uses the same routines as the PQL bot.

To execute a PQL program, the user can use options `-q` and `-pql` of the PQL tool. The latter specifies a path to a file that contains the program. An example command-line output of executing a PQL program is shown below. Here, the PQL tool is requested to execute the PQL program stored in the `prog.pql` file. The program requests to retrieve every model in the repository in which the "process payment" action, or a similar action, occurs in every execution the model describes; note that two similar actions, "process payment by cash" and "process payment by check," were found in the repository for the requested similarity threshold of 0.8. The tool retrieved two models that match the query. These are models with identifiers 364 and 778; see the last line of the listing.

```
>> java -jar PQL.TOOL-1.0.jar -q -pql=prog.pql
>> PQL query:  SELECT * FROM * WHERE AlwaysOccurs("process payment"[0.8]);
>> Attributes: [UNIVERSE]
>> Locations:  [UNIVERSE]
>> Task:       "process payment"[0.8] -> ["process payment by cash",
>>             "process payment by check"]
>>
>> Result:     [364, 778]
```

The PQL tool supports multi-threaded querying. The user can configure the desired number of threads to use for executing PQL programs. As a result of executing a PQL program, the tool returns a collection of matching and augmented models.

6 Discussion

The design of PQL aims to maximize the number of supported process querying and process manipulation techniques, as requested by the *process querying compromise* [24], which identifies a concrete process querying method as an intersection of implemented decidable, efficient, and suitable techniques. In this section, we discuss research problems that emerged during the design of PQL, and solutions to these problems that shaped PQL and will inform the future extensions to the language. First, Section 6.1 discusses four fundamental problems of process querying that PQL aims to solve. Then, Section 6.2 discusses problems that aim to ensure the quality of process querying and manipulation operations performed by PQL. Next, Section 6.3 summarizes conducted work to establish the suitability of PQL. Finally, Section 6.4 is devoted to the aspects related to the ability to compute PQL queries efficiently.

6.1 Querying and Manipulation

Given a process model and a process query that describes a collection of processes, the process querying problem is a decision problem that consists in checking whether the model describes processes from the collection.

Process querying problem. Given a process model and a description of a collection of processes, check if the model describes processes included in the collection.

PQL can be used to pose and solve process querying problems via `SELECT` statements. One may want to augment a process model so that the collection of processes it describes includes specified processes. This task can be fulfilled by solving the process insertion problem.

Process insertion problem. Given a process model and a description of a collection of processes, construct a process model that describes processes captured in the model and included in the collection.

PQL `INSERT` statements can be used to express and solve process insertion problems. In contrast, if a model needs to be augmented to describe processes of the original model without some specific processes, a process deletion problem must be solved.

Process deletion problem. Given a process model and a description of a collection of processes, construct a process model that describes processes captured in the model but not included in the collection.

One can use PQL `DELETE` statements to formulate and solve process deletion problems. However, if specific processes must be replaced in the collection of processes described by a model, a process update problem must be solved.

Process update problem. Given a process model, a description of a collection of source processes, and a description of a collection of target processes, construct a process model that describes processes captured in the model and included in the target collection but not included in the source collection.

PQL `UPDATE` statements can be used for expressing and solving process update problems. Future solutions to the above four problems will be considered for inclusion in PQL by implementing and offering them to the users via the corresponding PQL statements.

6.2 Quality

Given a process model and a query, process querying solves a decision problem with a yes-or-no answer that indicates whether the model matches the query or not. The quality of such a decision is also binary; the decision is either correct or not. Process manipulation is different, as a requested manipulation can be fulfilled to various degrees. To compare methods for manipulating process models, either to

select a method to implement as part of PQL or to choose an already implemented method for triggering during PQL query execution, one should be able to measure and compare their quality in terms of the resulting models they produce. The quality of manipulated process models can be compared against different aspects. Several of these aspects are discussed below, giving rise to three research problems.

Simplicity problem. A process model that results from a solution to a process insertion, deletion, or update problem should be simple.

It may be necessary to manually analyze a process model that results from PQL manipulations, for example, to obtain feedback on the model from a process analyst or a domain expert. Hence, the manipulated models must be comprehensible. That is, they should be simple to understand for human readers. Simplicity is the desired quality for many artifacts automatically learned from data using data mining and process mining techniques. The simplicity criteria for learned models are often implemented as realizations of the Occam's Razor principle [11] that states that a model should use as few constructs as possible. Alternatively, this principle can be interpreted as if a model should not be overcomplicated without necessity. Consequently, existing simplicity criteria [13–15] from the field of process mining [2] can be reused to assess the simplicity of the manipulated by PQL models. The model simplicity criteria that will be developed in the future may consider the specifics of the process manipulation problems, refer to Section 6.1.

Resemblance problem. A process model that results from a solution to a process insertion, deletion, or update problem for a given process model should resemble the original model.

As PQL manipulations are applied over a given process model, it may be desirable that a resulting manipulated model resembles the original model. This desire, again, can stem from the potential necessity to assess manipulated models manually, this time in the context of the original model. Indeed, the user may know the model they request to manipulate and, consequently, expect that the resulting model is not radically different from the model they know, especially if the intended changes to the model are not extensive. This intention to keep resemblance with the original model is similar to the desire of repaired models, studied in process mining [2], to resemble the original models that were repaired. Thus, measures of model resemblance developed in the context of process manipulation can draw inspiration from the corresponding measures studied as part of the process repair problem [10, 20].

Correctness problem. A process model that results from a solution to a process insertion, deletion, or update problem should describe the requested processes.

A solution to a process manipulation problem, either an insertion, deletion, or an update problem, should construct a process model that describes a specific, requested collection of processes. However, methods for process manipulation can produce models that do not fulfill this correctness criterion; for instance, to avoid constructing complex models or models that do not resemble the input models. Various measures can be introduced to assess the correctness of manipulated models in terms of the

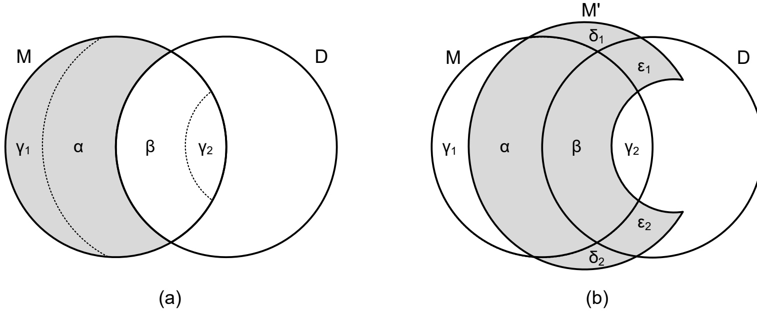

Fig. 6: A schematic visualization of the participating process collections in the context of a solution to the process deletion problem: (a) the problem definition and (b) a possible problem solution.

processes they describe. These measures can quantify and compare collections of processes that were requested and appeared, were requested and did not appear, were not requested and appeared, and were not requested and did not appear in the processes described by the manipulated model.

Fig. 6(a) visualizes an example process deletion problem schematically. Concretely, given a model that describes a collection of processes M, the problem requests to construct a model that describes processes captured in the input model but not in a collection of processes D. Hence, the resulting model should describe the collection of processes $M \setminus D$, denoted by the shaded region in the figure. In turn, Fig. 6(b) shows a collection of processes M' described by some model constructed as a solution to the problem superposed on the two process collections from Fig. 6(a). Several sets of processes emerge in this situation. Processes α are the processes that should and are described by the resulting model, while processes β are the processes that should not but are described by the resulting model. The resulting model does not describe processes γ_1 and γ_2. However, while processes γ_2 were correctly deleted, processes γ_1 should be present in M'. Processes $\delta_1 \cup \delta_2$ are not participating in the problem definition but are described by the resulting model. Finally, processes $\varepsilon_1 \cup \varepsilon_2$ were requested to be deleted, were not described by the input model, but ended up as described by the resulting model. A good solution to the process deletion problem should aim to minimize the sizes of sets γ_1, β, δ_1, δ_2, ε_1, and ε_2. The measures of the correctness of process manipulations should quantify this intuition to support the design of correct methods. Here, again, we can learn from the subarea of conformance checking [6, 26] in process mining [2], which studies ways to diagnose commonalities and discrepancies between processes.

Consider models 8 and 9 in Fig. 3 that can result from executing query Q6 presented in Section 2 on model 3 from Fig. 1. If we apply the reasoning from Fig. 6 to these two models, then for model 8 it holds that β is empty and $\gamma_1 = \mathtt{AG}(\mathtt{DED})^+$, where γ_1 is specified by a regular expression, while for model 9 sets β and γ_1 are

both empty. Hence, model 9 can be considered as a more correct, and hence a better, result of query Q6 than model 8.

6.3 Suitability

The suitability of a method refers to its quality of being appropriate for a purpose. Conducted empirical studies on the suitability of the current process querying methods guide their design and implementation.

To evaluate the suitability of the 4C behavioral predicates for the purpose of process querying and to identify the most relevant predicates to implement in PQL, we performed a user study [22]. In that study, we conducted semi-structured interviews with business analysts that actively work with process models. In the interviews, besides explaining the high-level design of PQL, we tested the understanding of twelve preselected 4C predicates and asked to evaluate their ability to fulfill the process querying tasks. The twelve predicates were selected to ensure they include, and combine in different ways, all the features of all the 4C predicates. Our questions to the stakeholders probed usefulness, importance, likelihood, and frequency of using the predicates in daily work. All the predicates were identified as suitable, while the six most relevant were implemented in PQL. These are the `CanOccur`, `AlwaysOccurs`, `Cooccur`, `Conflict`, `TotalCausal`, and `TotalConcurrent` predicates.

Process querying grounded in the collection of the 4C predicates, or any other set of similar predicates, has a fundamental limitation. A querying method that relies on a finite number of behavioral predicates can distinguish between a finite number of model classes [21], where any two models from the same class are considered equivalent by every query. The scenario-based process querying facilities of PQL extend its expressiveness [25]. PQL querying based on traces with wildcards, as explained in Section 3.1.2, can be used to express an intent to retrieve a model that describes processes that contain, or do not contain, *any* finite collection of processes and, thus, can be used to discriminate infinitely many models.

Future studies will strengthen the current results on the suitability of PQL for fulfilling process querying and process manipulation tasks.

6.4 Decidability and Efficiency

Karsten Wolf demonstrated that computations of all the 4C predicates currently implemented in PQL can be reduced, often via exponential space transformations, to model checking [32]. In that work of Karsten Wolf, the reduction of one 4C predicate, namely the *total existential concurrent* predicate, was left open, and its decidability, for the general case, is currently unknown. In addition, the proposed transformations for four 4C predicates are applicable only in the special case of the absence of auto-

concurrency in process models. Note that model checking over infinite-state systems is undecidable and is PSPACE-complete over finite-state systems [8], making it from challenging to impossible to evaluate the predicates at runtime. Hence, we precompute and store values of the predicates we can obtain in an index and access this index in close to real-time during the computation of PQL queries.

To perform scenario-based querying, that is, to check if a model describes a process that matches a sequence of actions with wildcards (see queries Q3 and Q4 in Section 2), first, the queried model gets transformed. The size of the transformed model is proportional to the size of the model and the scenario of interest. Then, an *optimal alignment* between the transformed model and a sequence of actions induced by the scenario of interest is constructed, and its cost is analyzed. The problem of computing an optimal alignment is equivalent to the reachability problem [3, 4], which is decidable [30] with the exponential space as the lower bound [17]. Despite its high computational complexity, the proposed method works in close to real-time on industrial and synthetic models [25]. To speed up query processing, we propose to use the untangling-based index [23] that allows identifying models that describe a process in which all actions from the scenario of interest occur. Then, further checks should be applied to verify if the actions occur in a requested order.

PQL queries that solve the process insertion problem are implemented using the impact-driven process model repair method [20]. Similar to scenario-based querying, the method relies on optimal alignments to compute queries. However, in this case, the alignments are used to identify the minimal required changes to the model to fulfill the query.

7 Conclusion

This chapter gives an overview of PQL, a domain-specific programming language for process querying and process manipulation. PQL is a declarative language with the SQL-like syntax. It is useful for managing process models stored in process repositories based on the processes that these models describe. Process querying is supported in PQL by means of the `SELECT` statements, while process manipulation is implemented using the `INSERT`, `DELETE`, and `UPDATE` statements. The chapter also discusses the currently supported features of the language, a publicly available implementation of a process repository with PQL support, future design and implementation efforts aimed at shaping the language, and open research problems triggered by the design of the language.

References

1. van der Aalst, W.M.P.: Formalization and verification of event-driven process chains. Inf. Softw. Technol. **41**(10), 639–650 (1999)
2. van der Aalst, W.M.P.: Process Mining—Data Science in Action, 2nd edn. Springer (2016)

3. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying history on process models for conformance checking and performance analysis. Data Min. Knowl. Discov. **2**(2), 182–192 (2012). DOI 10.1002/widm.1045
4. Adriansyah, A.: Aligning observed and modeled behavior. Ph.D. thesis, TU/e (2014). DOI 10.6100/IR770080. URL http://dx.doi.org/10.6100/IR770080
5. Billington, J., Christensen, S., van Hee, K.M., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The Petri net markup language: Concepts, technology, and tools. In: ICATPN, *LNCS*, vol. 2679, pp. 483–505. Springer (2003)
6. Carmona, J., van Dongen, B.F., Solti, A., Weidlich, M.: Conformance Checking - Relating Processes and Models. Springer (2018). DOI 10.1007/978-3-319-99414-7. URL https://doi.org/10.1007/978-3-319-99414-7
7. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Inf. Softw. Technol. **50**(12), 1281–1294 (2008)
8. Esparza, J., Nielsen, M.: Decidability issues for Petri nets—a survey. Bulletin of the EATCS **52**, 244–262 (1994)
9. Esparza, J., Römer, S., Vogler, W.: An improvement of McMillan's unfolding algorithm. Formal Methods in System Design (FMSD) **20**(3), 285–310 (2002)
10. Fahland, D., van der Aalst, W.M.: Model repair—aligning process models to reality. Information Systems (IS) **47**, 220–243 (2015). DOI 10.1016/j.is.2013.12.007. URL http://dx.doi.org/10.1016/j.is.2013.12.007
11. Grünwald, P.D.: The Minimum Description Length Principle (Adaptive Computation and Machine Learning). The MIT Press (2007)
12. Hack, M.: Decidability Questions for Petri Nets. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York (1975)
13. Kalenkova, A.A., Polyvyanyy, A., Rosa, M.L.: A framework for estimating simplicity of automatically discovered process models based on structural and behavioral characteristics. In: BPM, *Lecture Notes in Computer Science*, vol. 12168, pp. 129–146. Springer (2020)
14. Laue, R., Gruhn, V.: Complexity metrics for business process models. In: BIS, *LNI*, vol. P-85, pp. 1–12. GI (2006)
15. Lieben, J., Jouck, T., Depaire, B., Jans, M.: An improved way for measuring simplicity during process discovery. In: EOMAS@CAiSE, *Lecture Notes in Business Information Processing*, vol. 332, pp. 49–62. Springer (2018)
16. Lindland, O.I., Sindre, G., Sølvberg, A.: Understanding quality in conceptual modeling. IEEE Softw. **11**(2), 42–49 (1994)
17. Lipton, R.: The Reachability Problem Requires Exponential Space. Research report. Department of Computer Science, Yale University (1976)
18. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: Computer Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 663, pp. 164–177. Springer (1992)
19. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, Part I. Theoretical Computer Science (TCS) **13**, 85–108 (1981)
20. Polyvyanyy, A., van der Aalst, W.M.P., ter Hofstede, A.H.M., Wynn, M.T.: Impact-driven process model repair. ACM Trans. Softw. Eng. Methodol. **25**(4), 28:1–28:60 (2017)
21. Polyvyanyy, A., Armas-Cervantes, A., Dumas, M., García-Bañuelos, L.: On the expressive power of behavioral profiles. Formal Asp. Comput. **28**(4), 597–613 (2016). DOI 10.1007/s00165-016-0372-4. URL http://dx.doi.org/10.1007/s00165-016-0372-4
22. Polyvyanyy, A., ter Hofstede, A.H.M., Rosa, M.L., Ouyang, C., Pika, A.: Process query language: Design, implementation, and evaluation. CoRR **abs/1909.09543** (2019)
23. Polyvyanyy, A., La Rosa, M., ter Hofstede, A.H.M.: Indexing and efficient instance-based retrieval of process models using untanglings. In: CAiSE, *LNCS*, vol. 8484, pp. 439–456. Springer (2014)
24. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. Decision Support Systems **100**, 41–56 (2017). DOI 10.1016/j.dss.2017.04.011

25. Polyvyanyy, A., Pika, A., ter Hofstede, A.H.M.: Scenario-based process querying for compliance, reuse, and standardization. Inf. Syst. **93**, 101,563 (2020)
26. Polyvyanyy, A., Solti, A., Weidlich, M., Ciccio, C.D., Mendling, J.: Monotone precision and recall measures for comparing executions and specifications of dynamic systems. ACM Trans. Softw. Eng. Methodol. **29**(3), 17:1–17:41 (2020)
27. Polyvyanyy, A., Weidlich, M.: Towards a compendium of process technologies: The jBPT library for process model analysis. In: CAiSE Forum, *CEUR Workshop Proceedings*, vol. 998, pp. 106–113. CEUR-WS.org (2013)
28. Polyvyanyy, A., Weidlich, M., Conforti, R., Rosa, M.L., ter Hofstede, A.H.M.: The 4C spectrum of fundamental behavioral relations for concurrent systems. In: Petri Nets, *LNCS*, vol. 8489, pp. 210–232. Springer (2014)
29. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theoretical Computer Science (TCS) **6**, 223–231 (1978)
30. Reutenauer, C.: The Mathematics of Petri Nets. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1990)
31. Schmidt, K.: LoLA: A low level analyser. In: Application and Theory of Petri Nets (ICATPN), *Lecture Notes in Computer Science*, vol. 1825, pp. 465–474. Springer (2000)
32. Wolf, K.: Interleaving based model checking of concurrency and causality. Fundamenta Informaticae **161**(4), 423–445 (2018). DOI 10.3233/FI-2018-1709