Process Query Language: Design, Implementation, and Evaluation

Artem Polyvyanyy[a], Arthur H. M. ter Hofstede[b], Marcello La Rosa[a], Chun Ouyang[b], Anastasiia Pika[b]

[a]*School of Computing and Information Systems*
The University of Melbourne, Parkville, VIC, 3010, Australia
[b]*Queensland University of Technology*
2 George St., Brisbane City, QLD, 4000, Australia

Abstract

Organizations can benefit from the use of practices, techniques, and tools from the area of business process management. Through the focus on processes, they create process models that require management, including support for versioning, refactoring and querying. Querying thus far has primarily focused on structural properties of models rather than on exploiting behavioral properties capturing aspects of model execution. While the latter is more challenging, it is also more effective, especially when models are used for auditing or process automation. The focus of this paper is to overcome the challenges associated with behavioral querying of process models in order to unlock its benefits. The first challenge concerns determining decidability of the building blocks of the query language, which are the possible behavioral relations between process tasks. The second challenge concerns achieving acceptable performance of query evaluation. The evaluation of a query may require expensive checks in all process models, of which there may be thousands. In light of these challenges, this paper proposes a special-purpose programming language, namely Process Query Language (PQL) for behavioral querying of process model collections. The language relies on a set of behavioral predicates between process tasks, whose usefulness has been empirically evaluated with a pool of process model stakeholders. This study resulted in a selection of the predicates to be implemented in PQL, whose decidability has also been formally proven. The computational performance of the language has been extensively evaluated through a set of experiments against two large process model collections.

Keywords: Process querying, process repository, process model collection, process model, process instance, process, searching, retrieving, querying

Email addresses: `artem.polyvyanyy@unimelb.edu.au` (Artem Polyvyanyy), `a.terhofstede@qut.edu.au` (Arthur H. M. ter Hofstede), `marcello.larosa@unimelb.edu.au` (Marcello La Rosa), `c.ouyang@qut.edu.au` (Chun Ouyang), `a.pika@qut.edu.au` (Anastasiia Pika)

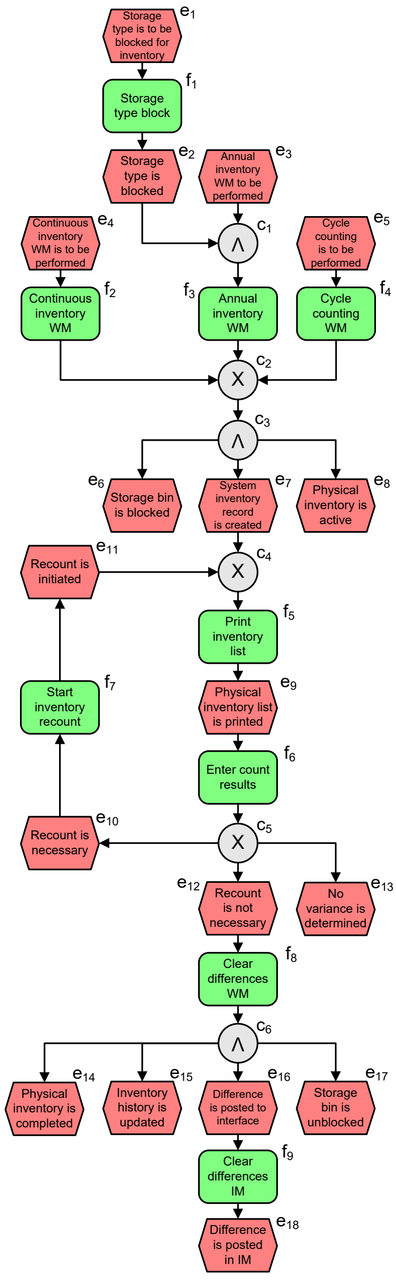
1. Introduction

Through the application of methods and techniques from the field of business process management, organizations can identify, model, analyze, redesign, automate, monitor, and query their business processes [1, 2, 3, 4]. This process-oriented thinking provides great benefits as making processes explicit through conceptual representations, i.e., process models, allows organizations to subject these processes to various forms of analysis, to use them as the basis for automated support, and to adapt them more easily as well as more rapidly to continual changes imposed by the organization's environment, both internal and external. As a consequence, some organizations have collected large numbers of process models. Examples of large process model collections reported in the literature are the SAP R/3 reference model (600+ models) [5], the IBM BIT collection in finance, telecommunication, and other domains (700+ models) [6], a collection of healthcare process models from a German health insurance company (4,000+ models) [7], and a collection of insurance process models from Suncorp—one of the largest insurance groups in Australia (6,000+ models) [8].

Process model collections evolve over time through model adaptations, mergers, and additions, and their maintenance poses significant challenges. To support these activities, it should be possible to query a potentially large repository of process models to retrieve models with specific characteristics. *Process querying* studies automated methods for managing, e.g., retrieving or manipulating, process models in process model repositories [4]. A *process querying method* is a technique that given a process model repository and a formal specification of an instruction to manage the repository, i.e., a *process query*, systematically implements the query in the repository [9].

Existing languages for specifying process queries, for example BP-QL [10] and BPMN-Q [11], predominantly focus on *structural* aspects of process models [12, 4]. Specifically, they treat process models as annotated directed graphs, where annotations denote types of graph nodes, e.g., process tasks, activities, events, and decisions. The semantics of such queries is based on paths and patterns in the process model graphs and is implemented via graph querying [13, 14]. An alternative way to query a process model repository is by using *behavioral relations* between model elements, e.g., relations which capture that process tasks can be performed in a particular order, in parallel, or can never be executed as part of the same *process instance*; we use the term process instance to refer to an artifact that encodes an execution of a process model, e.g., a sequence of tasks that can be executed according to the semantics of the process model. Process querying methods grounded in the querying of process model graphs are not suitable to adequately analyze behavioral relations induced by process models for at least two reasons. Firstly, behavioral relations are often defined over all process instances of a model. As a process model can describe infinitely many instances, it is challenging to specify a query that addresses all process instances of the model over the finite graph of this model. Secondly, infinitely many structurally different process models can describe the same behavioral relations [15, 16]. Consequently, it is challenging to specify a query that accounts for all possible structural patterns of models that can describe the relations.

The process model in Fig. 1 was extracted from the SAP R/3 reference model [5]. It is captured using the Event-driven Process Chain (EPC) language [17]. In this language, hexagons represent *events*, rounded rectangles encode *functions*, arrows capture the *control flow*, and circles represent *logical connectors*. The model in Fig. 1 should be retrieved as a response to the user's intent to find all process models in the repository that describe executions in which every occurrence of event "Physical inventory is active" (denoted by e_8 in the figure) *can* be triggered concurrently (at the same time) with *every* occurrence of function "Start inventory recount" (f_7), and *all* occurrences of function f_7 precede, or are the cause for, *all* occurrences of functions similar to "Clear differences", assuming that function names "Clear differences WM" (f_8) and "Clear differences IM" (f_9) are considered to be sufficiently similar to "Clear differences". The model describes infinitely many instances in which the functions and event from the query occur; the infinite number of instances is due to the ability to repeat function f_7 infinitely often by following the cycle in the control flow of the model. Consequently, it is not immediate to implement the query via querying over the model graph, as infinitely many graph walks must be checked. Moreover, Fig. 1 is one of infinitely many ways the functions and event from the query can be composed in a model to ensure the behavioral relations from the query hold. For example, the many potential occurrences of function f_7 in the model in Fig. 1 stem from the cyclic control flow. However, in another model that matches the example query, these occurrences may be due to the occurrences of different functions that



Storage type is to be blocked for inventory
e1
Storage type block
f1
Storage type is blocked
e2
Annual inventory WM to be performed
e3
Continuous inventory WM is to be performed
e4
C1
Cycle counting is to be performed
e5
Continuous inventory WM
f2
f3
Annual inventory WM
Cycle counting WM
f4
C2
X
C3
e6
Storage bin is blocked
System inventory record is created
e7
Physical inventory is active
e8
Recount is initiated
e11
C4
X
Print inventory list
f5
Start inventory recount
f7
Physical inventory list is printed
e9
Enter count results
f6
Recount is necessary
e10
C5
X
e12
Recount is not necessary
No variance is determined
e13
Clear differences WM
f8
C6
Physical inventory is completed
e14
Inventory history is updated
e15
Difference is posted to interface
e16
Storage bin is unblocked
e17
Clear differences IM
f9
Difference is posted in IM
e18

Fig. 1: An EPC model from SAP R/3.

all have the same name of "Start inventory recount". As, in general, one can construct a model with any number of such functions, it is not immediate to anticipate the number of different walks that need to be checked in the process model graph when designing the query.

The added expressiveness of a query language grounded in behavioral relations comes at a price. Behavioral relations cover a broad spectrum of inter-task dependencies that may be captured using special property specification languages, e.g., temporal logics. Temporal logics are powerful enough to be able to express properties that are undecidable [18, 19]. Hence, a query language that exploits behavioral relations needs to be carefully designed to support behavioral inter-task dependencies that can be computed.

While some behavioral relations are decidable, their use may not be intuitive to the stakeholders of the query language, i.e., the business analysts that will end up formulating queries. Thus, it is important that a relation is likely to be frequently used in queries in practice to warrant its inclusion in the language and that its formal meaning is close to its perceived meaning. Another consideration is that query evaluations are performed in a "reasonable" amount of time. In fact, it is anticipated that process model stakeholders may wish to see the answers to their queries in (almost) real-time, to quickly evaluate different scenarios when updating existing or creating new models.

This paper proposes Process Query Language (PQL)—a special-purpose programming language for querying repositories of process models based on *process instances* that these models describe. PQL programs are called *queries*. PQL allows formulating queries for retrieving process models from repositories using a selected number of behavioral inter-task relations, called *predicates*. The PQL predicates are built upon the 4C spectrum [20]—a systematic classification of possible behavioral relations between process model tasks according to four categories: conflict, co-occurrence, causality, and concurrency.

PQL implements the *process querying compromise* [21] by supporting decidable, efficiently computable queries whose *practical relevance* has been confirmed by practitioners in terms of the perceived usefulness, importance, and frequency of use. We demonstrate that PQL predicates are *decidable* by reducing their computations to the reachability problem [22], the covering problem [23], or the problem of structural analysis over a complete prefix [24, 25] of the unfolding [26] of the model. Although PQL predicates are computationally demanding, e.g., solving the reachability problem required exponential space [27], the conducted experiments demonstrate the feasibility of using PQL in practical settings.

To facilitate query formulation, PQL is provided with the abstract syntax and a concrete syntax, the latter inspired by the SQL language. To tackle the performance problem typical for checking behavioral properties, the implemented PQL runtime environment relies on the use of indexed behavioral relations, i.e., behavioral relations get precomputed and reused during the evaluation of PQL queries. The performance of PQL query evaluation is assessed through extensive experiments with real-life and synthetic process model collections.

In summary, the contributions of this paper are as follows:

- Empirical evidence for the appropriateness of behavioral process querying, i.e., the quality of behavioral process querying to be a proper way for retrieving process models based on behavioral inter-task relations;

- A selection of empirically justified behavioral inter-task relations for behavioral process querying based on quantitative feedback from prospective users;
- Design of a query language, viz. PQL, based on the selected behavioral inter-task relations and qualitative feedback from prospective users;
- An open-source implementation of a runtime environment for PQL queries;
- A performance evaluation of the PQL implementation that confirms the feasibility of running PQL queries in close to real-time over industrial repositories;
- A procedure for deciding whether two tasks are in the `TotalConcurrent` behavioral relation, one of the empirically justified PQL predicates whose decidability was never discussed in the literature.

These contributions build on and significantly extend our prior work. The abstract syntax of PQL is inspired by that of A Process-model Query Language (APQL) [28]. Furthermore, the dynamic semantics of PQL is grounded in the behavioral relations of the 4C spectrum [20]; this is the first time these relations are proposed for process querying. The denotational semantics of PQL, differently from APQL, is defined over occurrences of names, or labels, of process elements in process instances, like names of functions and events in EPCs in the example query above, while APQL addresses querying over element occurrences. Hence, PQL queries address querying over the meaning of the processes abstracting from how they are implemented in models. In addition, for the first time, the behavioral predicates included in PQL were selected based on the results of an empirical study with the stakeholders of the language. The declarative approach for process querying presented in this article complements the scenario-based querying [29] supported by PQL. The scenario-based querying allows specifying query conditions that address querying of imperative process descriptions. Neither can it implement the behavioral predicates proposed in this article, nor can the predicates implement scenario-based querying.

The next section motivates PQL by discussing several example PQL queries. Section 3 introduces basic notions that are used to support the subsequent discussions of the denotational semantics of PQL queries. Next, Section 4 discusses the results of an empirical study that suggest a selection of behavioral predicates for querying process models. Section 5 presents the PQL language. Section 6 is devoted to our implementation of PQL, while Section 7 reports results of an evaluation of this implementation using one industrial and one synthetic process model collection. Finally, Section 8 reviews related work, whereas Section 9 states concluding remarks.

2. Motivating Examples

This section motivates PQL by introducing its key elements via three example queries. The model in Fig. 1 sets the context for the examples; the queries refer to labels of functions and events in the model. To support the discussions, Table 1 lists six behavioral relations over functions and events of the model and the corresponding behavioral predicates that denote the relations. The

Table 1: Behavioral relations over functions and events of the model in Fig. 1 and corresponding behavioral predicates.

No.	Behavioral relations	Behavioral predicates
1	In *every* process instance, *every* occurrence of event "Physical inventory is active" (e_8) can be triggered concurrently with *every* occurrence of function "Start inventory recount" (f_7)	`TotalConcurrent(`e_8`,`f_7`)`
2	In *every* process instance, *all* occurrences of function "Start inventory recount" (f_7) precede *all* occurrences of functions "Clear differences WM" (f_8) and "Clear differences IM" (f_9)	`TotalCausal(`f_7`,`f_9`)`
3	Function "Print inventory list" (f_5) occurs in *every* process instance	`AlwaysOccurs(`f_5`)`
4	Either *both* or *neither* of the events "Storage type is blocked" (e_2) and "Annual inventory WM to be performed" (e_3) occur in a process instance	`Cooccur(`e_2`,`e_3`)`
5	Function "Start inventory recount" (f_7) can occur in *at least one* process instance	`CanOccur(`f_7`)`
6	Event "No variance is determined" (e_{13}) and function "Clear differences WM" (f_8) *never* occur together in a process instance	`Conflict(`e_{13}`,`f_8`)`

predicates are used in the example queries as underlying constructs of PQL. The precise definitions of the predicates are postponed till Section 4.1.

Example 1. Recall the example query from the Introduction: Retrieve all process models that describe executions in which every occurrence of event e_8 can be triggered concurrently with every occurrence of function f_7 (see predicate 1 in Table 1), and all occurrences of function f_7 precede all occurrences of every function with a label similar to "Clear differences", like functions f_8 and f_9 in Fig. 1 (see predicate 2). This query (*Q1*) can be captured in PQL as follows:

```
SELECT "ID" FROM "/SAP-R3-EPC-Repo"
WHERE TotalConcurrent("Physical inventory is active","Start inventory recount")
      AND TotalCausal("Start inventory recount",~"Clear differences");
```

This query retrieves models and their `ID`'s from location `"/SAP-R3-EPC-Repo"` in the process model repository. Note that `~"Clear differences"` in the PQL query above specifies a *task* (either an event or a function in an EPC model) with a label that is similar to `"Clear differences"`.

Example 2. The user wants to retrieve the models, with their `ID`'s and titles, where at least one of the functions "Continuous inventory WM" (f_2), "Annual inventory WM" (f_3), and "Print inventory list" (f_5) occurs in every instance (see predicate 3 in Table 1); or either both or neither of the events "Storage type is blocked" (e_2) and "Annual inventory WM to be performed" (e_3) occur in a process instance (predicate 4). The PQL query (*Q2*) below captures this intent.

```
SELECT "ID","Title" FROM "/SAP-R3-EPC-Repo"
WHERE AlwaysOccurs({"Continuous inventory WM","Annual inventory WM",
                    "Print inventory list"},ANY)
      OR Cooccur("Storage type is blocked","Annual inventory WM to be performed");
```

Query *Q2* contains one macro, see `AlwaysOccurs(`$\{f_2, f_3, f_5\}$`,ANY)`, which combines results of three behavioral predicates connected via logic OR operators, namely `AlwaysOccurs(`f_2`) OR AlwaysOccurs(`f_3`) OR AlwaysOccurs(`f_5`)`.

Example 3. The user wants to retrieve the models and all their attributes (e.g., titles, versions, and authors) that satisfy conditions *C1*–*C4* listed below.

(*C1*) Function "Start inventory recount" (f_7), event "No variance is determined" (e_{13}), function or event with a label similar to "Clear differences", such as function "Clear differences WM" (f_8) or function "Clear differences IM" (f_9), and function or event with a label similar to "Difference is posted", such as event "Difference is posted to interface" (e_{16}) or event "Difference is posted in IM" (e_{18}), can occur in some process instances;
(*C2*) Inventory recount is optional, i.e., function "Start inventory recount" (f_7) does not occur in at least one process instance;
(*C3*) There is no process instance in which function or event with a label similar to "Clear differences", such as functions "Clear differences WM" (f_8) and "Clear differences IM" (f_9), and event "No variance is determined" (e_{13}) both occur;
(*C4*) All occurrences of function "Start inventory recount" (f_7) precede all occurrences of functions or events with labels similar to "Clear differences", such as functions "Clear differences WM" (f_8) and "Clear differences IM" (f_9), and "Difference is posted", such as events "Difference is posted to interface" (e_{16}) and "Difference is posted in IM" (e_{18}).

The PQL query (*Q3*) listed below captures this user's intent.

```
x = {"Start inventory recount","No variance is determined"};
y = {~"Clear differences"};
z = y UNION {~"Difference is posted"};
w = GetTasksAlwaysOccurs(GetTasks());
SELECT * FROM "/SAP-R3-EPC-Repo"
WHERE CanOccur(x UNION z,ALL) AND                           -- (C1)
      (NOT ("Start inventory recount" IN w)) AND            -- (C2)
      Conflict("No variance is determined",y,ALL) AND       -- (C3)
      TotalCausal("Start inventory recount",z,ALL);         -- (C4)
```

In PQL, the user can use variables to store sets of *tasks* (for example, functions and events in the case of EPCs). In query *Q3*, variable x stores function f_7 and event e_{13}, y contains one task that refers to functions f_8 and f_9, and variable z stores the result of combining the task stored in y and the task that refers to events e_{16} and e_{18}; note that these definitions of variables are valid if the query is matched to the model in Fig. 1. Variable w stores the tasks that occur in every process instance of the model matched to the query. Note that `GetTasks()` retrieves all the tasks of the model and `GetTasksAlwaysOccurs()` selects from the input those tasks that occur in every process instance of the model.

The `WHERE` clause of query *Q3* captures the four conditions (*C1* to *C4*) of the user's query intent, as marked in the comments (starting with '--'). Firstly, predicate macro `CanOccur(`x` UNION `z`, ALL)` checks if every task in the set of tasks combined from x and z occurs in at least one process instance; e.g., `CanOccur(`f_7`)`, see predicate 5 in Table 1, is part of the check. Next, predicate `(NOT ("Start inventory recount" IN `w`))` checks if the occurrence of function f_7 is optional. Then, predicate macro `Conflict("No variance is determined",`y`,ALL)` checks if event e_{13} occurs in conflict with each of the tasks stored in variable y; e.g., it is necessary to check `Conflict(`e_{13}`,`f_8`)`, see predicate 6 in Table 1. Finally, predicate macro `TotalCausal("Start inventory recount",`z`,ALL)` checks if all occurrences of function f_7 precede all occurrences of every task stored in variable z; e.g., `TotalCausal(`f_7`,`f_9`)` is one of the required checks, see predicate 2 in Table 1.

3. Preliminaries

This section introduces basic notions. Specifically, Section 3.1 gives background on business process modeling. Then, Section 3.2 introduces multisets, sequences, and strings. Section 3.3 presents Petri net systems. Finally, Section 3.4 discusses processes and unfoldings of Petri net systems.

3.1. Business Process Modeling

Though the motivating examples discussed in Section 2 address querying of EPC models, the execution semantics of PQL is grounded in Petri net systems. As the semantics of many process modelings languages, including EPC [30], Business Process Execution Language (BPEL) [31], Yet Another Workflow Language (YAWL) [32], and Business Process Model and Notation (BPMN) [33] is formalized via mappings to Petri net systems, PQL can be applied over models captured in these languages.

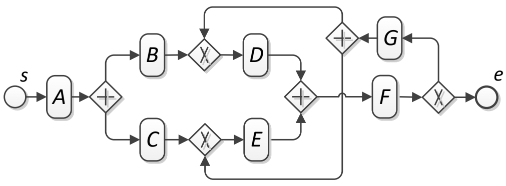


Fig. 2: A BPMN model.

For example, in a BPMN model, activities model process tasks and are drawn as rectangles with rounded corners. Gateways are visualized as diamonds. Exclusive gateways use a marker that is shaped like “×” inside the diamond shape, whereas parallel gateways use a marker that is shaped like “+” inside the diamond shape. Directed arcs encode control flow dependencies. Fig. 2 shows an example BPMN model. For simplicity, the model uses abstract task labels.

3.2. Multisets, Sequences, and Strings

Let A be a set. By $\mathcal{P}(A)$ and $\mathcal{B}(A)$, we denote the power set of A and the set of all finite multisets over A, respectively. We define a multiset $B \in \mathcal{B}(A)$ as a function $B : A \to \mathbb{N}_0$, where $\mathbb{N}_0$ denotes the set of all natural numbers including zero. For some multiset $B \in \mathcal{B}(A)$, $B(a)$ is the multiplicity of element $a \in A$ in B, i.e., the number of times element a appears in multiset B.

By $\sigma := \langle a_1, a_2, \ldots, a_n \rangle \in A^*$, we denote a sequence of length $n \in \mathbb{N}_0$ over a set A, $a_i \in A$, $i \in [1\,..\,n]$, where A^* denotes the Kleene star operation on set A.

The *empty* sequence is denoted by $\langle\rangle$. By $|\sigma|$, we indicate the number of all occurrences of elements in σ. By $\mathit{prefix}(\sigma, i)$, we denote the prefix of σ up to but excluding position i, whereas $\mathit{suffix}(\sigma, i)$ is the suffix of σ starting from and including position i, $i \in \mathbb{N}$. Let $\eta := \langle \mathtt{a}, \mathtt{b}, \mathtt{a}, \mathtt{b}, \mathtt{a}, \mathtt{h}, \mathtt{a}, \mathtt{l}, \mathtt{a}, \mathtt{m}, \mathtt{a}, \mathtt{h}, \mathtt{a} \rangle$ be a sequence. Then, $|\eta| = 13$, $\mathit{prefix}(\eta, 6) = \langle \mathtt{a}, \mathtt{b}, \mathtt{a}, \mathtt{b}, \mathtt{a} \rangle$, and $\mathit{suffix}(\eta, 6) = \langle \mathtt{h}, \mathtt{a}, \mathtt{l}, \mathtt{a}, \mathtt{m}, \mathtt{a}, \mathtt{h}, \mathtt{a} \rangle$.

An *alphabet* is a non-empty finite set. The elements of an alphabet are *characters.* A character *string* over an alphabet is a finite sequence of characters from the alphabet. The characters of a string are usually written next to one another. For example, $q := \mathtt{101011}$ is a string over $\{\mathtt{0}, \mathtt{1}\}$. The character string of length zero is called the *empty string* and is denoted by ϵ. By $\mathbb{C}$, we denote the universe of all finite character strings over letters of the English alphabet (both lower- and uppercase), numerals, punctuation, and whitespace characters.

3.3. Petri Net Systems, Workflow Systems, and Soundness

A Petri net system is a model of a distributed system [34].

Definition 3.1 (Petri net systems).
A *Petri net system*, or a *system*, is a 5-tuple $S := (P, T, F, \lambda, M)$, where P and T are finite disjoint sets of *places* and *transitions*, respectively, $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*, $\lambda : T \to \mathbb{C}$ is a *labeling function* that assigns labels to transitions, and $M \in \mathcal{B}(P)$ is a *marking* of S. ⌟

Transitions of a Petri net system encode *activities.* If $\lambda(t) \neq \epsilon$, $t \in T$, then t is *observable*; otherwise, t is *silent.* An element $x \in P \cup T$ is a *node* of S. A node x is an *input* of a node y iff $(x, y) \in F$, while a node y is an *output* of x. By $\bullet x$, we denote the *preset* of node x, i.e., the set of all input nodes of x, while by $x\bullet$, we denote the *postset* of x, i.e., the set of all output nodes of x.

A marking of a system encodes its state. A marking M is often interpreted as an assignment of *tokens* to places, i.e., marking M 'puts' $M(p)$ tokens at place p, $p \in P$. The execution semantics of a Petri net system is defined in terms of possible states and state transitions. Let $S := (P, T, F, \lambda, M)$ be a system. A transition $t \in T$ is *enabled* in M, denoted by $M[t\rangle$, iff every input place of t contains at least one token, i.e., $\forall p \in \bullet t : M(p) > 0$. If a transition $t \in T$ is enabled in M, then t can *occur*, which *leads to* a fresh marking $M' := (M \setminus \bullet t) \uplus t\bullet$ of S, i.e., transition t *consumes* one token from every input place of t and *produces* one token at every output place of t. By $M[t\rangle M'$, we denote the fact that an occurrence of t *leads from M to M'*. A finite sequence of transitions $\sigma := \langle t_1, t_2, \ldots, t_n \rangle \in T^*$, $n \in \mathbb{N}_0$, is an *occurrence sequence* of S iff σ is empty or there is a sequence of markings $\langle M_0, M_1, \ldots, M_n \rangle$, such that $M_0 = M$ and for every $i \in [1..n]$ it holds that $M_{i-1}[t_i\rangle M_i$; σ *leads from M_0 to M_n* denoted by $M_0[\sigma\rangle M_n$.

Petri net systems have an established graphical notation. In this notation, places are visualized as circles, transitions are drawn as rectangles with their labels depicted within the boundaries of the corresponding rectangles, every pair of nodes (x, y) in the flow relation is drawn as a directed arc from x to y, and tokens are depicted as black dots inside the assigned places.

A *workflow system* is a system with one *source* place, one *sink* place, every node on a directed path from the source to the sink, and marking that puts one token at the source place and no tokens elsewhere [35].

Fig. 3 shows a workflow system that encodes the execution semantics of the model in Fig. 1. It was obtained by first translating the model in Fig. 1 into a Petri net system [36] and then completing the system to a *workflow system* [37, 38]. The fresh elements introduced during the completion are highlighted in gray, while the fresh arcs, in addition, are drawn using dashed lines. In Fig. 3, transitions are labeled with short names while the full names are given in Fig. 1. For example, transitions with labels f_7 and e_9 represent function "`Start inventory recount`" and event "`Physical inventory list is printed`" in the EPC model, respectively. Transitions t_1, t_2, t_3, t_4, t_5, and t_6 introduced during the completion, as well as transitions c_1, c_3 and c_6 that correspond to the logical AND connectors in Fig. 1, are silent, while all the other transitions in Fig. 3 are observable.

Every occurrence sequence that leads to the marking that puts one token at the sink place and no tokens elsewhere is its execution.

Definition 3.2 (Executions).
An *execution* of a workflow system $S := (P, T, F, \lambda, [i])$ with the source place $i \in P$ is an occurrence sequence of S that leads to $[o]$, where $o \in P$ is the sink place of S. ⌟

By $\mathbb{E}_S$, we denote the set of all and only executions of workflow system S. Let $\sigma := \langle t_1, t_2, \ldots, t_n \rangle \in \mathbb{E}_S$, $n \in \mathbb{N}_0$, be an execution of S. Then, sequence $\alpha := \langle \lambda(t_1), \lambda(t_2), \ldots, \lambda(t_n) \rangle$ is the *label execution* of S induced by σ. The sequences of transitions $\langle t_2, e_1, f_1, e_3 \rangle$ and $\langle t_3, e_5, f_4, c_3, e_7, f_5, e_9, e_8, f_6, e_6, e_{13}, t_5, t_6 \rangle$ are two example occurrence sequences of the system in Fig. 3, whereas the latter is also its execution. Note that $\mathbb{E}_S$, where S is the workflow system in Fig. 3, is an infinite set of executions.

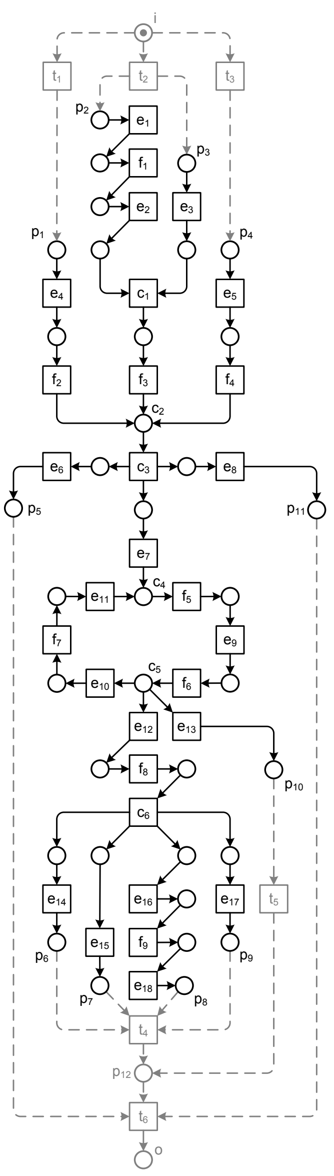


Fig. 3: A workflow system.

Petri net systems and workflow systems are subject to semantic correctness constraints. One widely-used semantic correctness criterion for workflow systems is *soundness* [35]. In a sound workflow system, every occurrence sequence can be extended (via occurrences of its enabled transitions) to an execution of the system, and every transition of the system is an element of at least one execution. For example, the workflow system in Fig. 3 is sound.

3.4. Processes and Unfoldings

Executions of workflow systems capture orderings of transition occurrences. One can rely on *processes* to adequately represent *causality* and *concurrency* relations on transition occurrences [39]. By R^+, we denote the transitive closure of a binary relation R. Let $f(X) := \{f(x) \mid x \in X\}$ and $f^{-1}(z) := \{y \in Y \mid f(y) = z\}$, where X is a subset of f's domain Y.

Definition 3.3 (Processes).
A *process* of a system $S := (P, T, F, \lambda, M)$ is a 4-tuple $\pi := (B, E, G, \rho)$, where B is a set of *conditions*, E is a set of *events*, $G \subseteq (B \times E) \cup (E \times B)$ is the *flow relation*, such that G^+ is irreflexive and $\forall\, b \in B : |\{e \in E \mid (e,b) \in G\}| \leq 1 \;\wedge\; |\{e \in E \mid (b,e) \in G\}| \leq 1$, and $\rho : B \cup E \to P \cup T$ is such that:

- $\rho(B) \subseteq P$ and $\rho(E) \subseteq T$, i.e., ρ preserves the nature of nodes,
- $\forall\, b_1, b_2 \in Min(\pi) : (b_1, b_2) \notin G^+ \wedge (b_2, b_1) \notin G^+$, $\forall\, b_1 \in B \setminus Min(\pi) \exists\, b_2 \in Min(\pi) : (b_2, b_1) \in G^+$, and $\forall p \in P : M(p) = |\rho^{-1}(p) \cap Min(\pi)|$, i.e., π starts at M, and
- for every event $e \in E$ and for every place $p \in P$ it holds that $|\{(p,t) \in F \mid t = \rho(e)\}| = |\rho^{-1}(p) \cap \bullet e|$ and $|\{(t,p) \in F \mid t = \rho(e)\}| = |\rho^{-1}(p) \cap e \bullet|$, i.e., ρ respects the environment of transitions.

Note that $Min(\pi) := \{b \in B \mid \forall e \in E : (e,b) \notin G\}$; $\bullet e := \{b \in B \mid (b,e) \in G\}$ and $e \bullet := \{b \in B \mid (e,b) \in G\}$. ⌟

Therefore, a process of a Petri net system is an acyclic bipartite graph, in which conditions and events are two disjoint sets of nodes, with a mapping from nodes of the process to nodes of the system. Fig. 5 shows three processes of the system in Fig. 4(a). Conditions and events of a process are drawn as places and transitions, respectively. The labels in the figures encode mappings of nodes of processes to nodes of the system. In particular, each condition b_i maps to place p_i, $i \in [1..7]$, and each event e_j maps to transition t_j, $j \in [1..6]$. In general, several conditions (events) of a process can refer to the same place (transition) of the corresponding system. For example, there exist processes of the system in Fig. 3 in which several conditions and several events refer to the same place and transition, respectively. These are the processes that describe multiple occurrences of transitions in the loop with entry place c_4 and exit place c_5.

Given a process π of a system S, by E_π and ρ_π we refer to events of π and the function that maps nodes of π to nodes of S, respectively. A process π of S can be interpreted as a collection of occurrence sequences of S, where every event $e \in E_\pi$ describes an occurrence of transition $\rho_\pi(e)$. For example, the process in Fig. 4(b) describes a single occurrence sequence of the system in Fig. 4(a), namely $\langle t_1, t_3, t_5 \rangle$, whereas the process in Fig. 4(c) describes two occurrence sequences of the system in Fig. 4(a), namely $\langle t_1, t_3, t_4, t_5, t_6 \rangle$ and $\langle t_1, t_3, t_5, t_4, t_6 \rangle$. Given a workflow system S, by Π_S we denote the set that contains all and only processes of S (up to isomorphism) that describe all the executions of S. Note that the set of all processes of a workflow system can be infinite. For example, the set of all processes of the system in Fig. 3 that describe all its executions is infinite. Let S be the workflow system in Fig. 4(a).

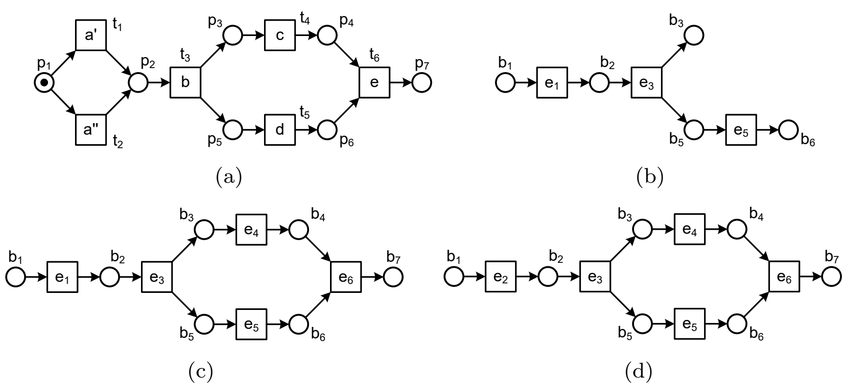


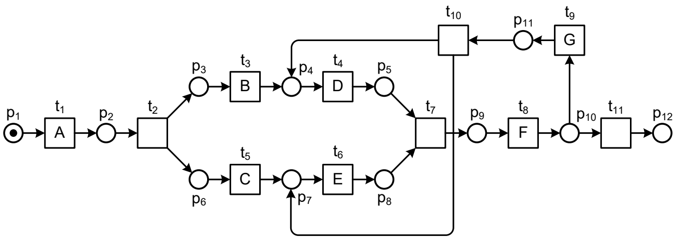
Fig. 4: (a) A workflow system, and (b)–(d) its three processes.

Then, Π_S is composed of the two processes shown in Figs. 4(c) and 4(d); these two processes describe all four executions of the system. Note that the process from Fig. 4(b) is not in Π_S because the occurrence sequence it describes is not an execution of S. Finally, by $\Delta_S(x,y)$, where $x,y \in \mathbb{C}$, we refer to the set $\{\pi \in \Pi_S \mid \exists\, e_1, e_2 \in E_\pi : \lambda(\rho_\pi(e_1)) = x \,\wedge\, \lambda(\rho_\pi(e_2)) = y\}$, i.e., the set that consists of every process in Π_S that contains an event that describes an occurrence of a transition with label x and an event that describes an occurrence of a transition with label y. Processes of systems can be characterized by the causality and concurrency relations over their nodes [26, 39]. An event $e_1 \in E_\pi$ *causes* event $e_2 \in E_\pi$ of a process π, denoted by $e_1 \rightarrowtail_\pi e_2$, iff $(e_1, e_2) \in G^+$. Two events $e_1 \in E_\pi$ and $e_2 \in E_\pi$ are *concurrent* in π, denoted by $e_1 \parallel_\pi e_2$, iff $(e_1, e_2) \notin G^+$ and $(e_2, e_1) \notin G^+$. Intuitively, the fact that event e_1 is a cause for event e_2 means that one has to observe an occurrence of transition described by e_1 prior to observing an occurrence of transition described by e_2. The fact that two events are concurrent means that the corresponding transitions can be enabled simultaneously in some occurrence sequence of the system and be performed one after another in any order. For example, for process π in Fig. 4(d) it holds that $e_2 \rightarrowtail_\pi e_6$, $e_3 \rightarrowtail_\pi e_4$, and $e_5 \parallel_\pi e_4$.

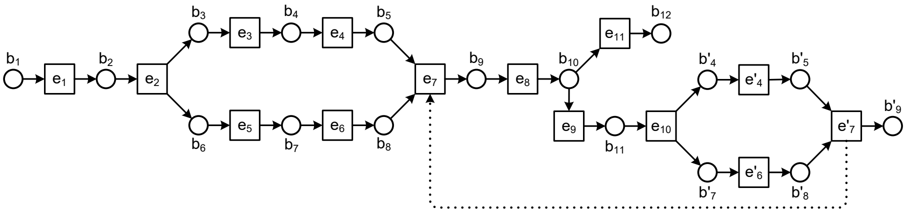
The *unfolding* of a system is a possibly infinite acyclic graph that encodes all the processes of the system [24, 25]. In [24], McMillan proposed an algorithm for constructing a finite initial part of the unfolding, called a *complete prefix* of the unfolding, which contains full information about all the processes of the system.

Fig. 5(a) shows a workflow system S obtained by translating the BPMN model in Fig. 2 to a Petri net system, whereas Fig. 5(b) shows a complete prefix of the unfolding U of S. In general, a complete prefix of the unfolding of a system $S := (P, T, F, \lambda, M)$ is a 4-tuple $U := (B, E, G, \rho)$, where B and E are disjoint sets of conditions and events, respectively, $G \subseteq (B \times E) \cup (E \times B)$ is the flow relation, such that G^+ is irreflexive, and $\rho : B \cup E \rightarrow P \cup T$ is a function that maps conditions to places and events to transitions.

In Fig. 5(b), conditions and events are shown as circles and rectangles, respectively. Every condition $b_i, b'_i, \ldots$ represents one token at place p_i, $i \in \mathbb{N}$ and every event $e_j, e'_j, \ldots$ represents one occurrence of transition t_j, $j \in \mathbb{N}$ [40]. McMillan proposes to associate every event e of a complete prefix of the unfolding



(a)



(b)

Fig. 5: (a) A workflow system, and (b) a complete prefix of its unfolding.

of a Petri net system with a marking that one reaches by firing all the transitions encoded by the events in the local configuration of e. The *local configuration* of an event e, denoted by $\lceil e \rceil$, is the set of events composed of e, all the events from which there is a directed path to e, and no other events. For example, $\lceil e_4 \rceil = \{e_1, e_2, e_3, e_4\}$ and $\lceil e_7 \rceil = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$. One reaches the marking $[p_5, p_6]$ by firing transitions represented by the events in $\lceil e_4 \rceil$, and the marking $[p_9]$ by firing transitions represented by the events in $\lceil e_7 \rceil$. These markings are encoded by the sets of conditions $\{b_5, b_6\}$ and $\{b_9\}$ in Fig. 5(b), respectively, and are denoted by $Cut(\lceil e_4 \rceil)$ and $Cut(\lceil e_7 \rceil)$, respectively. Note that the marking $[p_9]$ can also be reached in the system in Fig. 5(a) after occurrences of transitions encoded by the events in the local configuration of event e'_7.

If the construction of the prefix in Fig. 5(b) continues, the part that follows $Cut(\lceil e'_7 \rceil)$, will be isomorphic to the part of the prefix that follows $Cut(\lceil e_7 \rceil)$. Thus, McMillan proposes to stop the construction of the prefix at $Cut(\lceil e'_7 \rceil)$ and refers to e'_7 and e_7 as a *cutoff* and its *corresponding* event, respectively. We denote the set of all the cutoff events of a complete prefix of the unfolding U by $\mathit{cutoffs}(U)$. Given a cutoff event e, by $\mathit{corr}(e)$, we denote the corresponding event of e. Thus, $\mathit{cutoffs}(U)$, where U is a complete prefix of the unfolding in Fig. 5(b), is equal to $\{e'_7\}$, and $\mathit{corr}(e'_7) = e_7$. In the figure, the relation between the cutoff and its corresponding event is shown by the dotted arrow.

4. Behavioral Predicates

This section presents the results of an empirical study with process modeling experts.[1] The results confirm that process querying based on behavioral inter-task

[1]The study has been granted approvals on behalf of the University Human Research Ethics Committee, Queensland University of Technology, Australia, Ref. No. 1000001158, and the

relations is an appropriate method for retrieving process models from repositories and suggest a selection of basic behavioral predicates for inclusion in PQL. The section introduces a repertoire of behavioral predicates from our previous work [20], refer to Section 4.1, presents the design of our empirical experiment, Section 4.2, and summarizes the results of the experiment, Section 4.3.

4.1. Relations and Predicates

A *behavioral relation* over process tasks in a process model specifies an ordering constraint for occurrences of the tasks in the instances described by the model. It has been shown that there are four fundamental categories of binary behavioral relations over process tasks: *conflict*, *causality*, *concurrency*, and *co-occurrence*. These four categories of relations can be used to fully characterize any constraints over occurrences of process tasks [26, 41, 42]. An occurrence of a process task implies that all its causally dependent tasks have already occurred and none of the conflicting tasks has been or will be observed, whereas two concurrent tasks can be enabled for simultaneous execution [43]. Finally, co-occurrence describes two process tasks that both occur in the same instance of a process model. Note that the behavioral relations on tasks describe how they can be executed and not how they are semantically related. For example, two tasks in the *conflict* relation can never appear in the same instance of the model, not to be confused with semantic interference studied in Stroop experiments which look into how different concepts may conflict and, thus, complicate the understanding of a phenomenon [44].

The *4C spectrum* [20] is a systematic classification of behavioral relations grounded in the four categories of conflict, co-occurrence, causality, and concurrency. The spectrum classifies the relations at different levels of granularity. For example, given two process tasks, different relations from the spectrum assess whether in *all* or *some* instances of the model, *all* or *some* occurrences of one task are in a particular relation with all or some occurrences of the other task. For example, one of the 4C relations specifies that all occurrences of task A are concurrent to all occurrences of task B in all instances of a model, while another relation assesses whether at least one occurrence of A is concurrent to at least one occurrence of B in at least one instance of the process model. Given the three degrees of freedom, i.e., process instances, occurrences of the first task, and occurrences of the second task, and two parameters for each degree, i.e., all or some, there are eight granularities of a particular behavioral relation.

We assessed twelve 4C spectrum predicates for their relevance for querying process model collections. The selected predicates are listed in Table 2. Three factors guided the selection of the predicates. The selected predicates must cover all four behavioral categories. This decision should allow checking which categories of behavioral predicates are more relevant for process querying. The selected predicates must cover predicates of different granularity. This factor should allow verifying predicates of which "strength" are more relevant for

Human Ethics Advisory Group, the University of Melbourne, Australia, Ref. No. 1851972.

Table 2: Twelve behavioral predicates. The binary predicates assume that each of the tasks A and B can occur in at least one instance of the model.

Behavioral predicate	**Definition**
`CanOccur(A)`	Find all process models where task A occurs in *at least one* instance.
`AlwaysOccurs(A)`	Find all process models where task A occurs in *every* instance.
`Cooccur(A,B)`	Find all process models where it holds that if task A occurs in *some* instance then task B occurs in the *same* instance, and *vice versa*.
`Conflict(A,B)`	Find all process models where it holds that there is *no* instance in which tasks A and B *both* occur.
`ExistCausal(A,B)`	Find all process models where in *at least one* instance it holds that *some* occurrence of task A precedes *some* occurrence of task B.
`ExistTotalCausal(A,B)`	Find all process models where in *at least one* instance it holds that tasks A and B both occur and *every* occurrence of task A precedes *every* occurrence of task B.
`TotalExistCausal(A,B)`	Find all process models where for *every* instance in which tasks A and B *both* occur, it holds that *some* occurrence of task A precedes *some* occurrence of task B.
`TotalCausal(A,B)`	Find all process models where for *every* instance in which tasks A and B *both* occur, it holds that *every* occurrence of task A precedes *every* occurrence of task B.
`ExistConcurrent(A,B)`	Find all process models where in *at least one* instance it holds that *some* occurrence of task A can be executed at the same time with *some* occurrence of task B.
`ExistTotalConcurrent(A,B)`	Find all process models where in *at least one* instance it holds that tasks A and B *both* occur and *every* occurrence of task A can be executed at the same time with *every* occurrence of task B.
`TotalExistConcurrent(A,B)`	Find all process models where for *every* instance in which tasks A and B both occur, it holds that *some* occurrence of task A can be executed at the same time with *some* occurrence of task B.
`TotalConcurrent(A,B)`	Find all process models where for *every* instance in which tasks A and B *both* occur, it holds that *every* occurrence of task A can be executed at the same time with *every* occurrence of task B.

process querying. Finally, the list of selected predicates must be concise. This final decision was driven by the fact that the discussions of the predicates with the study participant should take at most one hour.

4.2. Experiment Design

Armed with the list of predicates from Table 2, we designed an experiment with two aims: (i) to understand the practical relevance of using behavioral predicates for querying process repositories, and (ii) to identify the most relevant predicates to implement in our query language. The experiment took the form of a one-hour semi-structured interview to seek expert opinions from practitioners that actively work with process models. The practitioners were contacted via public posting in dedicated Internet groups in business process management and process mining or approached directly using our industry network.

In the interviews, we explained to participants the rationale of the experiment and introduced the main characteristics of the envisioned query language. We explained the twelve predicates from Table 2 using simple examples and conducted short tests to ensure that interviewees grasped the meaning of the predicates. For example, to test the understanding of the `CanOccur` predicate, we checked if the participants can "find all process models where task A occurs in at least one instance" in an example collection of process models. After the preliminary explanations and tests, we proceeded with the questionnaire in three parts.

In the first part, we collected demographic information on the participants. For example, we asked the participants about the number and type of models managed, the key problems faced with these models, and the extent of process analysis conducted on these models daily.

In the second part, we used four metrics of data quality (usefulness, importance, likelihood, and frequency) as proxies for relevance by asking the following four questions for each predicate, using 5-point Likert-type scales for the answers:

◦ How **useful** would an answer to such a question type be for your process analysis work?
◦ How **important** is such a question type to your process analysis work?
◦ During process analysis, how **likely** does such a question type occur?
◦ During process analysis, how **frequently** does such a question type occur?

Usefulness and *importance* are two external metrics on data quality [45]. They focus on the use and effect of an information system (in our case, of a given predicate in the PQL language), addressing the purpose and justification of the system and its deployment in an organization. In the study, we adopted the definition of usefulness from [46], which states that the perceived usefulness of a system is "the degree to which a person believes that using a particular system would enhance their job performance." The importance of information is defined in [47] as a degree to which information is a necessary input for task accomplishment. Thus, usefulness and importance are measures of performance enhancement and appropriateness and are both task- and user-dependent.

While Wand and Wang [45] present a range of external metrics, such as timeliness, flexibility, sufficiency, and conciseness, we deemed usefulness and importance to be the most representative in our context (assessing the relevance of a behavioral predicate), in light of the need to keep the interview brief. Internal metrics, such as accuracy, have been proven formally in this article.

We complemented usefulness and importance with *likelihood* and *frequency*, two other data quality metrics which act as a proxy for the occurrence of a given predicate during process analysis. Specifically, these latter two metrics measure the *intensity* of using a predicate in an organization, i.e., the more likely and frequently a predicate occurs, the more intense its use [48]. Thus, likelihood and frequency are measures of significance and volume of problem occurrences. Likelihood is commonly understood as the condition of something being likely, or probable, while frequency is the rate at which something occurs over time.

Finally, in the third part of the questionnaire, we requested the interviewees to provide additional comments on the envisioned query language.

During the interviews, we explained the selected data quality metrics and their differences to the participants. The complete interview instrument, including the presentation slides we used to guide the discussions while progressing through the three parts of the interview, is publicly available.[2]

4.3. Experiment Results

We conducted 25 interviews with practitioners. The results of two interviews were discarded, one because of inconsistency in the obtained feedback (the verbal responses contradicted the written responses) and one due to the interviewee's background (CEO of a tool vendor with experience in managing internal processes only), leading to a total of 23 interviews taken to the analysis phase.

All the interviewees work, or have worked in the past, with process models; hence, all of them are potential future users of PQL. Most of our study participants have a degree in Information Technology, Computer Science, Information Systems, Engineering, or Economics; at the undergraduate and/or graduate level. Many study participants hold dual degrees, including degrees in psychology, accounting, biology, political science, business and marketing, management, and business process management (BPM). Four participants received a Ph.D. degree. In their organizations, they have various roles; for example, they are employed as business process analysts, business excellence managers, business architects, process architects, and BPM consultants. The professional experience of the participants ranges from half a year to over 40 years, with most of the interviewees having more than 7 years of professional experience (12.5 years on average). The interviewees reported that the number of process models they managed/analyzed in their practice varies from dozens to thousands (2,780 models on average). These models belong to different domains: insurance, banking, investment, business recovery, HR, finance and budgeting, procurement, product lifecycle, IT and change management, media, and healthcare. The type of models is also varied, ranging from simple and structured to large, complex, and unstructured models captured using EPC and BPMN languages. The most recurring problems in managing process models are maintenance and understandability, validation, compliance management, standardization, and audit. The profiles of the 23 interview participants are listed in Appendix A.

We analyzed the written responses and the transcripts of the discussions in the third part of the interviews to identify categories [49] that relate to the motivation for a language for behavioral querying of process model repositories. As a result, we identified these categories: the relevance of behavioral querying, use cases of behavioral querying, the relevance of behavioral predicates for querying, label similarity, and concrete syntax. These categories informed the design of PQL; refer to Section 5.1. Of the 23 interviewees, 18 explicitly commented

[2]`https://doi.org/10.26188/21670385`

Table 3: Medians and modes of the responses to the four questions on the relevance of the twelve behavioral predicates collected in 23 interviews (Median/Mode).

	`CanOccur`	`AlwaysOccurs`	`Cooccur`	`Conflict`	`ExistCausal`	`ExistTotalCausal`	`TotalExistCausal`	`TotalCausal`	`ExistConcurrent`	`ExistTotalConcurrent`	`TotalExistConcurrent`	`TotalConcurrent`
Useful	**4/5**	**4/4**	**4/5**	**4/4**	3/2	4/4	3/4	**4/5**	3/4	3/4	4/4	**4/4**
Important	**4/4**	**4/4**	**4/5**	**4/4**	3/2	4/4	3/2	**4/5**	3/2	3/2	3/4	**4/4**
Likely	**5/5**	**4/4**	**4/4**	**4/4**	3/4	3/4	4/4	**4/4**	3/4	3/3	3/4	**4/4**
Frequent	**4/3**	**3/3**	**3/3**	**3/3**	3/3	3/3	3/3	**3/3**	3/3	3/3	3/3	**3/3**
Total	**17/15**	**15/15**	**15/17**	**15/15**	12/11	14/15	13/13	**15/17**	12/13	12/12	13/15	**15/15**

on the usefulness and importance of querying process model repositories using behavioral predicates. Some of these comments are quoted in Appendix B.

To identify relevant predicates for inclusion in the query language, we analyzed the central tendency of the responses obtained in the second part of the interviews. As collected responses are ordinal, scores on the Likert scale, for each predicate and question, we analyzed the median and mode of the responses. The results are reported in Table 3. Each cell of the table between rows two and five and between columns two and thirteen reports the median and mode (median/mode) of the 23 responses collected for the question indicated in the first column of the corresponding row and the predicate indicated in the first row of the corresponding column. For example, the median and mode of the 23 collected responses on the usefulness of the `Cooccur` predicate are 'very useful' (4) and 'extremely useful' (5), respectively. The last row in Table 3 reports the sums of medians and modes over the four questions.

We selected the six most relevant, i.e., those with the highest median and mode values, behavioral predicates for inclusion in PQL. These are the `CanOccur`, `AlwaysOccurs`, `Cooccur`, `Conflict`, `TotalCausal`, and `TotalConcurrent` predicates; the corresponding columns are highlighted with bold font in the table. Each selected predicate has obtained a sum of at least 15 for median and mode. Moreover, all the median and mode values for the selected predicates have received scores of at least 3 and cover all four behavioral categories. Interestingly, the `TotalCausal` and `TotalConcurrent` predicates were perceived as more relevant than other versions of the causality and concurrency predicates. Arguably, they are simpler to understand and to relate to practice (e.g., for compliance purposes) since it requires all instances of a process model to satisfy the behavioral relation captured by the predicate. Finally, as `Cooccur` and `Conflict` are defined as macros over `CanCooccur` and `CanConflict` predicates of the 4C spectrum [20], we decided to also include these two latter predicates in the selection of the core PQL predicates. `CanCooccur(A,B)` verifies if the model specifies at least one instance that contains tasks A and B, while `CanConflict(A,B)` checks if the model describes an instance that contains task A but does not contain task B.

Table 4: Expected minimal medians of responses (p-value of 0.05).

	`CanOccur`	`AlwaysOccurs`	`Cooccur`	`Conflict`	`TotalCausal`	`TotalConcurrent`
Useful	4	3	4	3	4	4
Important	4	3	3	3	3	3
Likely	4	3	4	3	4	3
Frequent	3	3	3	4	3	3

We performed *sign tests* to check if the medians of the responses are significantly greater than specific values. The sign test is a nonparametric test for hypotheses about a population median given a sample of observations from that population [50]. These three observations justify the decision to perform sign tests: (i) the scales used to collect the responses, except for the likelihood scale, are not symmetric, (ii) the distances between the answers are not always uniform, and (iii) the collected responses are often not normally distributed.

For each question and behavioral predicate, we performed three one-tailed sign tests to check hypotheses of the form $H_0^x : M \leq x$, where M is the median response to the question w.r.t. the predicate and $x \in \{1.5, 2.5, 3.5\}$. For example, if one succeeds in rejecting hypothesis $H_0^{2.5}$, then the expected median response to the corresponding question w.r.t. the behavioral predicate is 3 or higher. For the six selected predicates, Table 4 reports expected minimal median values of responses to the four questions; obtained by rejecting the corresponding hypothesis based on p-values of at most 0.05. For example, the value of 3 for the usefulness of the `AlwaysOccurs` predicate reported in Table 4 indicates that we, based on the collected responses, were able to reject $H_0^{1.5}$ and $H_0^{2.5}$, but *not* $H_0^{3.5}$, for the corresponding combination of question and predicate. Thus, if one repeats the study, we are at least 95% confident that the median response on the usefulness of the `AlwaysOccurs` predicate will be at least 3. Thus, the expected responses to the four questions w.r.t. the six selected predicates are at least 'moderately useful' (3) or 'very useful' (4) for usefulness, 'moderately important' (3) or 'very important' (4) for importance, 'neutral' (3) or 'likely' (4) for likelihood, and 'occasionally' (3) or 'almost every time' (4) for frequency. Note that the minimal identified medians for the non-selected predicates were generally lower than for the selected predicates.

To check whether the number of conducted interviews was sufficient to obtain generalizable insights, we estimated the statistical power of our study using G*Power 3.1 [51]. Given a sample size of $n = 23$, expecting a medium effect size (0.3) and a low error probability (0.05), i.e., the probability that the observed result is due to chance, our experiment design achieves a statistical power of 0.93, which is well above the suggested threshold of 0.8.

5. Language

This section presents PQL. Section 5.1 summarizes the core principles followed in the design of PQL. Section 5.2 presents the abstract syntax of PQL, the core structure of the language. Section 5.3 is devoted to the discussion of one concrete syntax of the language, its machine- and human-readable specification. Section 5.4 is devoted to the dynamic semantics, or the meaning, of PQL queries. Section 5.5 states denotations of the PQL predicates and proposes techniques for computing them. Finally, Section 5.6 discusses example PQL queries.

5.1. Design Principles

PQL has been designed using the principles of suitability, simplicity, orthogonality, portability, decidability, and exploratory search support. Most of these principles are the standard principles of programming language design [52, 53], while the principle of exploratory search support, borrowed from information retrieval, was motivated by the study reported in Section 4.

Suitability. PQL queries should support the fulfillment of practical tasks. This principle is achieved by grounding the language in the behavioral predicates of practical relevance to process practitioners; refer to Section 4 for details.

Simplicity. PQL queries should allow capturing intents in short, succinct programs. The queries should be easy to read and comprehend. The concrete syntax of PQL is inspired by SQL, a well-known language for querying relational databases. Note that six participants of our study have explicitly suggested that the envisaged process querying language should resemble SQL, and most participants were familiar with SQL. Several quotes from the interviewees that support the use of an SQL-like syntax for PQL are included in Appendix B. Furthermore, to keep queries short, PQL macros provide users with a mechanism to express several atomic statements using a single PQL construct.

Orthogonality. PQL should be based on a small number of behavioral predicates that address orthogonal behavioral phenomena and allow combining them in many different ways to express complex queries. PQL relies on predicates grounded in the behavioral relations of the 4C spectrum [20] that systematizes the four orthogonal behavioral relations of causality, conflict, concurrency, and co-occurrence. Furthermore, PQL allows combining the predicates into propositional logic formulas to express complex query intents and supports set operations that can be used, for instance, to construct complex inputs to PQL macros.

Portability. PQL queries should be independent of implementation and execution environments and data formats. This principle is ensured by providing rigorous definitions of the syntax and semantics of the language. Consequently, one can implement PQL using different technologies that target various execution environments. The semantics of PQL operates over Petri net systems, which allows using PQL over process models captured in a wide range of modeling languages, as models captured using most of the well-established process modeling languages can be translated to Petri net systems [30, 32, 33, 54].

Decidability. Given a PQL query and a process model, it should be possible to decide if the model satisfies the query. For each PQL predicate, we demonstrate that it can indeed be computed.

Exploratory search. An exploratory search is an approach to information exploration which represents the activities carried out by users who are unfamiliar with the domain or unsure about their goals or ways to achieve their goals [55]. Often, these users do querying to study the domain or foster learning.

A user of PQL may be unfamiliar with process models stored in the repository or exact labels used to specify process tasks. Indeed, process models often suffer from the inconsistent usage of labels, even when developed for the same domain [56]. Consequently, a search procedure that relies on the exact comparison of task labels is likely to miss some important matches of similar tasks. To address this issue in PQL, task labels can be expanded. In information retrieval, a *query expansion* is a process of reformulating the query to improve the effectiveness of search results [57]. A task label can be reformulated into a similar label, e.g., using the technique proposed in [58]. A fresh label can then replace the original label in the seed query to obtain a new expanded query that can contribute the otherwise unanticipated relevant matches to the search procedure. For example, the user may be inclined to accept that the label "Print inventory list" used to model a function in Fig. 1 is similar to the label "Produce inventory document" used in a query. Several participants of the study reported in Section 4, suggested that the language for behavioral querying of process models should support the users in performing the exploratory search. Several quotes from the participants in support of this claim are listed in Appendix B.

5.2. Abstract Syntax

The abstract syntax of a PQL query can be represented as an abstract syntax tree. Each node of such a tree denotes a PQL construct. For example, Fig. 6 depicts the abstract syntax tree of sample PQL query *Q3* introduced in Section 2. Next, we discuss PQL constructs in detail, starting with the `Query` construct, the top-level construct of every PQL query.

$$\texttt{Query} \ \triangleq \ \mathit{vars} : \texttt{Variables};\ \mathit{atts} : \texttt{Attributes};\ \mathit{locs} : \texttt{Locations};\ \mathit{pred} : \texttt{Predicate}$$

The `Query` construct is defined as an *aggregate* production composed of four components. Thus, every PQL query is composed of variables, attributes, locations, and a predicate. Intuitively, a PQL query specifies an intent to retrieve models, and their attributes (*atts*), that are identified by the locations (*locs*) and satisfy the predicate (*pred*), where the evaluation of the predicate relies on information stored in the variables (*vars*). One can capture a PQL query using abstract syntactic expressions. For example, the statement $\texttt{q} \triangleq \texttt{Query}(\mathit{vars} : \mathit{vs};\ \mathit{atts} : \mathit{as};\ \mathit{locs} : \mathit{ls};\ \mathit{pred} : p)$ defines a query having vs, as, ls, and p, as variables, attributes, locations, and a predicate, respectively.

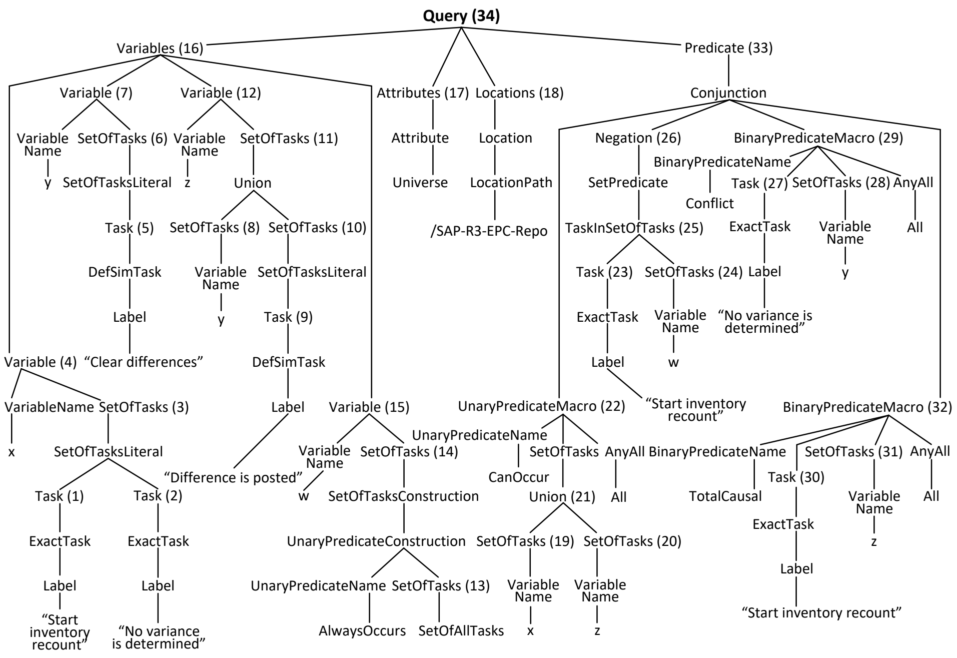


Fig. 6: Abstract syntax tree of example PQL query *Q3* from Section 2.

In PQL, variables, attributes, and locations are defined as *list* productions.

$$\begin{aligned}\texttt{Variables} &\triangleq \texttt{Variable}^* \\ \texttt{Attributes} &\triangleq \texttt{Attribute}^+ \\ \texttt{Locations} &\triangleq \texttt{Location}^+\end{aligned}$$

A PQL query defines a sequence of zero, one, or more variables, denoted by $\texttt{Variable}^*$. The lists of attributes and locations cannot be empty, denoted by $\texttt{Attribute}^+$ and $\texttt{Location}^+$. A PQL variable is defined as follows.

$$\begin{aligned}\texttt{Variable} &\triangleq \mathit{name} : \texttt{VariableName};\ \mathit{tasks} : \texttt{SetOfTasks} \\ \texttt{Attribute} &\triangleq \texttt{Universe} \mid \texttt{AttributeName} \\ \texttt{Location} &\triangleq \texttt{Universe} \mid \texttt{LocationPath}\end{aligned}$$

A PQL variable associates a symbolic name with a set of *PQL tasks*. Tasks are introduced in the language to refer to atomic units of observable behavior. They are the smallest irreducible concepts observed during the execution of process models. Each variable is an aggregate of two constructs: a variable name ($\mathit{name} : \texttt{VariableName}$) and a collection of tasks ($\mathit{tasks} : \texttt{SetOfTasks}$). When a predicate of a query is evaluated, every variable name mentioned in the predicate is replaced by the corresponding set of tasks.

The `Attribute` construct is specified as a *choice* production with two alternatives. Hence, a PQL attribute is either the *universe* attribute, denoted by `Universe`, or the *name* attribute, denoted by `AttributeName`. The universe attribute refers to all the attributes of process models in the repository. The name attribute allows retrieving specific properties of process models, e.g., the unique identifier, creation date, author, and version.

A location is used to identify process models that should be matched with the query. A location is either the *universe* location, denoted by `Universe`, or a *path* location, denoted by `LocationPath`. The universe location refers to all process models in the scope of the query (usually, all process models in the repository). Path locations allow fine-grained targeting of models. We assume that repositories support mechanisms to address models via unique path identifiers, e.g., using URIs [59] or XPath expressions [60].

PQL provides several alternatives for specifying sets of tasks.

$$\begin{aligned}
\texttt{SetOfTasks} \;\triangleq\;& \texttt{VariableName} \mid \texttt{SetOfAllTasks} \mid \texttt{SetOfTasksLiteral} \\
& \mid \texttt{SetOfTasksConstruction} \mid \texttt{Union} \mid \texttt{Intersection} \mid \texttt{Difference}
\end{aligned}$$

First, one can use the `VariableName` construct to refer to the set of tasks associated with the variable of that name. Second, one can use the `SetOfAllTasks` construct, a dynamically-valued constant that refers to the set of all tasks of the model currently being matched to the query. Third, one can specify a set of tasks literal using the `SetOfTasksLiteral` construct.

$$\begin{aligned}
\texttt{SetOfTasksLiteral} &\triangleq \texttt{Task}^{*} \\
\texttt{Task} &\triangleq \texttt{ExactTask} \mid \texttt{DefSimTask} \mid \texttt{SimTask} \\
\texttt{ExactTask} &\triangleq \mathit{label} : \texttt{Label} \\
\texttt{DefSimTask} &\triangleq \mathit{label} : \texttt{Label} \\
\texttt{SimTask} &\triangleq \mathit{label} : \texttt{Label};\ \mathit{sim} : \texttt{Similarity}
\end{aligned}$$

A PQL task is a collection of labels, i.e., character strings, similar to the given label up to the specified similarity degree threshold. The explanation of the differences between the `ExactTask`, `DefSimTask`, and `SimTask` constructs used to define a PQL task is postponed to Section 5.4.

One can build a set of tasks using the `SetOfTasksConstruction` construct.

$$\begin{aligned}
\texttt{SetOfTasksConstruction} &\triangleq \texttt{UnaryPredicateConstruction} \\
& \quad\mid \texttt{BinaryPredicateConstruction} \\
\texttt{UnaryPredicateConstruction} &\triangleq \mathit{name} : \texttt{UnaryPredicateName}; \\
& \quad \mathit{tasks} : \texttt{SetOfTasks} \\
\texttt{BinaryPredicateConstruction} &\triangleq \mathit{name} : \texttt{BinaryPredicateName}; \\
& \quad \mathit{tasks}_1 : \texttt{SetOfTasks}; \\
& \quad \mathit{tasks}_2 : \texttt{SetOfTasks};\ q : \texttt{AnyAll} \\
\texttt{AnyAll} &\triangleq \texttt{Any} \mid \texttt{All}
\end{aligned}$$

`UnaryPredicateConstruction` takes a set of tasks and a unary behavioral predicate as input and constructs the set of tasks that contains every task from the given set for which the predicate holds. `BinaryPredicateConstruction` allows selecting those tasks from the given set for which the given binary behavioral predicate holds, either with at least one or with all tasks taken from another given set of tasks. The choice of a quantifier type, existential or universal, to be used during the selections is implemented via the `AnyAll` construct.

PQL supports two unary and six binary behavioral predicates, see below.

$$\begin{aligned}
\texttt{UnaryPredicateName} &\triangleq \texttt{CanOccur} \mid \texttt{AlwaysOccurs}\\
\texttt{BinaryPredicateName} &\triangleq \texttt{CanConflict} \mid \texttt{CanCooccur} \mid \texttt{Conflict}\\
&\quad \mid \texttt{Cooccur} \mid \texttt{TotalCausal} \mid \texttt{TotalConcurrent}
\end{aligned}$$

Finally, a set of tasks can be constructed from other sets via the application of the fundamental set operations of *union*, *intersection*, and *difference*, denoted by the `Union`, `Intersection`, and `Difference` constructs, respectively.

Alternatives for specifying a PQL predicate are listed below.

$$\begin{aligned}
\texttt{Predicate} &\triangleq \texttt{UnaryPredicate} \mid \texttt{BinaryPredicate} \mid \texttt{UnaryPredicateMacro}\\
&\quad \mid \texttt{BinaryPredicateMacro} \mid \texttt{SetPredicate} \mid \texttt{TruthValue}\\
&\quad \mid \texttt{Negation} \mid \texttt{Conjunction} \mid \texttt{Disjunction} \mid \texttt{LogicalTest}
\end{aligned}$$

For instance, a predicate can be captured as a specimen of the `UnaryPredicate` or `BinaryPredicate` construct.

$$\begin{aligned}
\texttt{UnaryPredicate} &\triangleq \mathit{name} : \texttt{UnaryPredicateName};\ \mathit{task} : \texttt{Task}\\
\texttt{BinaryPredicate} &\triangleq \mathit{name} : \texttt{BinaryPredicateName};\ \mathit{task}_1 : \texttt{Task};\ \mathit{task}_2 : \texttt{Task}
\end{aligned}$$

PQL uses macros to combine results of several predicates into a single result.

$$\begin{aligned}
\texttt{UnaryPredicateMacro} &\triangleq \mathit{name} : \texttt{UnaryPredicateName};\ \mathit{tasks} : \texttt{SetOfTasks};\\
&\quad\ q : \texttt{AnyAll}\\
\texttt{BinaryPredicateMacro} &\triangleq \texttt{BinaryPredicateMacroTaskSet}\\
&\quad \mid \texttt{BinaryPredicateMacroSetSet}
\end{aligned}$$

The `UnaryPredicateMacro` construct is composed of a reference to a unary behavioral predicate (name : `UnaryPredicateName`), a set of tasks (tasks : `SetOfTasks`), and a quantifier (q : `AnyAll`). Intuitively, a single macro statement $p \triangleq \texttt{UnaryPredicateMacro}(\mathit{name} : n;\ \mathit{tasks} : ts;\ q : x)$ is equivalent to a check of whether it holds that for at least one (if x is set to `Any`) or for every (if x is set to `All`) task t in set of tasks ts statement $\texttt{UnaryPredicate}(p.\mathit{name};\ \mathit{task} : t)$ evaluates to *true*. Similarly, one can rely on the `BinaryPredicateMacro` construct to combine results of multiple `BinaryPredicate` checks.

$$\begin{aligned}
\texttt{BinaryPredicateMacroTaskSet} &\triangleq \mathit{name} : \texttt{BinaryPredicateName};\ \mathit{task} : \texttt{Task};\\
&\quad\ \mathit{tasks} : \texttt{SetOfTasks};\ q : \texttt{AnyAll}\\
\texttt{BinaryPredicateMacroSetSet} &\triangleq \mathit{name} : \texttt{BinaryPredicateName};\\
&\quad\ \mathit{tasks}_1 : \texttt{SetOfTasks};\ \mathit{tasks}_2 : \texttt{SetOfTasks};\\
&\quad\ q : \texttt{AnySomeEachAll};\\
\texttt{AnySomeEachAll} &\triangleq \texttt{Any} \mid \texttt{Some} \mid \texttt{Each} \mid \texttt{All}
\end{aligned}$$

The `BinaryPredicateMacroTaskSet` construct checks whether the binary behavioral predicate (*name* : `BinaryPredicateName`) holds between the given task (*task* : `Task`) and either at least one (if the `AnyAll` construct is instantiated with the `Any` specimen) or every (`All`) task in the given set of tasks (*tasks* : `SetOfTasks`). Similarly, `BinaryPredicateMacroSetSet` checks whether a binary behavioral predicate evaluates to *true* for certain pairs of tasks in the Cartesian product of two given sets of tasks. When the `Some` option is used to specify the `AnySomeEachAll` construct, PQL checks whether for *some* task in the first set of tasks the specified behavioral relation holds with each task in the second set of tasks. The `Each` option induces a check of whether for *each* task in the first set the behavioral relation holds with some task from the second set.

PQL supports checks of basic binary relations between (elements of) sets of tasks. These are captured in the production of the `SetPredicate` construct.

$$
\begin{aligned}
\texttt{SetPredicate} &\triangleq \texttt{TaskInSetOfTasks} \mid \texttt{SetComparison}\\
\texttt{TaskInSetOfTasks} &\triangleq \mathit{task} : \texttt{Task};\ \mathit{tasks} : \texttt{SetOfTasks}\\
\texttt{SetComparison} &\triangleq \mathit{tasks}_1 : \texttt{SetOfTasks};\ \mathit{oper} : \texttt{SetComparisonOperator};\\
&\quad\ \mathit{tasks}_2 : \texttt{SetOfTasks}\\
\texttt{SetComparisonOperator} &\triangleq \texttt{Identical} \mid \texttt{Different} \mid \texttt{OverlapsWith} \mid \texttt{SubsetOf}\\
&\quad\ \mid \texttt{ProperSubsetOf}
\end{aligned}
$$

PQL can be used to check if a task is a member of a given set of tasks using the `TaskInSetOfTasks` construct, and to compare sets of tasks using the `SetComparison` construct. PQL supports five set comparison operations that refer to checks of whether two sets of tasks are identical (`Identical`), different (`Different`), overlap (`OverlapsWith`), or whether one set of tasks is a subset (`SubsetOf`) or a proper subset (`ProperSubsetOf`) of another set of tasks.

PQL supports two truth values: *true* and *false*. To allow using complex logical statements, PQL supports standard logical operations of negation (`Negation`), conjunction (`Conjunction`), and disjunction (`Disjunction`). Finally, PQL allows testing whether a given logic value is *true* or *false*. These checks are reflected in the options of the `LogicalTest` construct proposed below.

$$
\begin{aligned}
\texttt{TruthValue} &\triangleq \texttt{True} \mid \texttt{False}\\
\texttt{LogicalTest} &\triangleq \texttt{IsTrue} \mid \texttt{IsNotTrue} \mid \texttt{IsFalse} \mid \texttt{IsNotFalse}
\end{aligned}
$$

For a grammar to be complete, all its constructs must be *terminal*. The following PQL constructs are terminal: `Any`, `Some`, `Universe`, `Different`, `SubsetOf`, `True`, `False`, `CanOccur`, and `Conflict`. All these constructs do not have an internal structure. Several PQL constructs are defined in terms of special sets. For example, PQL specifies `VariableName`, `AttributeName`, `LocationPath`, `Label`, and `Similarity`, as $\texttt{VariableName} \triangleq \mathit{id} : \mathbb{V}$, $\texttt{AttributeName} \triangleq \mathit{id} : \mathbb{C}$, $\texttt{LocationPath} \triangleq \mathit{id} : \mathbb{C}$, $\texttt{Label} \triangleq \mathit{value} : \mathbb{C}$, and $\texttt{Similarity} \triangleq \mathit{value} : [0\,..\,1]$, respectively, where $\mathbb{V}$ is the set of all legal PQL variable names.

PQL defines the `Negation` construct and four options associated with the `LogicalTest` construct in terms of a single `Predicate` component, e.g., $\texttt{IsTrue} \triangleq$

pred : `Predicate` and `Negation` ≜ *pred* : `Predicate`. Appendix C presents an implementation of the PLQ grammar that defines five remaining non-terminal constructs: `Conjunction`, `Disjunction`, `Union`, `Intersection`, and `Difference`.

5.3. Concrete Syntax

This section presents one concrete syntax of PQL; its machine- and human-readable representation. The concrete syntax is inspired by SQL—a programming language for managing data stored in a relational database management system (DBMS) [61]. We keep the core structure of PQL queries as similar as possible to that of SQL queries and reuse SQL keywords in PQL, given that the contexts are similar. The reason for this is threefold:

- Despite addressing different domains, dynamic processes versus static data, the languages serve the same purpose—querying for information.
- SQL is a widely used language supported by almost every DBMS. Its syntax is well-recognized by technical specialists and analysts. By closely following the concrete syntax of SQL, PQL becomes readily usable by many stakeholders.
- As suggested by interviewees of the study reported in Section 4, it would be beneficial for the syntax of the query language to resemble the syntax of SQL.

Given a PQL construct, its concrete syntax is defined by a function that takes a specimen of the abstract construct as input and returns a collection of character strings that are accepted as concrete encodings of the specimen. We denote this function by the name of the construct with subscript `c`. For example, the concrete syntax of a specimen of the `Query` construct is defined below.

$$\begin{aligned}\texttt{Query}_{\texttt{c}}(q : \texttt{Query}) \;\triangleq\;& \texttt{Variables}_{\texttt{c}}(q.\mathit{vars})\\ & \text{`}\texttt{SELECT}\text{'}\ \texttt{Attributes}_{\texttt{c}}(q.\mathit{atts})\\ & \text{`}\texttt{FROM}\text{'}\ \texttt{Locations}_{\texttt{c}}(q.\mathit{locs})\\ & (\text{`}\texttt{WHERE}\text{'}\ \texttt{Predicate}_{\texttt{c}}(q.\mathit{pred}))?\ \text{`}\texttt{;}\text{'}\end{aligned}$$

Thus, a PQL query is a character string that starts with a specification of variables, followed by the `SELECT` keyword, followed by a specification of attributes, followed by the `FROM` keyword, followed by a specification of locations, followed by the `WHERE` keyword, followed by a specification of a predicate, followed by the semicolon mark. There can be an arbitrary number of whitespace characters between any two subsequent components of a query string. The order of components is fixed, and the `WHERE` clause is optional in a query.

Specimens of PQL constructs associated with list productions are encoded as string concatenations of concrete forms of their components and whitespace characters. Sometimes, special symbols are injected between every two subsequent components and/or at the beginning and end of the encodings. For example, the concrete syntax of a list of variables is defined as follows.

$$\begin{aligned}\texttt{Variables}_{\texttt{c}}(\mathit{vs} : \texttt{Variables}) \;\triangleq\;& \mathit{isEmpty}(\mathit{vs})\ ?\ \text{`'} : \texttt{Variable}_{\texttt{c}}(\mathit{vs}.\mathit{FIRST})\\ & \texttt{Variables}_{\texttt{c}}(\mathit{vs}.\mathit{TAIL})\end{aligned}$$

The empty list of variables is encoded as the empty string. Otherwise, its encoding is constructed as a concatenation of a concrete form of its first element, denoted by *vs.FIRST*, and an encoding of the list of all its other elements, denoted by *vs.TAIL*. The concrete syntax of a PQL variable is defined below.

$$\texttt{Variable}_{\texttt{c}}(v : \texttt{Variable}) \triangleq \texttt{VariableName}_{\texttt{c}}(v.\mathit{name})\ \text{`}\texttt{=}\text{'}\ \texttt{SetOfTasks}_{\texttt{c}}(v.\mathit{tasks})\ \text{`}\texttt{;}\text{'}$$

The concrete syntax of every other specimen of a PQL construct associated with a list production includes a special symbol between every two subsequent elements. This is the comma symbol, i.e., ‘`,`’, for specimens of `Attributes`, `Locations`, and `SetOfTasksLiteral`, and the PQL keywords `UNION`, `INTERSECT`, `EXCEPT`, `AND`, and `OR`, for specimens of `Union`, `Intersection`, `Difference`, `Conjunction`, and `Disjunction`, respectively. In addition, every encoding of a specimen of the `SetOfTasksLiteral` construct must begin with the opening curly bracket, i.e., ‘`{`’, and end with the closing curly bracket, i.e., ‘`}`’. For example, the character string ‘`{"Buy item","Purchase product"}`’ encodes a specimen of `SetOfTasksLiteral` composed of two elements, where strings `"Buy item"` and `"Purchase product"` are valid encodings of tasks. Note that PQL supports three alternative concrete encodings of a PQL task.

$$\begin{aligned}
\texttt{ExactTask}_{\texttt{c}}(t : \texttt{ExactTask}) &\triangleq \text{`}\texttt{"}\text{'}\ \texttt{Label}_{\texttt{c}}(t.\mathit{label})\ \text{`}\texttt{"}\text{'}\\
\texttt{DefSimTask}_{\texttt{c}}(t : \texttt{DefSimTask}) &\triangleq \text{`}\sim\text{'}\ \text{`}\texttt{"}\text{'}\ \texttt{Label}_{\texttt{c}}(t.\mathit{label})\ \text{`}\texttt{"}\text{'}\\
\texttt{SimTask}_{\texttt{c}}(t : \texttt{SimTask}) &\triangleq \text{`}\texttt{"}\text{'}\ \texttt{Label}_{\texttt{c}}(t.\mathit{label})\ \text{`}\texttt{"}\text{'}\ \text{`}\texttt{[}\text{'}\ \texttt{Similarity}_{\texttt{c}}(t.\mathit{sim})\ \text{`}\texttt{]}\text{'}
\end{aligned}$$

Labels of PQL tasks must be enclosed in double quotes. A label can be preceded by the tilde symbol, i.e., ‘~’, or succeeded by an encoding of a similarity degree threshold enclosed in square brackets. The tilde symbol denotes that one is interested in all the tasks of which the label has a degree of similarity to the specified label that is equal to or is larger than some preconfigured value. A degree of similarity must be specified as a decimal representation of a real number greater or equal to zero and less than or equal to one, e.g., 0.5 or .95.

A specimen associated with a choice production is a specimen of one of the constructs from its list of alternatives. Thus, in what follows, we present concrete encodings of the remaining aggregate productions.

The `Universe` construct is denoted by the asterisk symbol, i.e., ‘`*`’. The specimens of the `AttributeName` and `LocationPath` constructs are denoted by character strings enclosed in double quotes. A concrete encoding of a specimen of the `VariableName` construct may contain lowercase letters from the English alphabet, digits, and the underscore symbol, i.e., ‘`_`’. It is necessary to use a letter or the underscore symbol at the start of a variable name.

Next, we define possible concrete encodings of `UnaryPredicateConstruction`, `BinaryPredicateConstruction`, and `SetOfAllTasks`.

$$\begin{aligned}
&\texttt{UnaryPredicateConstruction}_{\texttt{c}}(\mathit{upc} : \texttt{UnaryPredicateConstruction}) \triangleq\\
&\quad \text{`}\texttt{GetTasks}\text{'}\texttt{UnaryPredicateName}_{\texttt{c}}(\mathit{upc.name})\ \text{`}\texttt{(}\text{'}\ \texttt{SetOfTasks}_{\texttt{c}}(\mathit{upc.tasks})\ \text{`}\texttt{)}\text{'}
\end{aligned}$$

$\texttt{BinaryPredicateConstruction}_{\text{c}}(bpc : \texttt{BinaryPredicateConstruction}) \triangleq$
$\quad$ ‘`GetTasks`’$\texttt{BinaryPredicateName}_{\text{c}}(bpc.name)$
$\quad$ ‘`(`’ $\texttt{SetOfTasks}_{\text{c}}(bpc.tasks_1)$ ‘`,`’ $\texttt{SetOfTasks}_{\text{c}}(bpc.tasks_2)$ ‘`,`’ $\texttt{AnyAll}_{\text{c}}(bpc.q)$ ‘`)`’

$\texttt{SetOfAllTasks}_{\text{c}}(ts : \texttt{SetOfAllTasks}) \triangleq$ ‘`GetTasks`’ ‘`(`’ ‘`)`’

The concrete encodings of specimens of these constructs follow the syntax for specifying function calls used in many programming languages, i.e., a name of a function to be called followed by a comma-separated list of parameters which is enclosed in parentheses. For instance, one possible concrete encoding of a `SetOfAllTasks` specimen is ‘`GetTasks()`’. In the case of specimens of the `UnaryPredicateConstruction` and `BinaryPredicateConstruction` constructs, the names of functions are obtained by prefixing ‘`GetTasks`’ to names of unary and binary predicates, respectively. The remaining components are used as parameters of the corresponding functions. PQL exercises similar principles when specifying the concrete syntax of predicates and macros, both for the unary and binary cases. The concrete syntax for predicates proceeds as follows.

$\texttt{UnaryPredicate}_{\text{c}}(up : \texttt{UnaryPredicate}) \triangleq$
$\quad$ $\texttt{UnaryPredicateName}_{\text{c}}(up.name)$ ‘`(`’ $\texttt{Task}_{\text{c}}(up.task)$ ‘`)`’

$\texttt{BinaryPredicate}_{\text{c}}(bp : \texttt{BinaryPredicate}) \triangleq$
$\quad$ $\texttt{BinaryPredicateName}_{\text{c}}(bp.name)$ ‘`(`’ $\texttt{Task}_{\text{c}}(bp.task_1)$ ‘`,`’ $\texttt{Task}_{\text{c}}(bp.task_2)$ ‘`)`’

The concrete syntax for denoting the PQL macros overloads the syntax for specifying function calls that encode the PQL predicates, i.e., names of functions and types of outputs are the same, both for a given predicate and the corresponding macro. However, the types of inputs differ.

$\texttt{UnaryPredicateMacro}_{\text{c}}(upm : \texttt{UnaryPredicateMacro}) \triangleq$
$\quad$ $\texttt{UnaryPredicateName}_{\text{c}}(upm.name)$ ‘`(`’ $\texttt{SetOfTask}_{\text{c}}(upm.tasks)$ ‘`,`’
$\quad$ $\texttt{AnyAll}_{\text{c}}(upm.q)$ ‘`)`’

$\texttt{BinaryPredicateMacroTaskSet}_{\text{c}}(bpm : \texttt{BinaryPredicateMacroTaskSet}) \triangleq$
$\quad$ $\texttt{BinaryPredicateName}_{\text{c}}(bpm.name)$
$\quad$ ‘`(`’ $\texttt{Task}_{\text{c}}(bpm.task)$ ‘`,`’ $\texttt{SetOfTask}_{\text{c}}(bpm.tasks)$ ‘`,`’ $\texttt{AnyAll}_{\text{c}}(bpm.q)$ ‘`)`’

$\texttt{BinaryPredicateMacroSetSet}_{\text{c}}(bpm : \texttt{BinaryPredicateMacroSetSet}) \triangleq$
$\quad$ $\texttt{BinaryPredicateName}_{\text{c}}(bpm.name)$
$\quad$ ‘`(`’ $\texttt{SetOfTask}_{\text{c}}(bpm.tasks_1)$ ‘`,`’ $\texttt{SetOfTask}_{\text{c}}(bpm.tasks_2)$ ‘`,`’
$\quad$ $\texttt{AnySomeEachAll}_{\text{c}}(bpm.q)$ ‘`)`’

These syntax rules rely on the concrete encodings of `AnyAll` and `AnySomeEachAll`, which, when instantiated, are specified as a specimen of the `Any`, `Some`, `Each`, or `All` construct and are denoted by the PQL keywords `ANY`, `SOME`, `EACH`, and `ALL`, respectively. A specimen of `TaskInSetOfTasks` can be specified as follows.

$$\begin{aligned}\texttt{TaskInSetOfTasks}_{\mathsf{c}}(\mathit{in} : \texttt{TaskInSetOfTasks}) \;&\triangleq\; \texttt{Task}_{\mathsf{c}}(\mathit{in.task})\ \texttt{`IN'}\\ &\;\; \texttt{SetOfTask}_{\mathsf{c}}(\mathit{in.tasks})\end{aligned}$$

A specimen of `SetComparison` can be specified as two encodings of sets of tasks with a representation of a comparison operator in between.

$$\begin{aligned}&\texttt{SetComparison}_{\mathsf{c}}(\mathit{comp} : \texttt{SetComparison}) \;\triangleq\\ &\qquad \texttt{SetOfTask}_{\mathsf{c}}(\mathit{comp.tasks}_1)\ \texttt{SetComparisonOperator}_{\mathsf{c}}(\mathit{comp.oper})\\ &\qquad \texttt{SetOfTask}_{\mathsf{c}}(\mathit{comp.tasks}_2)\end{aligned}$$

A set comparison operator is instantiated in PQL via a choice between specimens of terminal constructs `Identical`, `Different`, `OverlapsWith`, `SubsetOf`, `ProperSubsetOf` encoded using keywords `EQUALS`, `NOT EQUALS`, `OVERLAPS WITH`, `IS SUBSET OF`, `IS PROPER SUBSET OF`, respectively. The terminal constructs `True` and `False` get encoded as keywords `TRUE` and `FALSE`, respectively.

Predicate names are encoded as names of the corresponding terminal constructs, e.g., `CanOccur` and `TotalCausal` have concrete encodings `'CanOccur'` and `'TotalCausal'`, respectively. Finally, the concrete encodings of `Negation` and the four logical test constructs are proposed below.

$$\begin{aligned}\texttt{Negation}_{\mathsf{c}}(\mathit{not} : \texttt{Negation}) \;&\triangleq\; \texttt{`NOT'}\ \texttt{Predicate}_{\mathsf{c}}(\mathit{not.pred})\\ \texttt{IsTrue}_{\mathsf{c}}(\mathit{test} : \texttt{IsTrue}) \;&\triangleq\; \texttt{Predicate}_{\mathsf{c}}(\mathit{test.pred})\ \texttt{`IS'}\ \texttt{`TRUE'}\\ \texttt{IsNotTrue}_{\mathsf{c}}(\mathit{test} : \texttt{IsNotTrue}) \;&\triangleq\; \texttt{Predicate}_{\mathsf{c}}(\mathit{test.pred})\ \texttt{`IS'}\ \texttt{`NOT'}\ \texttt{`TRUE'}\\ \texttt{IsFalse}_{\mathsf{c}}(\mathit{test} : \texttt{IsFalse}) \;&\triangleq\; \texttt{Predicate}_{\mathsf{c}}(\mathit{test.pred})\ \texttt{`IS'}\ \texttt{`FALSE'}\\ \texttt{IsNotFalse}_{\mathsf{c}}(\mathit{test} : \texttt{IsNotFalse}) \;&\triangleq\; \texttt{Predicate}_{\mathsf{c}}(\mathit{test.pred})\ \texttt{`IS'}\ \texttt{`NOT'}\ \texttt{`FALSE'}\end{aligned}$$

5.4. Dynamic Semantics

The dynamic semantics of PQL is captured in *meaning functions* that describe the effects of valid constructs using mathematical denotations over the following domains: $\mathbb{A}$, a universe of *attribute names*; $\mathbb{B}$, a universe of *attribute values*; $\mathbb{L}$, a universe of *locations*; $\mathbb{S}$, a universe of *Petri net systems*; $\mathbb{T} := \wp_{\geq 1}(\mathbb{C})$, the universe of all *tasks* over the universe of character strings.[3]

Let $\chi : \mathbb{A} \to \wp_{\geq 1}(\mathbb{B})$ be a function that maps attribute names onto sets of permissible attribute values. A PQL query formulates a request to retrieve process models (and their attributes) from a given process model repository.

Definition 5.1 (Repositories).
A *process model repository*, or a *repository*, is a 6-tuple $R := (S, A, L, \mathit{val}, \mathit{loc}, \precsim)$, where $S \subseteq \mathbb{S}$ is a finite set of *systems*, $A \subseteq \mathbb{A}$ is a finite set of *attribute names*, $L \subseteq \mathbb{L}$ is a set of *locations*, $\mathit{val} : S \times A \to \mathbb{B}$ is the *attribute value assignment* function, s.t. $\forall s \in S\ \forall a \in A : \mathit{val}(s, a) \in \chi(a)$, $\mathit{loc} : S \to L$ is the *location assignment* function, and $\precsim$ is a reflexive binary relation over L, called *location map*. ⌟

[3]Given a set A, by $\wp_{\geq 1}(A)$ we denote the set of all non-empty subsets of A, i.e., the power set of A without the empty set.

By `Repository`, we denote the universe of all possible repositories. Let $k := (k_1, k_2, \ldots, k_n) \in K_1 \times K_2 \times \ldots \times K_n$ be a point in n-dimensional space, where $K_1, K_2, \ldots, K_n$ are some sets. The *projection function* $\pi_i(k)$, $i \in [1\,..\,n]$, is defined as $\pi_i(k) := k_i$, where k_i is the i-th coordinate of k.

The meaning function of the `Query` construct is defined as follows.

$$\begin{array}{rcl}
M_{\texttt{Query}} & : & \texttt{Query} \times \texttt{Repository} \rightarrow \mathbb{S} \times \wp(\mathbb{A} \times \mathbb{B}) \\
M_{\texttt{Query}}[q : \texttt{Query}, & \triangleq & \{(s, x) \in \mathbb{S} \times \wp(\mathbb{A} \times \mathbb{B}) \mid \\
(S, A, L, val, loc, \lesssim) : \texttt{Repository}] & & (\exists\, l \in M_{\texttt{Locations}}(q.locs) : loc(s) \lesssim l) \;\wedge \\
 & & (\pi_1(x) = M_{\texttt{Attributes}}(q.atts)) \;\wedge \\
 & & (\forall (a, v) \in x : v = val(s, a)) \;\wedge \\
 & & (M_{\texttt{Predicate}}(q.pred, s, M_{\texttt{Variables}}(q.vars, s)))\}
\end{array}$$

Similar as in [62], we denote the meaning function of a construct `c` by $M_{\texttt{c}}$.

The result of executing a query q in the context of a repository consists of all the systems at locations nested in $q.locs$ that satisfy predicate $q.pred$, and attribute values of these systems requested in $q.atts$.

The meaning of a specimen of the `Locations` construct is defined below.

$$\begin{array}{rcl}
M_{\texttt{Locations}} & : & \texttt{Locations} \rightarrow \wp_{\geq 1}(\mathbb{L}) \\
M_{\texttt{Locations}}[ls : \texttt{Locations}] & \triangleq & \bigcup\limits_{i \in [1..|ls|]} M_{\texttt{Location}}(ls_i) \\
M_{\texttt{Location}} & : & \texttt{Location} \rightarrow \wp_{\geq 1}(\mathbb{L}) \\
M_{\texttt{Location}}[l : \texttt{Location}] & \triangleq & \begin{cases} \mathbb{L} & l \texttt{ is Universe} \\ \{M_{\texttt{LocationPath}}(l)\} & \textit{otherwise} \end{cases} \\
M_{\texttt{LocationPath}} & : & \texttt{LocationPath} \rightarrow \mathbb{L}
\end{array}$$

Note the use of the `is` operator above. This operator specifies a Boolean expression that checks if a specified specimen is a specimen of the given construct.

The meaning of a specimen of the `Attributes` construct is defined similarly to that of the `Locations` construct and is proposed below.

$$\begin{array}{rcl}
M_{\texttt{Attributes}} & : & \texttt{Attributes} \rightarrow \wp_{\geq 1}(\mathbb{A}) \\
M_{\texttt{Attributes}}[as : \texttt{Attributes}] & \triangleq & \bigcup\limits_{i \in [1..|as|]} M_{\texttt{Attribute}}(as_i) \\
M_{\texttt{Attribute}} & : & \texttt{Attribute} \rightarrow \wp_{\geq 1}(\mathbb{A}) \\
M_{\texttt{Attribute}}[a : \texttt{Attribute}] & \triangleq & \begin{cases} \mathbb{A} & a \texttt{ is Universe} \\ \{M_{\texttt{AttributeName}}(a)\} & \textit{otherwise} \end{cases} \\
M_{\texttt{AttributeName}} & : & \texttt{AttributeName} \rightarrow \mathbb{A}
\end{array}$$

The *overriding union* of $f : X_1 \rightarrow Y_1$ by $g : X_2 \rightarrow Y_2$, denoted by $f \oplus g$, is defined by $g \cup \{(x, f(x)) \mid x \in dom(f) \setminus dom(g)\}$. Given a sequence of functions fs, $|fs| = n$, $\bigoplus_{i=1}^{n} fs_i$ denotes the expression $((\ldots((fs_1 \oplus fs_2) \oplus fs_3) \oplus \ldots fs_{n-1}) \oplus fs_n)$.

Next, we propose the denotation of a specimen of the `Variables` construct.

$$\begin{aligned}
M_{\texttt{Variables}} &: \texttt{Variables} \times \mathbb{S} \to \wp(\mathbb{V} \times \wp(\mathbb{T})) \\
M_{\texttt{Variables}}[vs : \texttt{Variables}, s : \mathbb{S}] &\triangleq \bigoplus_{i=1}^{|vs|} \{ M_{\texttt{Variable}}(vs_i, s, M_{\texttt{Variables}}(\mathit{prefix}(vs, i), s)) \} \\
M_{\texttt{Variable}} &: \texttt{Variable} \times \mathbb{S} \times \wp(\mathbb{V} \times \wp(\mathbb{T})) \to \mathbb{V} \times \wp(\mathbb{T})
\end{aligned}$$

$$M_{\texttt{Variable}}[v : \texttt{Variable}, s : \mathbb{S}, \mathit{vals} : \wp(\mathbb{V} \times \wp(\mathbb{T}))] \triangleq (v.\mathit{name}.\mathit{id}, M_{\texttt{SetOfTasks}}(v.\mathit{tasks}, s, \mathit{vals}))$$

The denotation of a specimen vs of `Variables` is a set of variable-value pairs, where a value is a set of tasks, computed in the context of process model $s \in \mathbb{S}$ as the overriding union of meanings of the individual elements of vs. An element of vs at position $i \in [1..|vs|]$ is a specimen $v = vs_i$ of `Variable`. The meaning of v is derived in the context of s and variable-value pairs vals computed based on the prefix of vs up to but excluding position i, i.e., values of variables declared prior to v, and is defined as a pair composed of a legal PQL variable name associated with $v.\mathit{name}$ and a set of tasks that stems from the denotation of $v.\mathit{tasks}$.

Let $\mathit{labels} : \mathbb{S} \to \wp(\mathbb{C})$, be a function that, given a system, results in the set of all the labels of its observable transitions. Let $\mathit{similar} : \mathbb{C} \times [0\,..\,1] \to \wp_{\geq 1}(\mathbb{C})$ be a function that, given a label and a similarity level threshold, results in the set of all labels that have similarity scores with the given label equal or greater than the threshold. Then, the denotation of the `SetOfTasks` construct is as follows.

$$\begin{aligned}
M_{\texttt{SetOfTasks}} &: \texttt{SetOfTasks} \times \mathbb{S} \times \wp(\mathbb{V} \times \wp(\mathbb{T})) \to \wp(\mathbb{T}) \\
\begin{array}{r} M_{\texttt{SetOfTasks}}[\\ ts : \texttt{SetOfTasks}, \\ s : \mathbb{S},\ \mathit{vals} : \wp(\mathbb{V} \times \wp(\mathbb{T}))] \end{array} &\triangleq \begin{cases}
\mathit{vals}(ts.\mathit{id}) & ts \texttt{ is VariableName} \\
\{\{\lambda\} \in \mathbb{T} \mid \lambda \in \mathit{labels}(s)\} & ts \texttt{ is SetOfAllTasks} \\
\bigcup_{i \in [1..|ts|]} \{M_{\texttt{Task}}(ts_i)\} & ts \texttt{ is SetOfTasksLiteral} \\
M_{\texttt{SetOfTasksConstruction}}(ts, s, \mathit{vals}) & ts \texttt{ is SetOfTasksConstruction} \\
\bigcup_{i \in [1..|ts|]} M_{\texttt{SetOfTasks}}(ts_i, s, \mathit{vals}) & ts \texttt{ is Union} \\
\bigcap_{i \in [1..|ts|]} M_{\texttt{SetOfTasks}}(ts_i, s, \mathit{vals}) & ts \texttt{ is Intersection} \\
M_{\texttt{Difference}}(ts, s, \mathit{vals}) & ts \texttt{ is Difference}
\end{cases}
\end{aligned}$$

The denotation of the `Task` construct is as follows.

$$\begin{aligned}
M_{\texttt{Task}} &: \texttt{Task} \to \mathbb{T} \\
M_{\texttt{Task}}[t : \texttt{Task}] &\triangleq \begin{cases}
\{t.\mathit{label}\} & t \texttt{ is ExactTask} \\
\mathit{similar}(t.\mathit{label}, \mathit{defSim}) & t \texttt{ is DefSimTask} \\
\mathit{similar}(t.\mathit{label}, t.\mathit{sim}.\mathit{value}) & t \texttt{ is SimTask}
\end{cases}
\end{aligned}$$

A specimen of `Task` denotes a PQL task, i.e., a non-empty set of character strings. The meaning of a specimen t of `ExactTask` is a singleton that contains label $t.\mathit{label}$. If t is a specimen of `DefSimTask` or `SimTask`, then its denotation is the set of all labels in $\mathbb{C}$ that have similarity scores with $t.\mathit{label}$ greater than or equal to the default similarity threshold defSim or $t.\mathit{sim}.\mathit{value}$, respectively, where $\mathit{defSim} \in [0\,..\,1]$ is a global constant.

A set of tasks can be constructed by selecting tasks from a given set of tasks, where the selection is implemented using a behavioral relation. To this end, one can use `SetOfTasksConstruction`. Next, we give denotations of `UnaryPredicateConstruction` and `BinaryPredicateConstruction`.

$$M_{\texttt{UnaryPredicateConstruction}}[upc : \texttt{UnaryPredicateConstruction},\ s : \mathbb{S},\ vals : \wp(\mathbb{V} \times \wp(\mathbb{T}))] \triangleq \begin{cases} \{t \in M_{\texttt{SetOfTasks}}(upc.tasks, s, vals) \mid canOccur(s,t)\} & upc.name \texttt{ is CanOccur} \\ \{t \in M_{\texttt{SetOfTasks}}(upc.tasks, s, vals) \mid alwaysOccurs(s,t)\} & upc.name \texttt{ is AlwaysOccurs} \end{cases}$$

$$M_{\texttt{BinaryPredicateConstruction}}[bpc : \texttt{BinaryPredicateConstruction},\ s : \mathbb{S},\ vals : \wp(\mathbb{V} \times \wp(\mathbb{T}))] \triangleq \begin{cases} \{t_1 \in M_{\texttt{SetOfTasks}}(bpc.tasks_1, s, vals) \mid \exists\, t_2 \in M_{\texttt{SetOfTasks}}(bpc.tasks_2, s, vals) : canConflict(s, t_1, t_2)\} & (bpc.q \texttt{ is Any}) \wedge (bpc.name \texttt{ is CanConflict}) \\ \{t_1 \in M_{\texttt{SetOfTasks}}(bpc.tasks_1, s, vals) \mid \forall\, t_2 \in M_{\texttt{SetOfTasks}}(bpc.tasks_2, s, vals) : canConflict(s, t_1, t_2)\} & (bpc.q \texttt{ is All}) \wedge (bpc.name \texttt{ is CanConflict}) \\ \ldots \end{cases}$$

The use of a binary behavioral relation is determined by *bpc.name*. For example, *bpc.name* of type `CanConflict` calls for the use of the *CanConflict* relation. Similarly, a specimen *bpc.name* of type `CanCooccur`, `Conflict`, `Cooccur`, `TotalCausal`, and `TotalConcurrent`, signifies the use of the *canCooccur*, *conflict*, *cooccur*, *totalCausal*, and *totalConcurrent* relation, respectively (not shown above). The denotations of *canOccur* and *alwaysOccurs* and the denotations of all the binary predicates of PQL are proposed in Section 5.5.

As an example of constructing a set of tasks from other sets, the denotation of the `Difference` construct is proposed below.

$$M_{\texttt{Difference}}[d : \texttt{Difference},\ s : \mathbb{S},\ vals : \wp(\mathbb{V} \times \wp(\mathbb{T}))] \triangleq \begin{cases} M_{\texttt{SetOfTasks}}(d_1, s, vals) \setminus M_{\texttt{SetOfTasks}}(d_2, s, vals) & |d| = 2 \\ M_{\texttt{SetOfTasks}}(d_1, s, vals) \setminus M_{\texttt{Difference}}(suffix(d, 2), s, vals) & |d| > 2 \end{cases}$$

The PQL grammar in Appendix C specifies that brackets, i.e., `'()'`, have the highest priority, then difference $\setminus$, then intersection $\cap$, and finally union $\cup$. Hence, expression $A \cup B \cap C \setminus (D \cup E) \setminus F$ gets evaluated as $A \cup (B \cap (C \setminus ((D \cup E) \setminus F)))$.

The denotation of `Predicate` is as follows.

$$M_{\texttt{Predicate}} \quad : \quad \texttt{Predicate} \times \mathbb{S} \times \wp(\mathbb{V} \times \wp(\mathbb{T})) \rightarrow \{true, false\}$$

The `Predicate` construct is defined as a choice production of ten alternatives. In what follows, we discuss the meaning of six alternatives, whereas detailed discussions of the remaining four alternatives can be found in Appendix D.

`TruthValue` refers to values of *true* and *false* from Boolean logic.

$$M_{\texttt{TruthValue}}[p : \texttt{TruthValue}, s : \mathbb{S}, vals : \wp(\mathbb{V} \times \wp(\mathbb{T}))] \triangleq \begin{cases} true & p \texttt{ is True} \\ false & p \texttt{ is False} \end{cases}$$

A predicate can be defined as a negation, conjunction, or disjunction of predicates.

$$\begin{aligned}
M_{\texttt{Negation}}[p:\texttt{Negation}, s:\mathbb{S}, vals:\wp(\mathbb{V}\times\wp(\mathbb{T}))] &\triangleq \neg\, M_{\texttt{Predicate}}(p.pred, s, vals)\\
M_{\texttt{Conjunction}}[p:\texttt{Conjunction}, s:\mathbb{S}, vals:\wp(\mathbb{V}\times\wp(\mathbb{T}))] &\triangleq \bigwedge_{i\in[1..|p|]} M_{\texttt{Predicate}}(p_i, s, vals)\\
M_{\texttt{Disjunction}}[p:\texttt{Disjunction}, s:\mathbb{S}, vals:\wp(\mathbb{V}\times\wp(\mathbb{T}))] &\triangleq \bigvee_{i\in[1..|p|]} M_{\texttt{Predicate}}(p_i, s, vals)
\end{aligned}$$

The PQL grammar in Appendix C specifies that brackets, i.e., '()', have the highest priority, then negation $\neg$, then conjunction $\wedge$, and finally disjunction $\vee$. Thus, expression $\neg(a \vee b \wedge c) \vee d \wedge e$ is evaluated as $(\neg(a \vee (b \wedge c))) \vee (d \wedge e)$.

One can test whether a predicate evaluates to *true* or *false* as follows.

$$\begin{aligned}
M_{\texttt{LogicalTest}}[p:\texttt{LogicalTest},\\ s:\mathbb{S}, vals:\wp(\mathbb{V}\times\wp(\mathbb{T}))] \triangleq \begin{cases} M_{\texttt{Predicate}}(p.pred, s, vals) & p \texttt{ is IsTrue} \vee p \texttt{ is IsNotFalse}\\ \neg\, M_{\texttt{Predicate}}(p.pred, s, vals) & p \texttt{ is IsFalse} \vee p \texttt{ is IsNotTrue}\end{cases}
\end{aligned}$$

The `TaskInSetOfTasks` construct tests if a task is a member of the given set.

$$\begin{aligned}
M_{\texttt{TaskInSetOfTasks}}[p:\texttt{TaskInSetOfTasks},\\ s:\mathbb{S}, vals:\wp(\mathbb{V}\times\wp(\mathbb{T}))] \triangleq M_{\texttt{Task}}(p.task) \in M_{\texttt{SetOfTasks}}(p.tasks, s, vals)
\end{aligned}$$

In Table 5, we explain the meaning of query *Q3* from Section 2 when matched to the model in Fig. 1. The numbers in the first column in the table refer to the constructs represented as nodes in the abstract syntax tree in Fig. 6. For example, number 1 in the first column refers to node "Task (1)" in the figure. The second column contains denotations of the corresponding constructs. Finally, the third column contains our explanations. Assuming that the model in Fig. 1 is at a location in the set of locations denoted by node 18, it and its requested attributes are included in the result of *Q3*. Note that s in the last row of the table is the system in Fig. 3, the Petri net version of the model used to evaluate the behavioral predicates included in the query.

Table 5: Meaning of PQL query *Q3* from Section 2 explained using Fig. 6.

No.	**Denotation**	**Explanation**
1	{"Start inventory recount"}	This is a specimen of `ExactTask` and, thus, it denotes a singleton of its label.
2	{"No variance is determined"}	This is a specimen of `ExactTask` and, thus, it denotes a singleton of its label.
3	{{"Start inventory recount"}, {"No variance is determined"}}	This set of tasks is specified as a fixed value composed of tasks denoted by 1 and 2.
4	(x, {{"Start inventory recount"}, {"No variance is determined"}})	This variable associates symbolic name x with the set of tasks denoted by 3.
5	{"Clear differences", "Clear differences WM", "Clear differences IM"}	This is a specimen of `DefSimTask` and, thus, denotes the set of all labels in the repository that are similar to its label.
6	{{"Clear differences", "Clear differences WM", "Clear differences IM"}}	This set of tasks is specified as a fixed value composed of the task denoted by 5.
7	(y, {{"Clear differences", "Clear differences WM", "Clear differences IM"}})	This variable associates symbolic name y with the set of tasks denoted by 6.
8	{{"Clear differences", "Clear differences WM", "Clear differences IM"}}	This set of tasks is specified as the value associated by 7 with symbolic name y.
9	{"Difference is posted", "Difference is posted to interface", "Difference is posted to IM"}	This is a specimen of `DefSimTask` and, thus, denotes the set of all labels in the repository that are similar to its label.

10	{{“Difference is posted to IM”,“Difference is posted to interface”,“Difference is posted”}}	This set of tasks is specified as a fixed value composed of the task denoted by 9.
11	{{“Difference is posted”,“Difference is posted to interface”,“Difference is posted to IM”}, {“Clear differences”,“Clear differences WM”,“Clear differences IM”}}	This set of tasks is computed as the union of sets denoted by 8 and 10.
12	(z, {{“Difference is posted”,“Difference is posted to interface”,“Difference is posted to IM”}, {“Clear differences”,“Clear differences WM”,“Clear differences IM”}})	This variable associates symbolic name z with the set of tasks denoted by 11.
13	{{“Storage type is to be blocked for inventory”}, {“Storage type block”}, {“Storage type is blocked”}, ... }	This set of tasks is defined by all the labels in the model from Fig. 1; each function and event defines one PQL task in the set.
14	{{“Storage bin is blocked”}, {“System inventory record is created”}, {“Physical inventory is active”}, {“Print inventory list”}, {“Physical inventory list is printed”}, {“Enter count results”}}	The set of tasks from the set denoted by 13 that occur in every instance of the model in Fig. 1; the checks are performed on the workflow system in Fig. 3 using the result of Lemma 5.8.
15	(w, {{“Storage bin is blocked”}, {“System inventory record is created”}, {“Physical inventory is active”}, {“Print inventory list”}, {“Physical inventory list is printed”}, {“Enter count results”}})	This variable associates symbolic name z with the set of tasks denoted by 14.
16	{(x, {{“Start inventory recount”}, {“No variance is determined”}}), ... }	The set contains values of all the variables used in the query, i.e., it contains denotations of 4, 7, 12, and 15.
17	$\mathbb{A}$	All attribute names in the repository.
18	{/SAP-R3-EPC-Repo}	The set composed of one location specified by the location path "/SAP-R3-EPC-Repo".
19	{{“Start inventory recount”}, {“No variance is determined”}}	This set of tasks is specified as the value associated by 4 with symbolic name x.
20	{{“Difference is posted”,“Difference is posted to interface”,“Difference is posted to IM”}, {“Clear differences”,“Clear differences WM”,“Clear differences IM”}}	This set of tasks is specified as the value associated by 12 with symbolic name z.
21	{{“Clear differences”,“Clear differences WM”,“Clear differences IM”}, {“No variance is determined”}, {“Start inventory recount”}, {“Difference is posted”,“Difference is posted to interface”,“Difference is posted to IM”}}	This set of tasks is computed as the union of sets denoted by 19 and 20.
22	*true*; every task in the set denoted by 21 occurs in at least one instance of the model.	The checks are performed on the system in Fig. 3 using the result of Lemma 5.7.
23	{“Start inventory recount”}	This is a specimen of ExactTask and, thus, it denotes a singleton of its label.
24	{{“System inventory record is created”}, {“Print inventory list”}, {“Physical inventory is active”}, {“Physical inventory list is printed”}, {“Storage bin is blocked”}, {“Enter count results”}}	This set of tasks is specified as the value associated by 15 with symbolic name w.
25	*false*	The task denoted by 23 is not a member of the set of tasks denoted by 24.
26	*true*	Negation of denotation of 25.
27	{“No variance is determined”}	This is a specimen of ExactTask and, thus, it denotes a singleton of its label.
28	{{“Clear differences”,“Clear differences WM”,“Clear differences IM”}}	This set of tasks is specified as the value associated by 8 with symbolic name y.
29	*true*; the task denoted by 27 is in conflict with every task in the set denoted by 28.	The checks are performed on the workflow system in Fig. 3 based on Definition 5.6 using the technique proposed in [20].
30	{“Start inventory recount”}	This is a specimen of ExactTask and, thus, it denotes a singleton of its label.

31	{{"Difference is posted", "Difference is posted to interface", "Difference is posted to IM"}, {"Clear differences", "Clear differences WM", "Clear differences IM"}}	This set of tasks is specified as the value associated by 12 with symbolic name z.
32	*true*; the task denoted by 30 is in the total causal relation with every task in the set denoted by 31.	The checks are performed on the workflow system in Fig. 3 based on Definition 5.6 using the technique proposed in [20].
33	*true*	Conjunction of 22, 26, 29, and 32.
34	$\{(s, \{(\text{Author}, \text{SAP}), ...\}), ...\}$	A result of query *Q3* from Section 2.

5.5. Predicate Definitions and Computation

Section 5.5.1 presents formal definitions of the basic PQL predicates, while Section 5.5.2 discusses their computations.

5.5.1. Definitions

Before presenting definitions of the predicates over PQL tasks, as demanded by the semantics of PQL, we define them over character strings.

Predicates over character strings. Given a workflow system, the *canOccur* and *alwaysOccurs* unary predicates over character strings are defined as follows.

Definition 5.2 (Can occur and always occurs).
Let $S := (P, T, F, \lambda, M)$ be a workflow system and let $x \in \mathbb{C}$ be a non-empty character string. Then, x *can occur* in S, denoted by $canOccur(S, x)$, iff there is a label execution η of S such that $x \in \eta$, while x *always occurs* in S, denoted by $alwaysOccurs(S, x)$, iff for every label execution η of S it holds that $x \in \eta$.[4] ⌟

Hence, a character string $x \in \mathbb{C}$ can occur in a workflow system S iff one can observe x in some execution of S, i.e., an activity denoted by x can be performed in some business scenario captured in S. In turn, x always occurs in S, iff it is observed in every execution of S. Next, we define the binary predicates of PQL, starting with the basic conflict and co-occurrence predicates.

Definition 5.3 (Basic conflict and co-occurrence).
Let $S := (P, T, F, \lambda, M)$ be a workflow system and let $x, y \in \mathbb{C}$ be two non-empty character strings. Then, x *can conflict with* y in S, denoted by $canConflict(S, x, y)$, iff there is a label execution η of S such that $x \in \eta$ and $y \notin \eta$, while x and y *can co-occur* in S, denoted by $canCooccur(S, x, y)$, iff there is a label execution η of S such that $x \in \eta$ and $y \in \eta$. ⌟

Thus, x can conflict with y iff there is an execution of S that performs x but not y, whereas x and y can co-occur iff they both can occur in an execution of S. The 4C spectrum uses the basic conflict and co-occurrence relations as building blocks of other relations, two of which are among the selected PQL predicates.

Definition 5.4 (Conflict and co-occurrence).
Let $S := (P, T, F, \lambda, M)$ be a workflow system and let $x, y \in \mathbb{C}$ be two non-empty character strings. Then, x and y are in *conflict* in S, denoted by $conflict(S, x, y)$,

[4]Given a sequence σ, $x \in \sigma$ denotes the fact that x is an element of σ.

iff $\mathit{canConflict}(S,x,y) \wedge \mathit{canConflict}(S,y,x) \wedge \neg\, \mathit{canCooccur}(S,x,y)$, while x and y *co-occur* in S, denoted by $\mathit{cooccur}(S,x,y)$, iff $\neg\, \mathit{canConflict}(S,x,y) \wedge \neg\, \mathit{canConflict}(S,y,x) \wedge \mathit{canCooccur}(S,x,y)$. ⌟

Hence, x and y are in conflict in S iff they can be observed in some executions of S but never together in the same execution. In contrast, x and y co-occur in S iff they can be observed together in some executions of S, x is never observed in an execution that does not include an occurrence of y, and vice versa, y is never observed in an execution that does not include an occurrence of x.

Consider workflow system S in Fig. 4(a). The expression $\mathit{canOccur}(S,\mathtt{a}') \wedge \neg\, \mathit{alwaysOccurs}(S,\mathtt{a}')$ evaluates to *true*. There exists a label execution $\eta_1 := \langle \mathtt{a}',\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{e}\rangle$ of S that justifies that $\mathtt{a}'$ can occur in S, and a label execution $\eta_2 := \langle \mathtt{a}'',\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{e}\rangle$ of S that justifies that $\mathtt{a}'$ does not always occur in S. Note that strings $\mathtt{b}$, $\mathtt{c}$, $\mathtt{d}$, and $\mathtt{e}$, always occur in S as they are present in all the four label executions of S. Furthermore, both $\mathit{canConflict}(S,\mathtt{b},\mathtt{a}')$ and $\mathit{canCooccur}(S,\mathtt{b},\mathtt{a}')$ evaluate to *true*, as $\mathtt{b}$ can be observed without $\mathtt{a}'$, for example in η_2, and with $\mathtt{a}'$, for example in η_1. Note that $\mathtt{a}$ and $\mathtt{a}'$ are in conflict, i.e., $\mathit{conflict}(S,\mathtt{a},\mathtt{a}')$ evaluates to *true*, as these two strings are never observed together. Finally, $\mathtt{b}$ and $\mathtt{e}$ co-occur in S, i.e., $\mathit{cooccur}(S,\mathtt{b},\mathtt{e})$ evaluates to *true*.

The total causal and total concurrent relations are defined as follows.

Definition 5.5 (Total causality and total concurrency).
Let $S := (P,T,F,\lambda,M)$ be a workflow system and let $x,y \in \mathbb{C}$ be two non-empty character strings. Then, x and y are *total causal* in S, denoted by $\mathit{totalCausal}(S,x,y)$, iff $\forall\, \pi \in \Delta_S(x,y)\ \forall\, e_1 \in E_\pi\ \forall\, e_2 \in E_\pi : (e_1 \neq e_2 \wedge \lambda(\rho_\pi(e_1)) = x \wedge \lambda(\rho_\pi(e_2)) = y) \Rightarrow e_1 \rightarrowtail_\pi e_2$, while x and y are *total concurrent* in S, denoted by $\mathit{totalConcurrent}(S,x,y)$, iff $\forall\, \pi \in \Delta_S(x,y)\ \forall\, e_1 \in E_\pi\ \forall\, e_2 \in E_\pi : (e_1 \neq e_2 \wedge \lambda(\rho_\pi(e_1)) = x \wedge \lambda(\rho_\pi(e_2)) = y) \Rightarrow e_1 \parallel_\pi e_2$. ⌟

For example, $\mathit{totalCausal}(S,\mathtt{a}',\mathtt{c})$ holds *true* for system S in Fig. 4(a), i.e., in every execution of S in which $\mathtt{a}'$ and $\mathtt{c}$ both occur, every occurrence of $\mathtt{a}'$ precedes every occurrence of $\mathtt{c}$. In addition, $\mathit{totalConcurrent}(S,\mathtt{c},\mathtt{d})$ holds *true* in S.

Predicates over PQL tasks. Next, we lift the PQL predicates from input labels to input PQL tasks, given as sets of character strings. The concept of a PQL task allows handling several distinct labels as if they all represent the same activity. For example, two character strings $\mathtt{a}'$:=“`process payment by cash`” and $\mathtt{a}''$:=“`process payment by check`” may be seen as sufficiently similar to represent activity $\mathtt{a}$:=“`process payment`”. To implement this intuition, PQL uses the label unification principle proposed in [20].

Given a system S and a character string $x \in \mathbb{C}$, *label unification* of x in S is a transformation of S into system S' that is behaviorally equivalent to S but in which every occurrence of x is guaranteed to be triggered by a dedicated transition. Fig. 7 shows the result of performing label unification for labels $\mathtt{a}'$ and $\mathtt{a}''$ in the system in Fig. 4(a). The fresh elements introduced during the unification are highlighted in gray, while the fresh arcs, in addition, are depicted using the dashed lines. Unlike in the system in Fig. 4(a), transitions with labels $\mathtt{a}'$ and $\mathtt{a}''$ cannot occur in executions of the system in Fig. 7. In the transformed

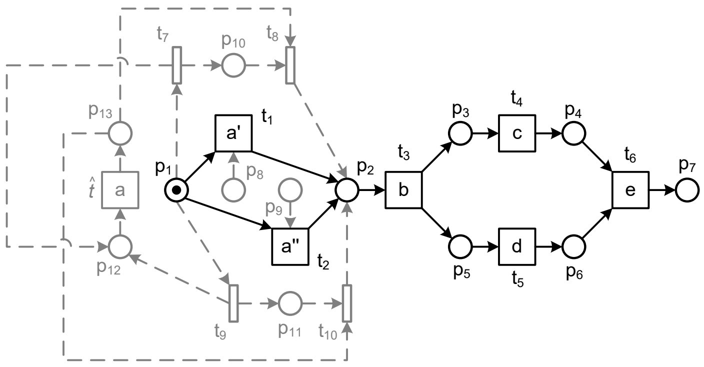


Fig. 7: The result of label unification of labels $\{\mathsf{a}, \mathsf{a}'\}$ in the system in Fig. 4(a).

system, tokens from the input places of the "forbidden" transitions t_1 and t_2 get rerouted to enable the fresh transition $\hat{t}$ that has label a; transition $\hat{t}$ is called the *solitary* transition for labels a' and a''. By $\mathit{unify}(S, X) = (S', \hat{t})$, $X \in \wp_{\geq 1}(\mathbb{C})$, we denote the result of the label unification of X in S, where S' is the resulting system with the solitary transition $\hat{t}$ for labels in X. By definition, the order of applying different label unifications has no impact on the resulting system.

Finally, the predicates over PQL tasks are defined as follows.

Definition 5.6 (Predicates over PQL tasks).
Let $S := (P, T, F, \lambda, M)$ be a workflow system and let $X, Y \in \wp_{\geq 1}(\mathbb{C})$ be two non-empty sets of non-empty character strings.

- X *can occur* in S, denoted by $\mathit{canOccur}(S, X)$, iff it holds that $\mathit{canOccur}(S', x)$, where $\mathit{unify}(S, X) = (S', x)$.
- X *always occurs* in S, denoted by $\mathit{alwaysOccurs}(S, X)$, iff it holds that $\mathit{alwaysOccurs}(S', x)$, where $\mathit{unify}(S, X) = (S', x)$.
- X *can conflict with* Y in S, denoted by $\mathit{canConflict}(S, X, Y)$, iff it holds that $\mathit{canConflict}(S'', x, y)$, where $\mathit{unify}(S, X) = (S', x)$ and $\mathit{unify}(S', Y) = (S'', y)$.
- X and Y *can co-occur* in S, denoted by $\mathit{canCooccur}(S, X, Y)$, iff it holds that $\mathit{canCooccur}(S'', x, y)$, where $\mathit{unify}(S, X) = (S', x)$ and $\mathit{unify}(S', Y) = (S'', y)$.
- X and Y are in *conflict* in S, denoted by $\mathit{conflict}(S, X, Y)$, iff it holds that $\mathit{conflict}(S'', x, y)$, where $\mathit{unify}(S, X) = (S', x)$ and $\mathit{unify}(S', Y) = (S'', y)$.
- X and Y *co-occur* in S, denoted by $\mathit{cooccur}(S, X, Y)$, iff it holds that $\mathit{cooccur}(S'', x, y)$, where $\mathit{unify}(S, X) = (S', x)$ and $\mathit{unify}(S', Y) = (S'', y)$.
- X and Y are *total causal* in S, denoted by $\mathit{totalCausal}(S, X, Y)$, iff it holds that $\mathit{totalCausal}(S'', x, y)$, where $\mathit{unify}(S, X) = (S', x)$ and $\mathit{unify}(S', Y) = (S'', y)$.
- X and Y are *total concurrent* in S, denoted by $\mathit{totalConcurrent}(S, X, Y)$, iff it holds that $\mathit{totalConcurrent}(S'', x, y)$, where $\mathit{unify}(S, X) = (S', x)$ and $\mathit{unify}(S', Y) = (S'', y)$.

⌟

Using predicates over PQL tasks instead of strings, one can verify, for example, that $\mathit{alwaysOccurs}(S, \{\mathsf{a}', \mathsf{a}''\})$, where S is the system in Fig. 4(a), holds *true*. Thus, the exploratory query "`SELECT * FROM * WHERE AlwaysOccurs(~a);`" should retrieve the system in Fig. 4(a), given that `~a` evaluates to $\{\mathsf{a}', \mathsf{a}''\}$.

5.5.2. Computations

Definition 5.6 specifies the eight selected predicates over PQL tasks in terms of the corresponding predicates over transitions. Techniques for computing the *canConflict*, *canCooccur*, and *totalCausal* predicates are available [20]. Using *canConflict* and *canCooccur*, one can compute the *conflict* and *cooccur* predicates. Next, we propose techniques that given a sound workflow system $S := (P, T, F, \lambda, M)$ compute whether $t \in T$ *can occur* in S, $t \in T$ *always occurs* in S, and $t_1 \in T$ and $t_2 \in T$ are *total concurrent* in S.

Computations of predicates are performed on systems that may result from label unifications, refer to Section 5.5.1. A label unification in a sound workflow system often leads to a system that is not even a workflow system due to the introduction of dead transitions, where a dead transition is a transition that is not part of any occurrence sequence of the system; for example, transitions t_1 and t_2 in Fig. 7 are dead. Dead transitions are kept in the resulting systems as they may be required to perform subsequent label unifications. However, once all the unifications are applied, dead transitions can be removed to result in a sound workflow system; this trivially follows from the definition of label unification. Thus, in what follows, we hold discussions for (sound) workflow systems.

Can occur. A transition $t \in T$ *can occur* in a workflow system $S := (P, T, F, \lambda, M)$ iff there exists an execution σ of S such that $t \in \sigma$. One can check whether a transition can occur in a workflow system by solving a reachability problem on a transformed version of the system. Given a system and a marking, the reachability problem consists of deciding if an occurrence sequence of the system leads to the marking. The reachability problem is decidable [22].

Lemma 5.7 (Can occur transition).
Let $S := (P, T, F, \lambda, M)$ be a workflow system with the sink place $o \in P$. A transition $t \in T$ can occur in S iff there is an occurrence sequence σ of $S' := (P \cup \{p'\}, T \cup \{t'\}, F \cup \{(p', t')\} \cup \{(p, t') \mid p \in \bullet t\} \cup \{(t', p) \mid p \in t\bullet\}, \lambda \cup \{(t', \epsilon)\}, M \uplus [p'])$, where $p' \notin P$ and $t' \notin T$ are a fresh place and a fresh transition, respectively, such that t' is an element of σ, i.e., $t' \in \sigma$, and σ leads to $[o]$. ⌟

The proof follows from the construction of S'. If t can occur in S, then there is an execution γ of S that contains t at some position j of γ. A sequence of transitions obtained from γ by replacing t at position j with t' is an occurrence sequence of S' that leads to $[o]$. Let σ be an occurrence sequence of S' that leads to $[o]$ and $t' \in \sigma$. Then, the sequence obtained from σ by replacing t' with t and keeping the order of the other elements is an execution of S.

Always occurs. A transition $t \in T$ *always occurs* in a sound workflow system $S := (P, T, F, \lambda, M)$ iff for every execution σ of S it holds that $t \in \sigma$. Again, one can check whether a transition always occurs in a sound workflow system by solving a reachability problem on a transformed version of the system.

Lemma 5.8 (Always occurs transition).
Let $S := (P, T, F, \lambda, M)$ be a workflow system with the sink place $o \in P$. A transition $t \in T$ always occurs in S iff there is no occurrence sequence σ of

$S' := (P \cup \{p'\}, T, F \cup \{(p',t)\}, \lambda, M \uplus [p'])$, where $p' \notin P$ is a fresh place, that leads to $[p',o]$. ⌟

The proof, again, follows from the construction of S'. If t always occurs in S, then every execution of S contains t. Let γ be an execution of S without t, then γ is an occurrence sequence of S' that leads to $[p',o]$. Let σ be an occurrence sequence of S' that leads to $[p',o]$. Then, σ is an execution of S without t.

Total concurrent. Let $S := (P,T,F,\lambda,M)$ be a workflow system. By $\Xi_S(t_1,t_2)$, where $t_1,t_2 \in T$, we denote the set $\{\pi \in \Pi_S \mid \exists\, e_1,e_2 \in E_\pi : \rho_\pi(e_1) = t_1 \;\wedge\; \rho_\pi(e_2) = t_2\}$, i.e., the set of all processes of S that contain events that describe the occurrences of transitions t_1 and t_2. Transitions $t_1 \in T$ and $t_2 \in T$ are *total concurrent* in a sound workflow system $S := (P,T,F,\lambda,M)$ iff $\forall\, \pi \in \Xi_S(t_1,t_2)\; \forall\, e_1 \in E_\pi\; \forall\, e_2 \in E_\pi : (e_1 \neq e_2 \;\wedge\; \rho_\pi(e_1) = t_1 \;\wedge\; \rho_\pi(e_2) = t_2) \Rightarrow e_1 \parallel_\pi e_2$.

Let $U := (B,E,G,\rho)$ be a complete prefix of the unfolding of a system $S := (P,T,F,\lambda,M)$. Then, it holds that $path(U,e_1,e_2)$ iff $(e_1,e_2) \in G^+$, or $(e_2,e_1) \in G^+$, or there exists a sequence of cutoffs $c \in cutoffs(U)^*$ such that (i) $\exists\, b \in Cut(\lceil c_1 \rceil) : (e_1,b) \in G^+$, (ii) $\exists\, b \in Cut(\lceil corr(c_{|c|}) \rceil) : (b,e_2) \in G^+$, and (iii) $\forall\, i \in [1..(|c|-1)] \exists\, b_1 \in Cut(\lceil corr(c_i) \rceil) \exists\, b_2 \in Cut(\lceil c_{i+1} \rceil) : (b_1,b_2) \in G^*$.

Lemma 5.9 (Total concurrent transitions).
Let $S := (P,T,F,\lambda,M)$ be a sound workflow system. Let $U := (B,E,G,\rho)$ be a complete prefix of the unfolding of S. Transitions $t_1,t_2 \in T$ are total concurrent in S iff $\forall e_1 \in E\, \forall e_2 \in E : (e_1 \neq e_2 \wedge \rho(e_1) = t_1 \wedge \rho(e_2) = t_2) \Rightarrow \neg\, path(U,e_1,e_2)$. ⌟

Let us assume that t_1 and t_2 are total concurrent in S and there exist two distinct events e_1 and e_2 that describe occurrences of t_1 and t_2, respectively, such that $path(U,e_1,e_2)$. Then, there is a process in $\Xi_S(t_1,t_2)$ that contains two distinct causal events that refer to t_1 and t_2. This follows immediately from the definition of a complete prefix of the unfolding, cf. [25], and the fact that every occurrence sequence of S can be extended to an execution. If for every two distinct events e_1 and e_2 that describe occurrences of t_1 and t_2 it holds that $\neg\, path(U,e_1,e_2)$, then t_1 and t_2 are total concurrent in S, as according to the definition of a complete prefix of the unfolding there exists no process in $\Xi_S(t_1,t_2)$ that evidences that some occurrences of t_1 and t_2 are causal.

Note that one can always construct a complete prefix of the unfolding of a bounded system [25], i.e., a system with a finite number of reachable states, while a sound workflow system is guaranteed to be bounded [35]. Using the prefix in Fig. 5(b) and Lemma 5.9, one can verify whether two given transitions are total concurrent in the system S in Fig. 5(a). For example, transitions t_3 and t_5 are total concurrent in S. Transitions t_3 and t_6 are not total concurrent in S because of events e_3 and e'_6 and the fact that $(e_3,e'_6) \in G^+$. Transitions t_9 and t_{11} are also not total concurrent in S because of events e_9 and e_{11} and the sequence of cutoff events $\langle e'_7 \rangle$; note that $(e_9,b'_9) \in G^+$ and $(b_9,e_{11}) \in G^+$.

5.6. Example Queries

To exemplify PQL, we use an example process repository consisting of ten process models sourced from the SAP R/3 reference model [5] and Polyvyanyy's

Ph.D. thesis [15] and depicted in BPMN in Fig. 8. For simplicity, the models in Fig. 8 use alphabet letters as abstract task labels. In addition, ID, version, date created, and author attribute is shown in the text annotation under each model. Note that model 9 in the figure Fig. 8 is the model from Fig. 2.

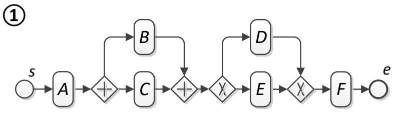
This repository can be formalized as the $(S, A, L, val, loc, \lesssim)$ tuple, see Definition 5.1, where S is the set of Petri net systems $\{s_1, \ldots, s_{10}\}$, s_i, $i \in \{1..10\}$, where s_i is obtained by translating model i from BPMN to Petri nets [33], A :={ID, Version, Date, Author}, L := {/, `/Ten-Models-BPMN`, `/SAP-R3-EPC-Repo`}, val := $\{(s_1, \text{ID}, 1), (s_1, \text{Version}, 1.0), (s_1, \text{Date}, \text{01-June-2017}), \ldots\}$, $loc := \cup_{i \in \{1..10\}}\{(s_i,$ `/Ten-Models-BPMN`$)\}$, and $\lesssim$ is such that $(l, l) \in \lesssim$ for every $l \in L$, and it holds that (`/Ten-Models-BPMN`, /) $\in \lesssim$ and (`/SAP-R3-EPC-Repo`, /) $\in \lesssim$.

All models in the repository are sound and have various structural characteristics. Models 1 to 5 are acyclic, while models 6 to 10 contain cycles. Models 1, 2, 6, and 10 are well-structured, where a model is *well-structured* if and only if every node with multiple outgoing arcs (a split) has a corresponding node with multiple incoming arcs (a join), and vice versa, such that the set of nodes between the split and the join induces a Single-Entry-Single-Exit component [63] [15]. Models 3, 4, 5, 7, 8, and 9 are unstructured. Model 7, 8, and 9 can be mapped to well-structured models. For example, model 10 is equivalent to model 9 and is well-structured. Note that models 3 to 5 are inherently unstructured. That is, if the concurrency relations between process tasks must be preserved, they do not have equivalent well-structured representations [15].

Ten sample PQL queries are listed below:

Q1.
```
SELECT "Author" FROM "/Ten-Models-BPMN"
WHERE CanOccur("D") AND Conflict("D","E");
```
Q2.
```
SELECT "Version" FROM "/Ten-Models-BPMN"
WHERE AlwaysOccurs("C") OR Cooccur("B","C");
```
Q3.
```
SELECT "Date" FROM "/Ten-Models-BPMN"
WHERE (CanOccur("G") AND (NOT Conflict("E","G"))) OR
      (TotalConcurrent("C","D") AND AlwaysOccurs("D"));
```
Q4.
```
SELECT "Author","Version" FROM "/Ten-Models-BPMN"
WHERE CanOccur({"F","G"},ALL) AND AlwaysOccurs({"F","G"},ANY);
```
Q5.
```
SELECT "Version","Date" FROM "/Ten-Models-BPMN"
WHERE Cooccur("B",{"C","D"},ALL) AND TotalConcurrent("B",{"C","D"},ANY);
```
Q6.
```
SELECT "Version","Author" FROM "/Ten-Models-BPMN"
WHERE Conflict({"A","B"},{"E","F"},ANY) OR
      (Cooccur({"A","B"},{"E","F"},EACH) AND
      TotalCausal({"A","B"},{"E","F"},ALL));
```
Q7.
```
SELECT "Date","Author" FROM "/Ten-Models-BPMN"
WHERE "C" IN (GetTasksAlwaysOccurs({"C"}) UNION
      GetTasksTotalCausal({"C"},{"B","D"},ALL));
```
Q8.
```
SELECT "Date","Version" FROM "/Ten-Models-BPMN"
WHERE "G" IN (GetTasksCanOccur({"G"}) INTERSECT
      GetTasksConflict({"G"},{"D","E","F"},ANY));
```
Q9.
```
SELECT "Version","Date","Author" FROM "/Ten-Models-BPMN"
```

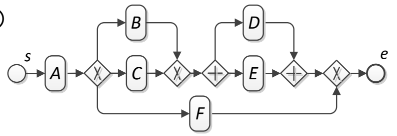
①



ID: 1 | Version: 1.0 | Date: 01-June-2017 | Author: SAP

✓ Q1 ✓ Q2 ☐ Q3 ☐ Q4 ☐ Q5 ✓ Q6 ✓ Q7 ☐ Q8 ☐ Q9 ✓ Q10

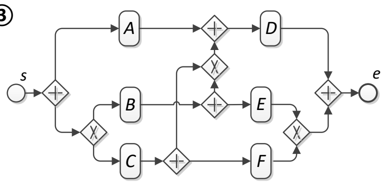
②



ID: 2 | Version: 1.1 | Date: 31-May-2017 | Author: SAP

☐ Q1 ☐ Q2 ☐ Q3 ☐ Q4 ☐ Q5 ✓ Q6 ✓ Q7 ☐ Q8 ☐ Q9 ✓ Q10

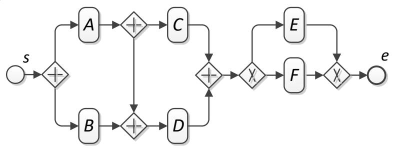
③



ID: 3 | Version: 2.0 | Date: 03-June-2017 | Author: Polyvyanyy

☐ Q1 ☐ Q2 ☐ Q3 ☐ Q4 ☐ Q5 ✓ Q6 ✓ Q7 ☐ Q8 ✓ Q9 ✓ Q10

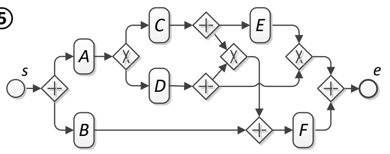
④



ID: 4 | Version: 1.2 | Date: 31-May-2017 | Author: SAP

☐ Q1 ✓ Q2 ✓ Q3 ☐ Q4 ✓ Q5 ☐ Q6 ✓ Q7 ☐ Q8 ✓ Q9 ☐ Q10

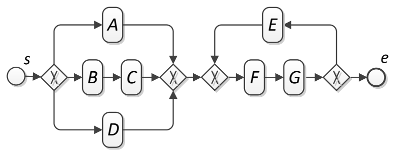
⑤



ID: 5 | Version: 1.2 | Date: 03-June-2017 | Author: Polyvyanyy

✓ Q1 ☐ Q2 ☐ Q3 ☐ Q4 ☐ Q5 ☐ Q6 ☐ Q7 ☐ Q8 ✓ Q9 ✓ Q10

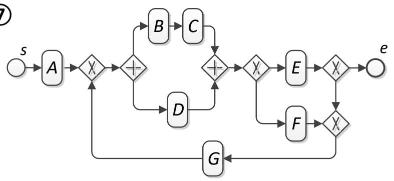
⑥



ID: 6 | Version: 2.0 | Date: 01-June-2017 | Author: SAP

☐ Q1 ✓ Q2 ✓ Q3 ✓ Q4 ☐ Q5 ☐ Q6 ☐ Q7 ☐ Q8 ✓ Q9 ✓ Q10

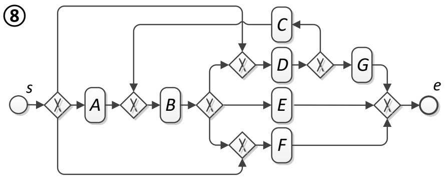
⑦



ID: 7 | Version: 1.2 | Date: 01-June-2017 | Author: SAP

☐ Q1 ✓ Q2 ✓ Q3 ☐ Q4 ☐ Q5 ☐ Q6 ✓ Q7 ☐ Q8 ✓ Q9 ☐ Q10

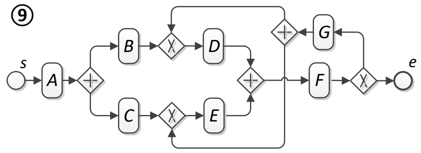
⑧



ID: 8 | Version: 2.0 | Date: 03-June-2017 | Author: SAP

☐ Q1 ☐ Q2 ☐ Q3 ☐ Q4 ☐ Q5 ☐ Q6 ☐ Q7 ✓ Q8 ☐ Q9 ☐ Q10

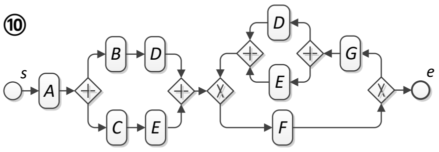
⑨



ID: 9 | Version: 1.2 | Date: 05-June-2017 | Author: Polyvyanyy

☐ Q1 ✓ Q2 ✓ Q3 ✓ Q4 ✓ Q5 ☐ Q6 ✓ Q7 ☐ Q8 ✓ Q9 ☐ Q10

⑩



ID: 10 | Version: 2.0 | Date: 05-June-2017 | Author: Polyvyanyy

☐ Q1 ✓ Q2 ✓ Q3 ✓ Q4 ✓ Q5 ☐ Q6 ✓ Q7 ☐ Q8 ✓ Q9 ☐ Q10

Fig. 8: A repository of ten process models captured in BPMN with their attributes information and results of evaluation of ten sample queries over each of these models.

```
     WHERE GetTasksCooccur({"A","B","C"},{"D","E"},ANY) NOT EQUALS
           GetTasksTotalConcurrent({"A","B","C"},{"D","E"},ANY);
Q10. SELECT * FROM "/Ten-Models-BPMN" WHERE ({"A","B","E","F"} EXCEPT
           GetTasksCooccur({"A","B","E","F"},{"C","D"},ALL)) OVERLAPS WITH
           GetTasksConflict({"A","B","E","F"},{"C","D"},ANY);
```

These queries exploit the various features of the PQL language, including atomic behavioral predicates, predicate macros, logical operations, construction of task sets using predicates, set operations, and set comparison operations. We executed the queries on the repository of ten models from Fig. 8. Each query was evaluated over each model according to the semantics of PQL defined in Sections 5.4 and 5.5. Each evaluation result indicates whether the model satisfies the condition specified in the `WHERE` clause of the query. In Fig. 8, this is depicted by a query name being ticked (satisfies) or not ticked (does not satisfy) under each model. Since models 9 and 10 are behaviorally equivalent, every query evaluation over these models yields the same result.

Table 6 shows the results of the queries. For each query, only the corresponding systems (and their attributes) that satisfy the `WHERE` condition of the query are retrieved. For example, Query *Q1* is designed to retrieve those process models where (i) task labeled `"D"` (task `D` for short) occurs in at least one process instance (`CanOccur("D")`) and (ii) there is no process instance in which tasks `D` and `E` both occur (`Conflict("D","E")`). The first condition holds in all models in Fig. 8, while the second condition holds in models 1 and 5 only. For example, in model 8, although tasks `D` and `E` are on two exclusive branches (subsequent to task `B`), both tasks may occur in one process instance since task `D` is part of a cycle. Another interesting example concerns query *Q3* and its condition `TotalConcurrent("C","D")`. This condition holds for a process model if and only if for every instance in which tasks `C` and `D` both occur, every occurrence of task `C` can be executed at the same time with every occurrence of task `D`. For example, as model 3 is unstructured, it is hard to evaluate this condition without an in-depth understanding of the processes the model describes.

6. Implementation

The querying method described in Section 5 has been implemented and is publicly available.[5] The implementation exhibits a well-defined application programming interface (API) to facilitate integration with other software products. This API can be accessed via command-line interfaces (CLIs) of two utilities: the PQL bot and the PQL tool. We refer to them as *PQL Tools*. Conjointly, they implement the *PQL environment*. The PQL bot is used to prepare models for querying, while the PQL tool executes PQL queries over the indexed models.

The PQL environment is implemented using Java, ANTLR, and MySQL. Java is chosen due to its "architecture-neutral and portable" principle. As a

[5]`https://github.com/processquerying/PQL.git`

Table 6: Results of ten sample queries over the repository of ten models from Fig. 8.

Query	Query result
Q1	{(s_1,{(Author, SAP)}), (s_5,{(Author, Polyvyanyy)})}
Q2	{(s_1,{(Version, 1.0)}), (s_4,{(Version, 1.2)}), (s_6,{(Version, 2.0)}), (s_7,{(Version, 1.2)}), (s_9,{(Version, 1.2)}), (s_{10},{(Version, 2.0)})}
Q3	{(s_4,{(Date, 31-May-2017)}), (s_6, {(Date, 01-June-2017)}), (s_7,{(Date, 01-June-2017)}), (s_9,{(Date, 05-June-2017)}), (s_{10},{(Date, 05-June-2017)})}
Q4	{(s_6,{(Author, SAP), (Version, 2.0)}), (s_9,{(Author, Polyvyanyy), (Version, 1.2)}), (s_{10},{(Author, Polyvyanyy), (Version, 2.0)})}
Q5	{(s_4,{(Version, 1.2), (Date, 31-May-2017)}), (s_9,{(Version, 1.2), (Date, 05-June-2017)}), (s_{10},{(Version, 2.0), (Date, 05-June-2017)})}
Q6	{(s_1,{(Version, 1.0), (Author, SAP)}), (s_2,{(Version, 1.1), (Author, SAP)}), (s_3,{(Version, 2.0), (Author, Polyvyanyy)})}
Q7	{(s_1,{(Date, 01-June-2017), (Author, SAP)}), (s_2,{(Date, 31-May-2017), (Author, SAP)}), (s_3,{(Date, 03-June-2017), (Author, Polyvyanyy)}), (s_4,{(Date, 31-May-2017), (Author, SAP)}), (s_7,{(Date, 01-June-2017), (Author, SAP)}), (s_9,{(Date, 05-June-2017), (Author, Polyvyanyy)}), (s_{10},{(Date, 05-June-2017), (Author, Polyvyanyy)})}
Q8	{(s_8, {(Date, 03-June-2017), (Version, 2.0)})}
Q9	{(s_3,{(Version, 2.0), (Date, 03-June-2017), (Author, Polyvyanyy)}), (s_4,{(Version, 1.2), (Date, 31-May-2017), (Author, SAP)}), (s_5, {(Version, 1.2), (Date, 03-June-2017), (Author, Polyvyanyy)}), (s_6, {(Version, 2.0), (Date, 01-June-2017), (Author, SAP)}), (s_7,{(Version, 1.2), (Date, 01-June-2017), (Author, SAP)}), (s_9,{(Version, 1.2), (Date, 05-June-2017), (Author, Polyvyanyy)}), (s_{10},{(Version, 2.0), (Date, 05-June-2017), (Author, Polyvyanyy)})}
Q10	{(s_1,{(ID, 1), (Version, 1.0), (Date, 01-June-2017), (Author, SAP)}), (s_2,{(ID, 2), (Version, 1.1), (Date, 31-May-2017), (Author, SAP)}), (s_3,{(ID, 3), (Version, 2.0), (Date, 03-June-2017), (Author, Polyvyanyy)}), (s_5,{(ID, 5), (Version, 1.2), (Date, 03-June-2017), (Author, Polyvyanyy)}), (s_6,{(ID, 6), (Version, 2.0), (Date, 01-June-2017), (Author, SAP)})}

result, the environment can be deployed on various platforms running different operating systems. The PQL tool uses an ANTLR [64] generated parser that can build and walk syntax trees of PQL queries. Given a context-free grammar expressed using extended Backus-Naur Form [65] as input, ANTLR generates the Java code of the grammar parser. The PQL grammar, which incorporates its abstract syntax and concrete syntax captured using ANTLR notation is listed in Appendix C. The PQL tool relies on a special index of behavioral relations, a data structure that improves the computation speed of behavioral relations at the cost of time for its construction and space for its storage. The tool uses this index at runtime to avoid having to compute PQL predicates every time a new query is issued. The index is stored in a MySQL relational database system.

PQL utilities can be configured to use a particular model checking tool for computing behavioral predicates, an information retrieval engine for assessment of label similarities, label similarity thresholds, the maximum number of threads used by the PQL tool when executing queries, the time that PQL bots sleep, i.e., stay idle, between two subsequent indexing jobs, and the maximal allowed time to index a single model. The PQL environment can be configured to use one of the three integrated information retrieval engines for scoring label similarities. These are Apache Lucene[6], Themis-IR [66], and the label similarity scoring approach based on the Levenshtein distance [67].

The PQL Bot. The PQL bot systematically indexes models stored by the PQL tool. Once a model is indexed, it can be matched to a query. One can start multiple PQL bot instances simultaneously to index multiple models in parallel. To construct an index, a PQL bot instance computes and stores all the behavioral predicates over all the PQL tasks of the model. A call to a PQL indexing routine takes as input a workflow system described in the Petri Net Markup Language (PNML) format [68].

The computation of a PQL predicate reduces to one of these three problems: the *reachability problem* [22], the *covering problem* [23], or the *structural analysis over a complete prefix* [24, 25] of the *unfolding* [26] of the system; refer to Section 5.5 and [20] for details. The PQL bot uses the LoLA tool version 2.0 [69] for solving the reachability and covering problems.[7] The PQL bot relies on the implementation of the algorithm by Esparza et al. [25] to construct finite complete prefixes of unfoldings available as part of the `jBPT` initiative [70].

Appendix E lists the CLI commands of the PQL bot and its sample output.

The PQL Tool. The PQL tool can store, index, delete, and query process models. The user can specify the number of computation threads to use when evaluating PQL queries. As a result of executing a PQL query, the tool returns a collection of matching models and PQL tasks (sets of activity labels) that have triggered the retrieval of the models. We refer the reader to Appendix E for the list of CLI commands of the PQL tool and its sample output.

7. Evaluation

Using the implementation presented in Section 6, we conducted experiments to assess the performance of PQL. Section 7.1 introduces the datasets used in the evaluation. Next, Section 7.2 and Section 7.3 discuss the evaluation of indexing and querying performance, respectively. The experiments were performed on a computer with 8GB of RAM and 3.4GHz quad-core Intel Core CPU (8 logical processors), running Windows 7 and JVM 1.7. Finally, in Appendix F, the reader can find additional discussions of the results of our experiments.

7.1. Datasets

Process models. The study was conducted using 493 industrial and 1,000 synthetic process models.[8] All the 1,493 models are sound workflow systems. We obtained the industrial models from the SAP R/3 Reference Model [5], a collection of 604 EPCs in various domains such as sales, production, and procurement

[6]`https://lucene.apache.org/`
[7]`http://service-technology.org/lola/`
[8]The models are available via `https://doi.org/10.26188/21937259`.

used to customize the SAP R/3 ERP system. We converted the EPCs to Petri net systems and completed them to workflow systems. Next, we filtered out the unsound systems, resulting in 493 sound models. We complemented this collection of real-life models with a collection of synthetic workflow systems, which we generated using the tool described by Yan et al. [71]. This tool takes a seed process model collection and generates models that share similar structural and label characteristics to the seed. We used the 604 EPCs in the SAP R/3 collection as the seed, with a multiplier of 50, to generate 30,200 artificial EPCs. We converted these EPCs into workflow systems and filtered out the unsound models, leading to 16,769 sound workflow systems. Finally, from these 16,769 systems, we randomly selected 1,000 systems, each with more than 10 nodes.

Table 7: Structural characteristics of the industrial process models.

	#P	**#T**	**#F**	**#OT**	**#XS**	**#XJ**	**#AS**	**#AJ**	**#PG**	**#B**	**#R**
Average	17.85	15.91	35.7	8.57	0.64	0.65	1.13	1.13	5.73	1.78	0.002
Minimum	4	3	6	2	0	0	0	0	1	0	0
Maximum	74	86	180	35	6	6	6	6	31	9	1
Std. Dev.	12.73	11.89	27.61	6.21	0.89	0.91	1.21	1.22	4.93	1.66	0.05

Tables 7 and 8 provide statistics on the number of places (#P), transitions (#T), flow arcs (#F), observable transitions (#OT), XOR-splits (#XS), XOR-joins (#XJ), AND-splits (#AS), AND-joins (#AJ), polygons (#PG), bonds (#B), and rigids (#R) in the models of the two collections. These are common characteristics for comparing the structural properties of model collections [71]; see Appendix F for details. The tables demonstrate that industrial and synthetic models have similar structural characteristics.

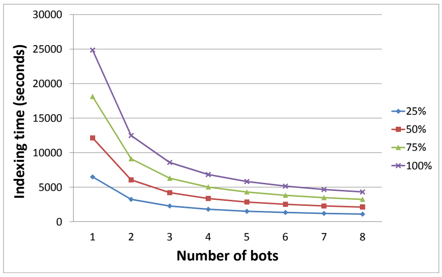
Table 8: Structural characteristics of the synthetic process models.

	#P	**#T**	**#F**	**#OT**	**#XS**	**#XJ**	**#AS**	**#AJ**	**#PG**	**#B**	**#R**
Average	16.63	13.52	32.4	10.95	0.45	0.43	1.26	1.22	6.192	1.609	0.116
Minimum	5	5	10	3	0	0	0	0	1	0	0
Maximum	86	84	190	67	8	5	7	6	39	8	2
Std. Dev.	11.14	9.01	23.08	7.48	0.85	0.77	1.04	0.98	4.99	1.15	0.34

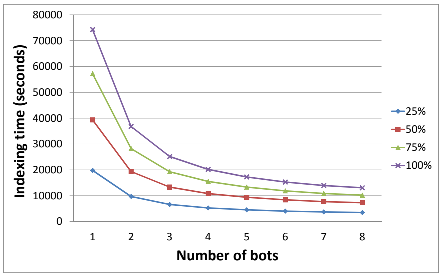
Queries. We designed 150 PQL query templates. Each template is a PQL query with placeholders for activity labels. During the experiments, the placeholders were instantiated with random labels. The query templates were developed to exploit the various features of the PQL grammar. According to the PQL features they support, the query templates were divided into three categories and further subdivided into groups and subgroups. The first category contains six query templates capturing individual atomic behavioral predicates. The second category contains 50 query templates that result from combining atomic predicates via logical operations. Finally, the third category contains 94 query templates that use predicate macros and construction of task sets using predicates with set operations and comparisons. Table 9 lists all the PQL query categories, groups, subgroups, and provides numbers of query templates accordingly (see column "# Templates"). For details on the query templates, refer to Appendix F, while Appendix G lists all the 150 PQL query templates.

Table 9: Categories, Groups, and Subgroups of PQL query templates.

Category.Group.Subgroup	Name	# Templates
1.a	Unary atomic predicates	2
1.b	Binary atomic predicates	4
2.a.1	Negations of predicates (Same)	6
2.a.2	Conjunctions of predicates (Same)	18
2.a.3	Disjunctions of predicates (Same)	18
2.b.1	Conjunctions of predicates (Mixed)	3
2.b.2	Disjunctions of predicates (Mixed)	3
2.b.3	Combinations of logical operations	2
3.a.1	Unary predicate macros	12
3.a.2	Binary predicate macros (Task-Set)	24
3.a.3	Binary predicate macros (Set-Set)	36
3.b.1	Constructions via unary predicates	2
3.b.2	Constructions via binary predicates	8
3.b.3	Constructions with set operations	5
3.b.4	Constructions with set comparisons	7



(a)



(b)

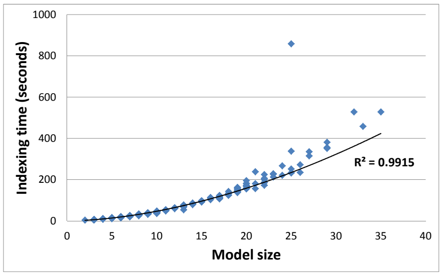
Fig. 9: Impact of PQL bots on indexing time: (a) industrial and (b) synthetic models.

7.2. *Indexing Performance*

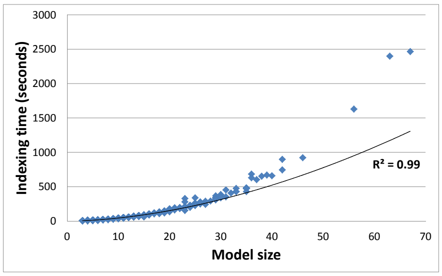
We conducted four experiments to measure the performance of PQL bots and the impact of different factors on indexing time. In what follows, for each of these four experiments, we detail its setup and discuss the obtained measurements.

Experiment 1.1: Impact of PQL bots on indexing time. The goal of this experiment was to measure the performance of PQL bots. We measured the time of indexing the industrial and synthetic models using different numbers of bots (from 1 to 8). Each indexing exercise was repeated three times, and we recorded the average indexing times of the three runs. In all the runs, the bots were configured to index the label similarity threshold of 1.0. The procedure was repeated for different parts of the process model collections, i.e., using 25%, 50%, 75%, and 100% of models in each collection. Models for each part of each process model collection were selected randomly.

Fig. 9 plots the indexing times (in seconds) for different parts of process model collections against different numbers of PQL bots for (a) the industrial and (b) synthetic models. The two plots demonstrate that adding bots decreases indexing time, though the decrease in indexing time gets less pronounced with



(a)



(b)

Fig. 10: Impact of model size on indexing time: (a) industrial and (b) synthetic models.

the increase in the number of bots. This experiment confirmed that the indexing time grows linearly with the size of a process model collection.

Experiment 1.2: Impact of model size on indexing time. This experiment aimed to assess the impact of the size of a model on its indexing time. The two model collections were indexed three times (using one bot), and for each model, we recorded the average indexing time of the three runs. The bot was configured to index the label similarity threshold of 1.0.

Fig. 10 plots the indexing times (in seconds) against different sizes of workflow systems for (a) the industrial and (b) synthetic models. In this experiment, the size of a workflow system is measured as the number of its observable transitions. The average indexing time of a model in the industrial collection is 50.3 seconds, with a minimum of 4.0 seconds (for a model with 2 observable transitions) and a maximum of 858.7 seconds (for a model with 25 observable transitions). We noticed that 95% of models in the industrial collection were indexed in less than 200 seconds. The average indexing time of a model in the synthetic collection is 74.0 seconds, with a minimum of 6.3 seconds (for a model with 3 observable transitions) and a maximum of 2,465.7 seconds (for a model with 67 observable transitions). More than 95% of models in the synthetic collection were indexed in less than 250 seconds. In general, the indexing times demonstrated polynomial dependency on the sizes of the indexed models.

Experiment 1.3: Impact of label similarity threshold on indexing time. This experiment aimed to assess the impact of different label similarity thresholds on the average time required to index a model. In this experiment, we varied the label similarity threshold (0.5, 0.6, 0.7, 0.8, 0.9, and 1.0) used to index the models. Both process model collections were indexed with one bot three times (for each similarity threshold), and average indexing times for the three runs were recorded. The Lucene-VSM label comparison method was used in all the runs. The average indexing times of an industrial model for the label similarity thresholds 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0 are 49.73, 50.3, 50.44, 50.48, 50.3, and 50.31 seconds, respectively.The measured average indexing times of a synthetic model for the label similarity thresholds 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0 are 74.53, 74.63, 74.62, 74.57, 74.61, and 74.48 seconds, respectively.

There is no strong relation between the label similarity thresholds and the average indexing times. We conclude that the impact of different label similarity thresholds on indexing times is negligible. This conclusion can be explained by the small number of unique labels in both collections (2,278 unique labels in the industrial models and 10,473 unique labels in the synthetic models) and the fact that the labels are short character strings (on average 4.84 and 6.08 words in a label of an industrial and a synthetic model, respectively), whereas modern information retrieval engines are known to be efficient on collections that comprise millions of natural language documents of much larger sizes [72].

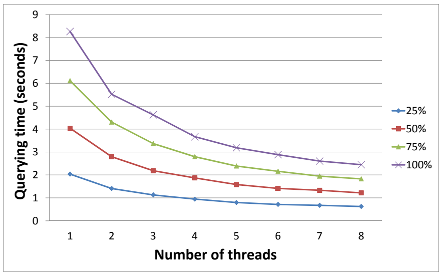
Experiment 1.4: Impact of index size on indexing time. This experiment aimed to measure the impact of index size on the average time required to index a model. To this end, we randomly split the industrial collection into four sets of (approximately) the same size (Sets 1–4). Each set was indexed four times: once when the index was empty, and then after 25%, 50%, and 75% of the collection was indexed. The index size does not significantly affect the average indexing time. We noticed a negligible increase in the indexing time with the growth of the index size. This observation can be explained by the fact that the introduced overhead is due to write operations on the PQL index, which modern database management systems can efficiently handle.

7.3. Querying Performance

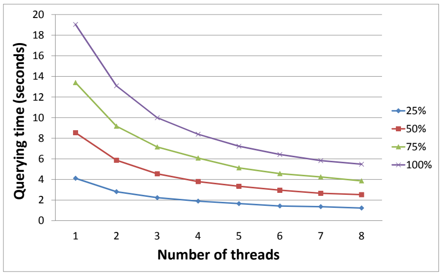
We conducted three experiments to assess the performance of executing different types of PQL queries. The queries were generated from the templates discussed in Section 7.1. To conduct the evaluation, for each model collection and template, we generated three queries by populating the template with labels randomly selected from all the labels in the collection.

Experiment 2.1: Impact of query threads on querying time. This experiment measured the impact of the number of query threads and the size of a model collection on the querying time. We varied the number of query threads (from 1 to 8) and executed all the generated queries on different parts of the collections (25%, 50%, 75%, and 100% of models in each collection). For each part of each collection, models were randomly selected three times, and the average querying times for the three runs were recorded. The models were indexed using the label similarity threshold of 1.0.

Fig. 11 plots the querying times (in seconds) for different parts of process model collections against different numbers of query threads for (a) the industrial and (b) synthetic models. It demonstrates that additional query threads decrease querying time. As in Experiment 1.1, the gain in performance gets less pronounced with the increase in the number of query threads. For example, querying the industrial models with one thread took on average 8.259 seconds, two threads managed to accomplish queries over these models in 6.109 seconds (1.35 times faster than using one thread), while eight threads used 2.037 seconds to execute a query over the whole collection (four times faster than with one thread). A similar trend was observed for the synthetic models. The relation between the querying times and the number of threads is best captured by power functions with negative exponents. Also, the collected measurements reveal that the querying time grows linearly with the size of a process model collection.



(a)



(b)

Fig. 11: Impact of threads on querying time: (a) industrial and (b) synthetic models.

Experiment 2.2: Impact of query types on querying time. This experiment aimed to assess the impact of different query types on querying time. It uses the same setup as Experiment 2.1. In this experiment, we measured querying times for different query groups discussed in Section 7.1.

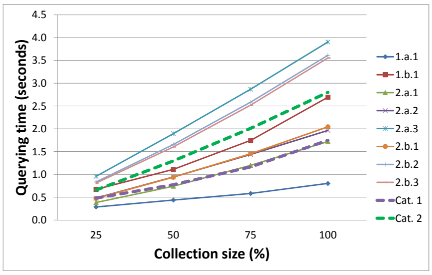
Fig. 12 plots the average querying times for different model collection sizes and different numbers of query threads. Figs. 12(a), 12(c), 12(e) and 12(g) show the linear dependency between the number of models in a collection and querying times for different query types, while Figs. 12(b), 12(d), 12(f), and 12(h) demonstrate a trend that is similar to the one observed in Experiment 2.1. The results confirm the feasibility of using PQL in industrial settings. With eight query threads, the Category 1 queries were, on average, accomplished in 0.47 seconds for the industrial collection and 1.61 seconds for the synthetic collection.

Experiment 2.3: Impact of label similarity on querying time. This experiment aimed to measure the impact of label similarity on querying time. We repeated Experiment 2.1 with the following modifications: (i) the bots were configured to index label similarity thresholds of 0.75 and 1.0, (ii) all the query templates were augmented to include the "tilde" symbol immediately before every activity label (thus, similar labels were considered during querying), and (iii) the tool was configured to use the default label similarity threshold of 0.75.

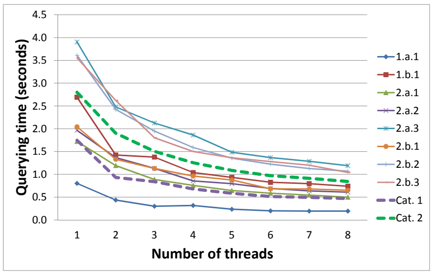
We compared the observed querying times with the querying times obtained in Experiment 2.1. For the industrial and synthetic models, the average querying times (using one thread) are, respectively, 10.9% and 9.47% higher for queries that account for similar labels. This small overhead can be explained by the fact that the PQL tool indexes behavioral relations at the level of PQL tasks and, thus, at runtime, the overhead is only due to the additional time required to retrieve information on the indexed PQL tasks.

8. Related Work

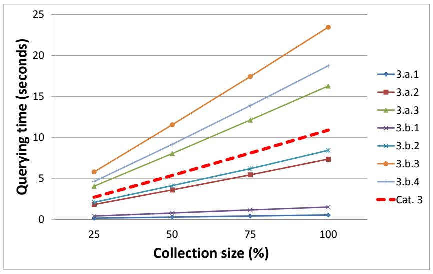
Recently, we conducted a systematic literature review of the state-of-the-art methods for querying process repositories [4]. As part of that study, we designed a framework for developing process querying methods. The framework is an abstract system in which components can be selectively replaced to result in a new process querying method. According to this framework, PQL addresses querying of *formal* process models using a query language with a *formal* semantics that



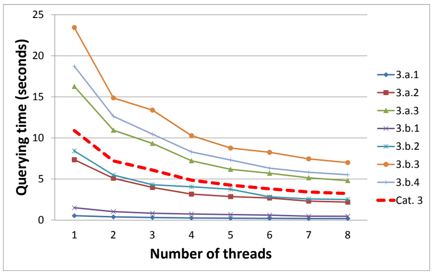
(a)



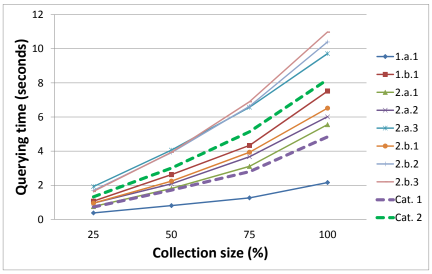
(b)



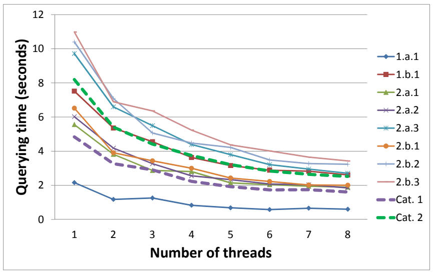
(c)



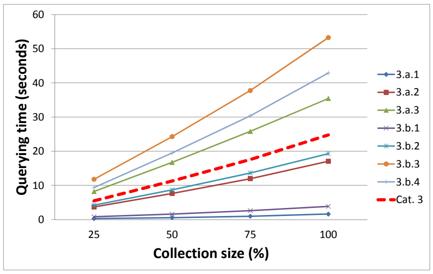
(d)



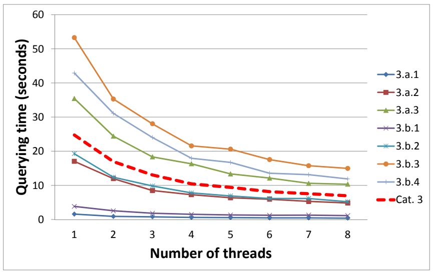
(e)



(f)



(g)



(h)

Fig. 12: The average querying times for different model collection sizes (a,c,e,g) and different numbers of query threads (b,d,f,h); PQL queries in Categories 1 and 2 for the industrial models (a,b) and synthetic models (e,f) and PQL queries in Category 3 for the industrial models (c,d) and synthetic models (g,h).

implements the *read* querying intent, i.e., is designed to retrieve models from repositories. Two recent surveys of process querying methods [12, 4] confirm that query languages with these characteristics constitute a gap in the area of process querying. In [73], we categorized and summarized the existing process querying methods. According to that classification, two prominent existing works for behavioral querying of process models that share similar characteristics with PQL are the works on "Behavior Query Language" (BQL) [74] and "A Process-model Query Language" (APQL) [28]. Next, we briefly review APQL and BQL, summarize some important results on behavioral predicates, and discuss several related techniques from the areas outside process querying.

Behavioral querying. PQL is inspired by our work on APQL and redesigns APQL in many respects. PQL differs from APQL in that it is grounded in a different and empirically justified collection of behavioral predicates, extends the abstract syntax of APQL, proposes a concrete syntax, has an implementation that demonstrates its feasibility, and operates on the level of observable behaviors, i.e., incorporates mechanisms for interpreting two different tasks as such that have the same meaning for the purpose of evaluating the queries. Compared to BQL, PQL is rigorous since both the syntax and semantics of the language are formally defined. BQL is grounded in three behavioral predicates. All these predicates belong to the 4C spectrum and were included in our empirical study. These three predicates are `Conflict` (called "exclude" in [74]), `ExistCausal` ("precede"), and `ExistConcurrent` ("parallel with"), refer to Section 4.1 for more details. As a result of our empirical evaluation, only one of these three predicates, namely `Conflict`, was selected to be included in the set of eight core PQL predicates. As to `ExistCausal` and `ExistConcurrent`, the stakeholders have given their preference to "stronger" predicates of `TotalCausal` and `TotalConcurrent`, which ensure that the respective behavioral relation holds for all (rather than only for some) occurrences of the tasks. Finally, the fundamental difference between PQL and BQL is that BQL predicates are defined over occurrences of tasks, while PQL predicates are defined over aggregations (derived using quantification) over the occurrences of tasks.

Model Checking. Model checking studies problems that can verify properties of process models [75, 76]. A model checking problem is a problem that, given a formal specification of a property, usually captured using some *specification language*, and a process model, answers whether the property holds in the model. To solve a problem, model checking often proceeds by constructing an alternative representation of the model that indicates whether the given property holds or not in the model. Model checking techniques usually use *temporal logics* as property specification languages, e.g., linear temporal logic (LTL) and computational tree logic (CTL). Model checking techniques can be employed for process querying to retrieve process models that fulfill a given property [77].

As recently demonstrated by Wolf [78], computations of most of the 4C behavioral relations can be reduced, via non-trivial transformations that require exponential space, to classical interleaving-based model checking problems. Only one problem remains completely unsolved, whereas several problems were solved in the absence of auto-concurrency. In fact, based on the results reported in [20, 78], we know that all the predicates selected for inclusion into PQL, refer to Section 4.3, can be reduced to model checking problems. However, one cannot directly apply the proposed solutions for process querying. Note that the proposed in [78] LTL and CTL properties for computing the 4C relations are formulated over the transformed models and, hence, do not convey the meanings of the properties to be computed, which makes them unsuitable for user interpretations. Also, the performance of the approach proposed in [78] has not been evaluated; note that model checking on infinite-state systems is undecidable and is PSPACE-complete on finite-state systems [18]. Such an evaluation, and development of new efficient techniques for computing the 4C relations, may contribute to the development of PQL.

Behavioral predicates. Dwyer et al. [79] report the results of a survey of property specifications (a.k.a. behavioral predicates) captured in *temporal logics*, e.g., LTL or CTL. The authors collected and classified 555 properties, most specified in LTL, from various domains, including hardware protocols, communication protocols, avionics, operating systems, and database systems. Interestingly, the authors conclude that "even with significant expertise, dealing with the complexity of such a specification [a temporal logic property] can be daunting" and suggest that often "complexity is addressed by the definition and use of abstraction". PQL implements such an abstraction. There are no translations from PQL predicates to temporal logic properties over concepts of the original process model; see above. However, it is easy to see that several properties surveyed by Dwyer et al. [79] can be expressed as logical expressions over the 4C relations. A comprehensive study of such translations is future work.

To the best of our knowledge, no existing works study the relevance of behavioral predicates for querying (business) process models. Future empirical studies will aim at gaining a better understanding of the suitability of behavioral predicates for process querying, as per the process querying compromise between decidable, efficiently computable, and suitable process querying methods [4].

Verification of software systems. In [80], the authors proposed FLAVERS—a finite-state verification technique to analyze whether a concurrent software system satisfies a given user-specified property. To perform the verification, FLAVERS constructs an abstract representation of the system. This abstraction step comes at the price of precision of the analysis results. In our work, we suggest using PQL to query repositories of business process models. However, one can explore PQL for querying, testing, and verifying software systems. Unlike FLAVERS, PQL is precise, i.e., the result of every PQL query is free from false positive and false negative errors. FLAVERS and PQL differ in how they interpret models of analyzed models, according to the interleaving and noninterleaving semantics of concurrent systems [81], respectively. Thus, PQL can be used to express properties that address the potential simultaneous execution of instructions of software systems.

In software engineering, well-established finite-state verification techniques, like the methods based on classical model checking or the FLAVERS technique, are used to inform the continuous improvement of processes [82, 83]. PQL can enrich this repertoire of verification techniques for detecting errors in semantically rich process models with user-interpretable and relevant properties.

Declarative process discovery. Process mining helps business analysts in dealing with the complexities and uncertainties introduced by business processes. It aims to discover, monitor, and improve processes observed in the real world using the knowledge accumulated in event logs recorded by information systems [84]. An event log is a collection of traces, each comprising events executed in a business process. Process discovery involves obtaining a good process model that describes the behavior recorded in an event log. Declarative process discovery aims to construct such process models as collections of declarative constraints over possible executions of business activities. These declarative constraints are often expressed as behavioral predicates, similar to those used in PQL.

MINERful is a declarative process discovery algorithm [85]. The algorithm performs the statistical analysis of the input event log and then uses the derived knowledge to compose the declarative constraints, which collectively describe the traces recorded in the event log. In particular, MINERful discovers Declare constraints [86], a repertoire of LTL templates. In [87], the authors propose another algorithm capable of discovering Declare constraints from event logs called UnconstrainedMiner. In that work, Declare templates are translated to regular expressions to address the problem of LTL semantics over finite traces.

Declarative process discovery algorithms can target the discovery of the 4C constraints, particularly the PQL predicates. This will allow overcoming two limitations of the existing techniques: the inability to encode noninterleaving semantics [81] and the lack of empirical justification of the discovered constraints.

9. Conclusion

This article presents a query language, called Process Query Language (PQL), for retrieving process models. The language is grounded in empirically justified behavioral predicates that can be used to check whether a model describes executions of interest. To facilitate adoption, the language has an SQL-like concrete syntax. PQL supports exploratory querying by searching models that describe executions with activity labels similar to those specified in PQL queries. The language has been implemented, and its runtime performance has been evaluated using real-life and synthetic process model collections. The experiments confirm the feasibility of computing PQL queries in close to real-time.

We acknowledge several limitations of this work that give rise to future work. First, the expressiveness of PQL is limited by the basic predicates it supports, a fundamental limitation of every query language based on behavioral predicates [16]. Expressiveness can be increased by using execution templates with wildcards [29], such as to express a search intent like “find all process models that allow execution $\langle \mathtt{a}, \mathtt{b}, *, \mathtt{b}, \mathtt{b}, *, \mathtt{c}, * \rangle$” with wildcards representing any sequence of tasks. Second, the design of PQL should be supported by further empirical evidence. Such evidence includes exploring relations between desired queries and PQL capabilities, evaluating how users specify complex queries, and studying the usefulness of the exploratory search capabilities of the language. Third, PQL considers control-flow in process models and not other perspectives, like data-flow and resources. To broaden applicability, future versions of PQL can support these perspectives. Fourth, PQL can be extended with functionality that is auxiliary to behavioral querying, such as aggregate functions over retrieved processes. PQL predicates operate over the global process scope, i.e., over processes that represent completed model executions. Similar to some properties surveyed by Dwyer et al. [79], PQL predicates can be generalized to operate over other scopes, e.g., between given process conditions. Fifth, PQL can be extended with manipulation statements, e.g., `INSERT`, `DELETE`, `UPDATE`, to allow adding, deleting, and modifying executions supported by process models.

Another limitation of PQL is that queried process models must come from the class of sound workflow systems, but in practice, process models may not be sound. It is interesting to expand the applicability of PQL to unsound models. Finally, despite the benefits of behavior-based querying, behavior conditions that trigger a model match to a query may be complex. Different techniques for explaining query results can be explored to address this challenge.

Acknowledgements. We thank Jan Recker for his contribution to the design of the experiment for evaluating the practical relevance of using behavioral predicates for querying process models.

References

[1] C. Ouyang, M. Dumas, W. M. P. van der Aalst, A. H. M. ter Hofstede, J. Mendling, From business process models to process-oriented software systems, ACM Trans. Softw. Eng. Methodol. 19 (1) (2009) 2:1–2:37.

[2] M. Weske, Business Process Management: Concepts, Languages, Architectures, 3rd Edition, Springer, 2019.

[3] M. Dumas, M. La Rosa, J. Mendling, H. A. Reijers, Fundamentals of Business Process Management, 2nd Edition, Springer, 2018.

[4] A. Polyvyanyy, C. Ouyang, A. Barros, W. M. P. van der Aalst, Process querying: Enabling business intelligence through query-based process analytics, Decis. Support Syst. 100 (2017) 41–56.

[5] T. Curran, G. Keller, A. Ladd, SAP R/3 Business Blueprint: Understanding the Business Process Reference Model, Prentice-Hall, Inc., 1998.

[6] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, K. Wolf, Analysis on demand: Instantaneous soundness checking of industrial business process models, Data Knowl. Eng. 70 (5) (2011) 448–466.

[7] A. Polyvyanyy, S. Smirnov, M. Weske, Reducing complexity of large EPCs, in: MobIS, Vol. P-141 of LNI, GI, 2008, pp. 195–207.

[8] M. La Rosa, M. Dumas, R. Uba, R. M. Dijkman, Business process model merging: An approach to business process consolidation, ACM Trans. Softw. Eng. Methodol. 22 (2) (2013) 11:1–11:42.

[9] A. Polyvyanyy (Ed.), Process Querying Methods, Springer, 2022.

[10] C. Beeri, A. Eyal, S. Kamenkovich, T. Milo, Querying business processes with BP-QL, in: VLDB, ACM, 2005, pp. 1255–1258.

[11] A. Awad, BPMN-Q: A language to query business processes, in: EMISA, Vol. P-119 of LNI, GI, 2007, pp. 115–128.

[12] J. Wang, T. Jin, R. K. Wong, L. Wen, Querying business process model repositories: A survey of current approaches and issues, World Wide Web 17 (3) (2014) 427–454.

[13] P. Barceló, L. Libkin, J. L. Reutter, Querying regular graph patterns, J. ACM 61 (1) (2014) 8:1–8:54.

[14] L. Libkin, W. Martens, D. Vrgoc, Querying graphs with data, J. ACM 63 (2) (2016) 14:1–14:53.

[15] A. Polyvyanyy, Structuring process models, Ph.D. thesis, University of Potsdam (2012).

[16] A. Polyvyanyy, A. Armas-Cervantes, M. Dumas, L. García-Bañuelos, On the expressive power of behavioral profiles, Formal Aspects Comput. 28 (4) (2016) 597–613.

[17] A. Scheer, O. Thomas, O. Adam, Process modeling using event-driven process chains, in: Process-Aware Information Systems, Wiley, 2005, pp. 119–145.

[18] J. Esparza, M. Nielsen, Decidability issues for Petri nets: A survey, Bull. EATCS 52 (1994) 244–262.

[19] J. Esparza, K. Heljanko, Unfoldings: A Partial-Order Approach to Model Checking, Monographs in Theoretical Computer Science. An EATCS Series, Springer, 2008.

[20] A. Polyvyanyy, M. Weidlich, R. Conforti, M. La Rosa, A. H. M. ter Hofstede, The 4C spectrum of fundamental behavioral relations for concurrent systems, in: Petri Nets, Vol. 8489 of LNCS, Springer, 2014, pp. 210–232.

[21] A. Polyvyanyy, Introduction to process querying, in: Process Querying Methods, Springer, 2022, pp. 1–18.

[22] M. Hack, Decidability Questions for Petri Nets, Outstanding Dissertations in the Computer Sciences, Garland Publishing, New York, 1975.

[23] C. Rackoff, The covering and boundedness problems for vector addition systems, Theor. Comput. Sci. 6 (1978) 223–231.

[24] K. L. McMillan, Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits, in: Computer Aided Verification, Vol. 663 of LNCS, Springer, 1992, pp. 164–177.

[25] J. Esparza, S. Römer, W. Vogler, An improvement of McMillan's unfolding algorithm, Formal Methods Syst. Des. 20 (3) (2002) 285–310.

[26] M. Nielsen, G. D. Plotkin, G. Winskel, Petri nets, event structures and domains, part I, Theor. Comput. Sci. 13 (1981) 85–108.

[27] R. Lipton, The Reachability Problem Requires Exponential Space, Yale University, Department of Computer Science, Research Report 62, New Haven, Connecticut, 1976.

[28] A. H. M. ter Hofstede, C. Ouyang, M. La Rosa, L. Song, J. Wang, A. Polyvyanyy, APQL: A process-model query language, in: AP-BPM, Vol. 159 of LNBIP, Springer, 2013, pp. 23–38.

[29] A. Polyvyanyy, A. Pika, A. H. M. ter Hofstede, Scenario-based process querying for compliance, reuse, and standardization, Inf. Syst. 93 (2020) 101563.

[30] W. M. P. van der Aalst, Formalization and verification of event-driven process chains, Inf. Softw. Technol. 41 (10) (1999) 639–650.

[31] N. Lohmann, E. Verbeek, C. Ouyang, C. Stahl, Comparing and evaluating Petri net semantics for BPEL, Int. J. Bus. Process. Integr. Manag. 4 (1) (2009) 60–73.

[32] H. M. W. Verbeek, W. M. P. van der Aalst, A. H. M. ter Hofstede, Verifying workflows with cancellation regions and OR-joins: An approach based on relaxed soundness and invariants, Comput. J. 50 (3) (2007) 294–314.

[33] R. M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, Inf. Softw. Technol. 50 (12) (2008) 1281–1294.

[34] W. Reisig, Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies, Springer, 2013.

[35] W. M. P. van der Aalst, Verification of workflow nets, in: ICATPN, Vol. 1248 of LNCS, Springer, 1997, pp. 407–426.

[36] B. F. van Dongen, M. H. Jansen-Vullers, H. M. W. Verbeek, W. M. P. van der Aalst, Verification of the SAP reference models using EPC reduction, state-space analysis, and invariants, Comput. Ind. 58 (6) (2007) 578–601.

[37] B. Kiepuszewski, A. H. M. ter Hofstede, W. M. P. van der Aalst, Fundamentals of control flow in workflows, Acta Informatica 39 (3) (2003) 143–209.

[38] A. Polyvyanyy, L. García-Bañuelos, M. Dumas, Structuring acyclic process models, Inf. Syst. 37 (6) (2012) 518–538.

[39] U. Goltz, W. Reisig, The non-sequential behavior of Petri nets, Inf. Control. 57 (2/3) (1983) 125–147.

[40] A. Polyvyanyy, M. La Rosa, C. Ouyang, A. H. M. ter Hofstede, Untanglings: A novel approach to analyzing concurrent systems, Formal Aspects Comput. 27 (5-6) (2015) 753–788.

[41] J. Engelfriet, Branching processes of Petri nets, Acta Informatica 28 (6) (1991) 575–591.

[42] S. Haar, C. Kern, S. Schwoon, Computing the reveals relation in occurrence nets, Theor. Comput. Sci. 493 (2013) 66–79.

[43] P. Baldan, S. Crafa, A logic for true concurrency, J. ACM 61 (4) (2014) 24:1–24:36.

[44] L. van Maanen, H. van Rijn, J. P. Borst, Stroop and picture—word interference are two sides of the same coin, Psychonomic Bulletin & Review 16 (6) (2009) 987–999.

[45] Y. Wand, R. Y. Wang, Anchoring data quality dimensions in ontological foundations, Commun. ACM 39 (11) (1996) 86–95.

[46] F. D. Davis, Perceived usefulness, perceived ease of use, and user acceptance of information technology, MIS Q. 13 (3) (1989) 319–340.

[47] D. F. Larcker, V. P. Lessig, Perceived usefulness of information: A psychometric examination, Decision Sciences 11 (1) 121–134.

[48] J. Mendling, M. Strembeck, J. Recker, Factors of process model comprehension: Findings from a series of experiments, Decis. Support Syst. 53 (1) (2012) 195–206.

[49] G. W. Ryan, H. R. Bernard, Techniques to identify themes, Field Methods 15 (1) (2003) 85–109.

[50] P. Sprent, Sign Test, Springer Berlin Heidelberg, 2011, pp. 1316–1317.

[51] F. Faul, E. Erdfelder, A. Buchner, A.-G. Lang, Statistical power analyses using G*Power 3.1: Tests for correlation and regression analyses, Behavior Research Methods 41 (4) (2009) 1149–1160.

[52] C. A. R. Hoare, Hints on programming language design, Tech. rep. (1973).

[53] K. C. Louden, Programming Languages: Principles and Practices, Advanced Topics Series, Cengage Learning, 2011.

[54] N. Lohmann, E. Verbeek, R. M. Dijkman, Petri net transformations for business processes: A survey, Trans. Petri Nets Other Model. Concurr. 2 (2009) 46–63.

[55] R. W. White, R. A. Roth, Exploratory Search: Beyond the Query-Response Paradigm, Synthesis Lectures on Information Concepts, Retrieval, and Services, Morgan & Claypool Publishers, 2009.

[56] H. Leopold, Natural Language in Business Process Models: Theoretical Foundations, Techniques, and Applications, Vol. 168 of LNBIP, Springer, 2013.

[57] C. D. Manning, P. Raghavan, H. Schütze, Introduction to information retrieval, Cambridge University Press, 2008.

[58] A. Awad, A. Polyvyanyy, M. Weske, Semantic querying of business process models, in: EDOC, IEEE Computer Society, 2008, pp. 85–94.

[59] URI Planning Interest Group, URIs, URLs, and URNs: Clarifications and recommendations 1.0, Tech. rep., W3C (2001).

[60] W3C XSL/XML Query Working Groups, The XPath 2.0 standard (2007).

[61] C. Date, H. Darwen, A Guide to the SQL Standard: A User's Guide to the Standard Database Language SQL, 4th Edition, Addison-Wesley, 1996.

[62] B. Meyer, Introduction to the Theory of Programming Languages, Prentice-Hall, 1990.

[63] A. Polyvyanyy, J. Vanhatalo, H. Völzer, Simplified computation and generalization of the refined process structure tree, in: WS-FM, Vol. 6551 of LNCS, Springer, 2010, pp. 25–41.

[64] T. J. Parr, The Definitive ANTLR 4 Reference, Oreilly and Associate Series, Pragmatic Programmers, LLC, 2013.

[65] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, 3rd Edition, Addison-Wesley Longman Publishing Co., Inc., 2006.

[66] A. Polyvyanyy, Evaluation of a novel information retrieval model: eTVSM, Master's thesis, University of Potsdam (2007).

[67] Y. Li, B. Liu, A normalized Levenshtein distance metric, IEEE Trans. Pattern Anal. Mach. Intell. 29 (6) (2007) 1091–1095.

[68] J. Billington, S. Christensen, K. M. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, M. Weber, The Petri net markup language: Concepts, technology, and tools, in: ICATPN, Vol. 2679 of LNCS, Springer, 2003, pp. 483–505.

[69] K. Schmidt, LoLA: A low level analyser, in: ICATPN, Vol. 1825 of LNCS, Springer, 2000, pp. 465–474.

[70] A. Polyvyanyy, M. Weidlich, Towards a compendium of process technologies: The jBPT library for process model analysis, in: CAiSE Forum, Vol. 998 of CEUR Workshop Proceedings, CEUR-WS.org, 2013, pp. 106–113.

[71] Z. Yan, R. M. Dijkman, P. Grefen, Generating process model collections, Softw. Syst. Model. 16 (4) (2017) 979–995.

[72] R. Baeza-Yates, B. A. Ribeiro-Neto, Modern Information Retrieval: The concepts and technology behind search, 2nd Edition, Pearson Education Ltd., Harlow, England, 2011.

[73] A. Polyvyanyy, Business Process Querying, Springer, 2019.

[74] T. Jin, J. Wang, L. Wen, Querying business process models based on semantics, in: DASFAA, Springer, 2011, pp. 164–178.

[75] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, J. D. Reese, Model checking large software specifications, IEEE Trans. Software Eng. 24 (7) (1998) 498–520.

[76] C. Baier, J. Katoen, Principles of Model Checking, MIT Press, 2008.

[77] A. Gurfinkel, M. Chechik, B. Devereux, Temporal logic query checking: A tool for model exploration, IEEE Trans. Software Eng. 29 (10) (2003) 898–914.

[78] K. Wolf, Interleaving based model checking of concurrency and causality, Fundam. Informaticae 161 (4) (2018) 423–445.

[79] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, Patterns in property specifications for finite-state verification, in: ICSE, ACM, 1999, pp. 411–420.

[80] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, G. Naumovich, Flow analysis for verifying properties of concurrent software systems, ACM Trans. Softw. Eng. Methodol. 13 (4) (2004) 359–430.

[81] V. Sassone, M. Nielsen, G. Winskel, Models for concurrency: Towards a classification, Theor. Comput. Sci. 170 (1-2) (1996) 297–348.

[82] L. A. Clarke, G. S. Avrunin, L. J. Osterweil, Using software engineering technology to improve the quality of medical processes, in: ICSE Companion, ACM, 2008, pp. 889–898.

[83] L. J. Osterweil, M. Bishop, H. M. Conboy, H. Phan, B. I. Simidchieva, G. S. Avrunin, L. A. Clarke, S. Peisert, Iterative analysis to improve key properties of critical human-intensive processes: An election security example, ACM Trans. Priv. Secur. 20 (2) (2017) 5:1–5:31.

[84] W. M. P. van der Aalst, Process Mining: Data Science in Action, 2nd Edition, Springer Berlin Heidelberg, 2016.

[85] C. D. Ciccio, M. Mecella, On the discovery of declarative control flows for artful processes, ACM Trans. Manag. Inf. Syst. 5 (4) (2015) 24:1–24:37.

[86] W. M. P. van der Aalst, M. Pesic, H. Schonenberg, Declarative workflows: Balancing between flexibility and support, Comput. Sci. Res. Dev. 23 (2) (2009) 99–113.

[87] M. Westergaard, C. Stahl, H. A. Reijers, UnconstrainedMiner: Efficient discovery of generalized declarative process models, Tech. Rep. BPM reports; Vol. 1328; BPMcenter.org (2013).

[88] T. J. Parr, R. W. Quong, ANTLR: A predicated-*ll(k)* parser generator, Softw. Pract. Exp. 25 (7) (1995) 789–810.

[89] M. Weidlich, A. Polyvyanyy, J. Mendling, M. Weske, Causal behavioural profiles: Efficient computation, applications, and evaluation, Fundam. Informaticae 113 (3-4) (2011) 399–435.

Appendix A. Interview Participants

Table A.10 lists profiles, including information on the roles, years of experience, number of process models managed, and the domains of the managed models, of the 23 participants of the interviews to inform the design of the PQL language. The participants are listed in the chronological order of the conducted interviews.

Table A.10: Profiles of the interview participants.

No.	Role	Experience (years)	Models managed	Models domains
1	Business analysts	NA	420	Motor insurance and home insurance
2	Business excellence manager; BSc in Economics and Accounting	1.5	50	Banking, IT, and legal
3	Senior business excellence manager	20	100	Banking and insurance
4	Business analyst and consultant	15	over 30000	HR, finance, and procurement
5	BPM&SOA consultant; BSc in Engineering	7	NA	Insurance, healthcare, media, investment banking, and telecom
6	BPM software product manager and business analyst; BSc in Informatics	15	many	Product management
7	Business process analyst; BSc in IT and MSc ni BPM	0.5	40-50	Commercial insurance
8	Process control engineer, management consultant, and business owner; MSc in P.Eng.	over 40	NA	Workflow management and healthcare
9	Business architect, enterprise architect, and business process expert	1	100	Investment and infrastructure development
10	Business consultant and technology advisor	over 10	NA	NA
11	Business analyst, business architect, BPM coach, modeling trainer, and business designer	over 20	hundreds	NA
12	Management consultant; MSc Mechanical Engineering and PhD in real-time production planning	21	NA	Shipping, container terminals, postal services, manufacturing, finance, HR, and IT
13	Business and process architect; MSc of IT	10	300-500	Energy
14	CEO and process mining consulting	NA	many	NA
15	World bank business analyst; MBA	10	over 100	Mining, sales, procurement, HR, and logistics
16	BPM consultant; MSc in Computer Science	30	NA	Manufacturing, banking, healthcare, and insurance
17	BPM consultant, BPM Lead, and process architect; MSc in Professional Accounting, MSc in IT, and MBA in e-Commerce	over 7	NA	NA
18	Senior consultant; BSc in Business and Marketing	3	1000	NA
19	Senior BPM consultant; BSc in Economics	11	over 1000	Banking, mining, and government
20	Solutions architect	over 12	many	Sales and production planning, stock management, procurement, and investment management
21	Business process technology consultant	over 7	73	Risk assessment, trading, HR, information management, safety, environment, and maintainance
22	Senior consultant; PhD in Computer Science	0.5	6	Processing and provision of data in a backend service
23	Consultant	15	NA	NA

Appendix B. Interview Quotes

This appendix lists interesting quotes from the participants of the conducted interviews on the design of the PQL language. First, we list quotes that relate to the usefulness and importance of the PQL language.

- "*Yes, definitely, it would be a good idea to be able to analyze our process repository in a way like this [process querying].*" (business analyst).
- "*If you're trying to look for something that you can improve on, having these queries and trying to find processes would help, so rather than businesses coming to you, you can be a bit more proactive.*" (business excellence manager).
- "*That [process querying] would be extremely useful. It would save enormous amounts of time. Actually, it would enable us to undertake an analysis that we can't do at the moment. So, it would open more doors for us to actually be able to do different, potentially more valuable things.*" (senior business excellence manager).
- "*This [process querying] can be useful from a governance perspective, i.e., to be able to check process controls are in place. This also can assist with risks associated with the process.*" (business analyst).
- "*I've been looking for years for some solutions in this field [process querying] ... it's for me very important and it is extremely useful to get this information*" (BPM software product manager and business analyst).
- "*I think from an overall strategic level it'll bring a lot of benefits because different parts of the organization operate in different ways and being able to actually analyze it through kind of a structured query language could be useful for an analyst. I can see it being quite highly useful because repositories right now are static and searching through can be time-consuming.*" (business analyst).

Next, we present quotes that relate to the SQL-like syntax of the PQL language.

- "*SQL-like query language would be good ... people will adopt it if they can relate it so something familiar, like SQL.*" (BPM consultant).
- "*SQL-like language is a science with which you can pose precise questions.*" (management consultant).
- "*I think from an overall strategic level it'll bring a lot of benefits because different parts of the organization operate in different ways and being able to actually analyze it [process repository] through kind of a structured query language could be useful for an analyst.*" (business analyst).

Finally, we list quotes that relate to the exploratory search capabilities of PQL.

- "*It should be possible to specify activities where tasks similar to 'Apply discount' can occur because having the exact string gets very difficult ... someone thinks 'apply discount' ... someone calls it 'discount the thing' or whatever. Is it something you also consider? I think this is something that makes such a query language quite powerful*" (CEO of a company working in BPM and process mining consultancy)

- “*Sometimes the customers’ language is different, but they mean the same thing . . . that [support for label similarities] would be something helpful that I would definitely use.*” (senior consultant)
- “*I think that [support for label similarities] will be interesting because people look for something at different angles to see the same thing.*” (business analyst)

Appendix C. PQL Grammar

This appendix specifies the complete grammar of PQL in the ANTLR notation. ANTLR (ANother Tool for Language Recognition) is a parser generator for reading and translating structured text or binary files [88, 64]. ANTLR can take a grammar of a language as input and generate source code for a parser that can build and walk syntax trees [62]. The language must be specified using a context-free grammar expressed using extended Backus-Naur Form [65].

```
// PQL version 1.0 grammar for ANTLR v4
// [The "BSD licence"]
// Copyright (c) 2014-2023 Artem Polyvyanyy
// All rights reserved.

grammar PQL;

query      : variables
             SELECT attributes
             FROM locations
             (WHERE predicate)? EOS ;

variables  : variable* ;
variable   : varName ASSIGN
               setOfTasks EOS ;

varName    : VARIABLE_NAME ;

attributes : attribute (SEP attribute)* ;
attribute  : universe
           | attributeName ;

locations  : location (SEP location)* ;
location   : universe
           | locationPath ;

universe          : UNIVERSE ;
attributeName     : STRING ;
locationPath      : STRING ;

setOfTasks : tasks
           | union
           | intersection
           | difference ;

tasks      : varName
           | setOfAllTasks
           | setOfTasksLiteral
           | setOfTasksConstruction
           | setOfTasksParentheses ;

setOfAllTasks     :
             GET_TASKS LP RP;

setOfTasksLiteral  :
           LB (task (SEP task)*)? RB ;

task       : approximate label
           | label (LSB similarity RSB)? ;

approximate: TILDE ;
label      : STRING ;
similarity : SIMILARITY ;

setOfTasksConstruction     :
             unaryPredicateConstruction
           | binaryPredicateConstruction ;

unaryPredicateConstruction  :
             (GET_TASKS)unaryPredicateName
             LP setOfTasks RP ;

binaryPredicateConstruction :
             (GET_TASKS)binaryPredicateName
             LP setOfTasks SEP setOfTasks
             SEP anyAll RP ;

anyAll     : ANY | ALL ;

unaryPredicateName : CAN_OCCUR
           | ALWAYS_OCCURS;

binaryPredicateName: CAN_CONFLICT
           | CAN_COOCCUR
           | CONFLICT
           | COOCCUR
           | TOTAL_CAUSAL
           | TOTAL_CONCUR ;

predicate  : proposition
           | conjunction
           | disjunction
           | logicalTest ;

proposition: unaryPredicate
           | binaryPredicate
           | unaryPredicateMacro
           | binaryPredicateMacro
           | setPredicate
           | truthValue
           | parentheses
           | negation ;

unaryPredicate      : unaryPredicateName
             LP task RP ;

binaryPredicate     : binaryPredicateName
             LP task SEP task RP ;

unaryPredicateMacro : unaryPredicateName
             LP setOfTasks SEP anyAll RP ;

binaryPredicateMacro:
             binaryPredicateMacroTaskSet
           | binaryPredicateMacroSetSet ;

binaryPredicateMacroTaskSet :
             binaryPredicateName LP task
             SEP setOfTasks SEP anyAll RP ;

binaryPredicateMacroSetSet  :
           binaryPredicateName
           LP setOfTasks SEP setOfTasks
           SEP anyEachAll RP ;

anySomeEachAll  : ANY | SOME | EACH | ALL ;

setPredicate: taskInSetOfTasks
           | setComparison ;

taskInSetOfTasks : task IN setOfTasks ;

setComparison    : setOfTasks
           setComparisonOperator
           setOfTasks ;

setComparisonOperator : identical
           | different
           | overlapsWith
           | subsetOf
           | properSubsetOf ;

truthValue : TRUE
           | FALSE ;

logicalTest: isTrue
           | isNotTrue
           | isFalse
           | isNotFalse ;

union      : (tasks | difference |
             intersection) UNION (tasks |
             difference | intersection)
             (UNION (tasks | difference
             | intersection))* ;

intersection : (tasks | difference)
               INTERSECTION
               (tasks | difference)
               (INTERSECTION (tasks
               | difference))* ;

difference   : tasks DIFFERENCE tasks
             | tasks DIFFERENCE
               difference ;

negation     : NOT proposition ;

isTrue       : proposition IS TRUE ;
isNotTrue    : proposition IS NOT TRUE ;
isFalse      : proposition IS FALSE ;
isNotFalse   : proposition IS NOT FALSE ;

disjunction  : (proposition | logicalTest |
         conjunction) OR (proposition |
         logicalTest | conjunction) (OR
         (proposition | logicalTest
         | conjunction))* ;

conjunction  : (proposition | logicalTest)
         AND (proposition | logicalTest)
         (AND (proposition
         | logicalTest))* ;

parentheses  : LP proposition RP
        | LP conjunction RP
        | LP disjunction RP
        | LP logicalTest RP ;

setOfTasksParentheses : LP varName RP
        | LP universe RP
        | LP setOfTasksLiteral RP
        | LP setOfTasksConstruction RP
        | LP union RP
        | LP difference RP
        | LP intersection RP
        | LP setOfTasksParentheses RP ;

UNIVERSE     : ’*’ ;

STRING       : DQ ( ESC_SEQ
             | ~(’\\’|’"’) )* DQ ;
VARIABLE_NAME: (’a’..’z’|’_’)
               (’a’..’z’|’0’..’9’|’_’)*;
SIMILARITY   : ’1’ | ’0’ (’.’ ’0’..’9’+)?
             | ’.’ ’0’..’9’+ ;

LP          : ’(’ ;
RP          : ’)’ ;
LB          : ’{’ ;
RB          : ’}’ ;
LSB         : ’[’ ;
RSB         : ’]’ ;
DQ          : ’"’ ;
EOS         : ’;’ ;
SEP         : ’,’ ;
ASSIGN      : ’=’ ;
TILDE       : ’~’ ;

ESC_SEQ     : ’\\’ (’\"’|’\\’|’/’|’b’|
            ’f’|’n’|’r’|’t’)
            | UNICODE_ESC ;
UNICODE_ESC : ’\\’ ’u’ HEX_DIGIT
            HEX_DIGIT HEX_DIGIT HEX_DIGIT ;
HEX_DIGIT   : (’0’..’9’|
            ’a’..’f’|’A’..’F’) ;
WS          : [ \r\t\n]+ -> skip ;
LINE_COMMENT: ’--’ ~[\r\n]* -> skip ;

SELECT      : ’SELECT’ ;
FROM        : ’FROM’ ;
WHERE       : ’WHERE’ ;

EQUALS      : ’EQUALS’ ;
OVERLAPS    : ’OVERLAPS’ ;
WITH        : ’WITH’ ;
```

```
SUBSET       : 'SUBSET' ;
PROPER       : 'PROPER' ;
GET_TASKS    : 'GetTasks' ;

NOT          : 'NOT' ;
AND          : 'AND' ;
OR           : 'OR' ;

ANY          : 'ANY' ;
SOME         : 'SOME' ;
EACH         : 'EACH' ;
ALL          : 'ALL' ;

IN           : 'IN' ;
IS           : 'IS' ;
OF           : 'OF' ;

TRUE         : 'TRUE' ;
FALSE        : 'FALSE' ;

identical        : EQUALS ;
different        : NOT EQUALS ;
overlapsWith     : OVERLAPS WITH ;
subsetOf         : IS SUBSET OF ;
properSubsetOf   : IS PROPER SUBSET OF ;

UNION            : 'UNION' ;
INTERSECTION     : 'INTERSECT' ;
DIFFERENCE       : 'EXCEPT' ;

CAN_OCCUR        : 'CanOccur' ;
ALWAYS_OCCURS    : 'AlwaysOccurs' ;
CAN_CONFLICT     : 'CanConflict' ;
CAN_COOCCUR      : 'CanCooccur' ;
CONFLICT         : 'Conflict' ;
COOCCUR          : 'Cooccur' ;
TOTAL_CAUSAL     : 'TotalCausal' ;
TOTAL_CONCUR     : 'TotalConcurrent' ;
```

Appendix D. Predicate Denotations

This appendix lists mathematical denotations of four alternatives of specifying the `Predicate` construct.

Every specimen of `SetComparison` denotes a set comparison operation.[9]

$$\begin{array}{r} M_{\texttt{SetComparison}} \\ [p : \texttt{SetComparison}, \\ s : \mathbb{S}, vals : \wp(\mathbb{V} \times \wp(\mathbb{T}))] \end{array} \triangleq \begin{cases} M_{\texttt{SetOfTasks}}(p.tasks_1, s, vals) = M_{\texttt{SetOfTasks}}(p.tasks_2, s, vals) & p.oper \texttt{ is Identical} \\ M_{\texttt{SetOfTasks}}(p.tasks_1, s, vals) \neq M_{\texttt{SetOfTasks}}(p.tasks_2, s, vals) & p.oper \texttt{ is Different} \\ M_{\texttt{SetOfTasks}}(p.tasks_1, s, vals) \cap M_{\texttt{SetOfTasks}}(p.tasks_2, s, vals) \neq \varnothing & p.oper \texttt{ is OverlapsWith} \\ M_{\texttt{SetOfTasks}}(p.tasks_1, s, vals) \subseteq M_{\texttt{SetOfTasks}}(p.tasks_2, s, vals) & p.oper \texttt{ is SubsetOf} \\ M_{\texttt{SetOfTasks}}(p.tasks_1, s, vals) \subset M_{\texttt{SetOfTasks}}(p.tasks_2, s, vals) & p.oper \texttt{ is ProperSubsetOf} \end{cases}$$

The denotation of a specimen of `UnaryPredicate` is a unary behavioral relation.

$$\begin{array}{r} M_{\texttt{UnaryPredicate}}[p : \texttt{UnaryPredicate}, \\ s : \mathbb{S}, vals : \wp(\mathbb{V} \times \wp(\mathbb{T}))] \end{array} \triangleq \begin{cases} canOccur(s, M_{\texttt{Task}}(p.task)) & p.name \texttt{ is CanOccur} \\ alwaysOccurs(s, M_{\texttt{Task}}(p.task)) & p.name \texttt{ is AlwaysOccurs} \end{cases}$$

Note that the behavioral relation is computed for the system matched to the query. Similarly, the meaning of a specimen of `BinaryPredicate` is a binary behavioral relation. The reader can refer to Section 5.5 to learn about methods for computing unary and binary PQL predicates.

[9]As the meaning of the `SetOfTasks` construct is a set composed of the meanings of all and only its `Task` elements, each interpreted as a set of character strings, the identity of elements is defined as set equality over sets of character strings.

$$
\begin{array}{r}M_{\texttt{BinaryPredicate}}\\ [p : \texttt{BinaryPredicate},\\ s : \mathbb{S}, vals : \wp(\mathbb{V} \times \wp(\mathbb{T}))]\end{array} \triangleq \begin{cases}
\mathit{canConflict}(s, M_{\texttt{Task}}(p.task_1), M_{\texttt{Task}}(p.task_2)) & p.name \texttt{ is } \texttt{CanConflict} \\
\mathit{canCooccur}(s, M_{\texttt{Task}}(p.task_1), M_{\texttt{Task}}(p.task_2)) & p.name \texttt{ is } \texttt{CanCooccur} \\
\mathit{conflict}(s, M_{\texttt{Task}}(p.task_1), M_{\texttt{Task}}(p.task_2)) & p.name \texttt{ is } \texttt{Conflict} \\
\mathit{cooccur}(s, M_{\texttt{Task}}(p.task_1), M_{\texttt{Task}}(p.task_2)) & p.name \texttt{ is } \texttt{Cooccur} \\
\mathit{totalCausal}(s, M_{\texttt{Task}}(p.task_1), M_{\texttt{Task}}(p.task_2)) & p.name \texttt{ is } \texttt{TotalCausal} \\
\mathit{totalConcurrent}(s, M_{\texttt{Task}}(p.task_1), M_{\texttt{Task}}(p.task_2)) & p.name \texttt{ is } \texttt{TotalConcurrent}
\end{cases}
$$

PQL uses the mechanism of macros to combine the results of several unary or binary relations. A specimen of `UnaryPredicateMacro` combines several results of unary behavioral relations using existential and universal quantifiers.

$$
\begin{array}{r}M_{\texttt{UnaryPredicateMacro}}\\ [p : \texttt{UnaryPredicateMacro},\\ s : \mathbb{S}, vals : \wp(\mathbb{V} \times \wp(\mathbb{T}))]\end{array} \triangleq \begin{cases}
\exists\, t \in M_{\texttt{SetOfTasks}}(p.tasks, s, vals) : \mathit{canOccur}(s, t) & (p.name \texttt{ is CanOccur}) \wedge (p.q \texttt{ is Any}) \\
\forall\, t \in M_{\texttt{SetOfTasks}}(p.tasks, s, vals) : \mathit{canOccur}(s, t) & (p.name \texttt{ is CanOccur}) \wedge (p.q \texttt{ is All}) \\
\exists\, t \in M_{\texttt{SetOfTasks}}(p.tasks, s, vals) : \mathit{alwaysOccurs}(s, t) & (p.name \texttt{ is AlwaysOccurs}) \wedge (p.q \texttt{ is Any}) \\
\forall\, t \in M_{\texttt{SetOfTasks}}(p.tasks, s, vals) : \mathit{alwaysOccurs}(s, t) & (p.name \texttt{ is AlwaysOccurs}) \wedge (p.q \texttt{ is All})
\end{cases}
$$

For example, a specimen *p* of `UnaryPredicateMacro` such that *p.name* is of type `AlwaysOccurs` and *p.q* is of type `All` denotes the truth value of *true* in the context of system *s* iff for every task *t* in the set of tasks defined by *p.tasks* it holds that *t* occurs in every instance of *s*. PQL offers macros that relate a task to a set of tasks and macros that relate two sets of tasks. One can use a specimen of the `BinaryPredicateMacroTaskSet` construct to test if a task is in a binary behavioral relation with some (or all) tasks in a given set.

$$
\begin{array}{r}M_{\texttt{BinaryPredicateMacroTaskSet}}\\ [p : \texttt{BinaryPredicateMacroTaskSet},\\ s : \mathbb{S}, vals : \wp(\mathbb{V} \times \wp(\mathbb{T}))]\end{array} \triangleq \begin{cases}
\exists\, t \in M_{\texttt{SetOfTasks}}(p.tasks, s, vals) : \mathit{canConflict}(s, M_{\texttt{Task}}(p.task), t) & (p.name \texttt{ is CanConflict}) \wedge (p.q \texttt{ is Any}) \\
\forall\, t \in M_{\texttt{SetOfTasks}}(p.tasks, s, vals) : \mathit{canConflict}(s, M_{\texttt{Task}}(p.task), t) & (p.name \texttt{ is CanConflict}) \wedge (p.q \texttt{ is All}) \\
\ldots
\end{cases}
$$

We only specify the denotation of specimen *p* for which it holds that *p.name* is of type `CanConflict`. Note that the denotations of specimens for other binary predicate names are defined analogously. For instance, one can use specimen *p* of `BinaryPredicateMacroTaskSet` such that *p.name* is of type `TotalCausal` and *p.q* is of type `All` to check if the task defined by *p.task* is in the `TotalCausal` behavioral relation with every task in the set of tasks defined by *p.tasks*.

The denotation of a specimen of `BinaryPredicateMacroSetSet` is as follows.

$$
\begin{array}{r}
M_{\texttt{BinaryPredicateMacroSetSet}} \\
{[p : \texttt{BinaryPredicateMacroSetSet},} \\
s : \mathbb{S}, vals : \wp(\mathbb{V} \times \wp(\mathbb{T}))]
\end{array}
\triangleq
\begin{cases}
\begin{array}{r} \exists\, t_1 \in M_{\texttt{SetOfTasks}}(p.tasks_1, s, vs) \\ \exists\, t_2 \in M_{\texttt{SetOfTasks}}(p.tasks_2, s, vs) : \\ canConflict(s, t_1, t_2) \end{array} & \begin{array}{r} (p.name \texttt{ is} \\ \texttt{CanConflict}) \wedge \\ (p.q \texttt{ is Any}) \end{array} \\
\begin{array}{r} \exists\, t_1 \in M_{\texttt{SetOfTasks}}(p.tasks_1, s, vs) \\ \forall\, t_2 \in M_{\texttt{SetOfTasks}}(p.tasks_2, s, vs) : \\ canConflict(s, t_1, t_2) \end{array} & \begin{array}{r} (p.name \texttt{ is} \\ \texttt{CanConflict}) \wedge \\ (p.q \texttt{ is Some}) \end{array} \\
\begin{array}{r} \forall\, t_1 \in M_{\texttt{SetOfTasks}}(p.tasks_1, s, vs) \\ \exists\, t_2 \in M_{\texttt{SetOfTasks}}(p.tasks_2, s, vs) : \\ canConflict(s, t_1, t_2) \end{array} & \begin{array}{r} (p.name \texttt{ is} \\ \texttt{CanConflict}) \wedge \\ (p.q \texttt{ is Each}) \end{array} \\
\begin{array}{r} \forall\, t_1 \in M_{\texttt{SetOfTasks}}(p.tasks_1, s, vs) \\ \forall\, t_2 \in M_{\texttt{SetOfTasks}}(p.tasks_2, s, vs) : \\ canConflict(s, t_1, t_2) \end{array} & \begin{array}{r} (p.name \texttt{ is} \\ \texttt{CanConflict}) \wedge \\ (p.q \texttt{ is All}) \end{array} \\
\ldots
\end{cases}
$$

Note that the missing definitions are similar to those shown above. For example, one can use a specimen p of `BinaryPredicateMacroSetSet` such that *p.name* is of type `Conflict` and *p.q* is of type `Any` to check if some task in the set of tasks defined by $p.tasks_1$ is in the `Conflict` behavioral relation with some task in the set of tasks defined by $p.tasks_2$.

Appendix E. Implementation Details

This appendix contains further details on PQL tools presented in Section 6.

The PQL Bot. When initializing a PQL bot instance, one can configure it via CLI. Some options of the PQL bot CLI are listed in Table E.11. Every PQL bot instance has a unique name, which can be assigned using option `-n`. If this option is not used, a random unique name is assigned. Once started, a PQL bot instance indexes stored (but not yet indexed) models in succession. One can use CLI options `-s` and `-i` to specify the time to sleep, i.e., to stay idle, between two successive indexing tasks, and the maximal time to attempt indexing of a model. If these options are not used, the parameters get configured based on the values in the global configuration file. If indexing of a model can not be completed within the given time frame, the model is marked as such that can not be indexed using this version of the bot, and the bot proceeds with indexing the next model. The `-h` and `-v` CLI options, respectively, print the help message and get the version of the invoked PQL bot instance.

Table E.11: CLI options of the PQL bot.

Option	**Option (short)**	**Parameter**	**Description**
`--help`	`-h`		Print help message
`--index`	`-i`	`<number>`	Maximal indexing time (in seconds)
`--name`	`-n`	`<string>`	Name of this bot (maximum 36 characters)
`--sleep`	`-s`	`<number>`	Time to sleep between indexing jobs (in seconds)
`--version`	`-v`		Get version of this bot

Once started, a PQL bot instance runs as a background process until it is shut down. An example command line output of a PQL bot instance is listed below.

```
>> java -jar PQL.BOT-1.0.jar -n=Brisbane -s=60 -i=86400
>> ====================================================================
>> Process Query Language (PQL) Bot ver. 1.0
>> ====================================================================
>> Name:               Brisbane
>> Sleep time:         60s
>> Max. index time:    86400s
>> ====================================================================
>> 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
>> ...
```

The PQL Tool. Table E.12 lists some CLI options of the PQL tool. The PQL tool allows a user to store a given model (option `-s`), check if a model can be indexed (option `-c`), index a model (option `-i`), delete a model and its index (option `-d`), visualize the parse tree of a given query (option `-p`), execute a query (options `-q`), and reset the PQL environment (option `-r`). The CLI can be used to access help information (option `-h`) and the version of the tool (option `-v`).

Table E.12: CLI options of the PQL tool.

Option	**Option (short)**	**Parameter**	**Description**	**Required opt.**
`--check`	`-c`		Check if model can be indexed	`-id`
`--delete`	`-d`		Delete model (and its index)	`-id`
`--help`	`-h`		Print help message	
`--index`	`-i`		Index model	`-id`
`--identifier`	`-id`	`<string>`	Model identifier	`-id`
`--parse`	`-p`		Show PQL query parse tree	`-pql`
`--pnmlPath`	`-pnml`	`<path>`	PNML path	
`--pqlPath`	`-pql`	`<path>`	PQL path	
`--query`	`-q`		Execute PQL query	`-pql`
`--reset`	`-r`		Reset this PQL instance	
`--store`	`-s`		Store model	`-pnml (-id)`
`--version`	`-v`		Get version of this tool	

To store models, the CLI option `-s` must be accompanied by option `-pnml` that specifies a path either to a single PNML file or to a directory that contains PNML files. If a path to a PNML file is used, the call to the PQL tool must include option `-id` to specify a unique identifier to associate with the model. Otherwise, models are attempted to be stored using their file names as unique identifiers. Once stored, a 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 that was used to store the model. When indexing a model, the PQL tool uses the same routines as the PQL bot.

Note that the dynamic semantics of PQL is implemented over sound workflow systems—a special class of Petri net systems. One can check whether a given Petri net system is a sound workflow system by calling the PQL tool with option `-c`. Note that every request to index a model in the PQL environment is automatically preceded by a soundness check of this model. One can delete a model using option `-d`. By deleting a model, the user also deletes its index. Both options `-c` and `-d` require option `-id` to uniquely identify a model to be checked and deleted, respectively. To execute a PQL query, a user can use option `-q` together with option `-pql` that specifies a path to a file that contains a PQL query captured using the concrete syntax proposed in Section 5.3. To visualize the parse tree of a PQL query, one can use option `-p` together with option `-pql`. Finally, one can reset the PQL environment using option `-r`. By resetting the environment, one deletes all stored models and indexes.

An example command line output of executing a PQL query discussed in Section 5.5 using the PQL tool is shown below.

```
>> java -jar PQL.TOOL-1.0.jar -q -pql=query.pql
>> PQL query:  SELECT * FROM * WHERE AlwaysOccurs("process payment"[0.75]);
>> Attributes: [UNIVERSE]
>> Locations:  [UNIVERSE]
>> Task:       "process payment"[0.75] -> ["process payment by cash",
>>             "process payment by check"]
>> Result:     [Fig.4.pnml]
```

Appendix F. Discussion of Evaluation Results

This appendix contains discussions of the results reported in Section 7.

Datasets. Tables 7 and 8 report statistics on the structural properties of the two model collections used in the evaluation. By XOR-join and XOR-split we refer to a place with multiple input transitions and a place with multiple output transitions, respectively. By AND-join and AND-split we refer to a transition with multiple input places and a transition with multiple output places, respectively. Polygons, bonds, and rigids are different types of SESE components in the WF-tree of a workflow system [89], where a polygon is a sequence of SESE components in which every two subsequent components share one node, a bond is a collection of SESE components that share entry and exit nodes, and a rigid is an unstructured component [15].

Queries. Table 9 lists all the PQL query categories, groups, and subgroups and provides the number of query templates used in the evaluation. The first category contains six query templates capturing individual atomic behavioral predicates. These can be divided into two groups, one covering two unary predicates (1.a) and the other covering four binary predicates (1.b).

The second category is a result of combining atomic predicates via logical operations. This category has two groups, one for connecting the same predicates (2.a), and the other for connecting mixed predicates (2.b). The former group can be further divided into three subgroups which capture, respectively, the negation of each predicate (2.a.1), the conjunctions of each predicate twice, three, or four times (2.a.2), and the disjunctions of each predicate twice, three, or four times (2.a.3). The latter group also has three subgroups which capture the conjunctions of two, three, or four different predicates (2.b.1), the disjunctions of two, three, or four different predicates (2.b.2), and different combinations of three logical operations between mixed predicates (2.b.3).

The third category captures predicate macros in one group (3.a) and the construction of task sets using predicates in the other group (3.b). The first group has three subgroups. One subgroup applies each of the unary predicate macros over a task set of two, three, or four tasks in conjunction (`ALL`) or disjunction (`ANY`) (3.a.1); one subgroup applies each of the binary predicate macros between a single task and a task set of two, three, or four tasks in conjunction or disjunction (3.a.2); and one subgroup applies each of the binary predicate macros between two task sets each consisting of two, three, or four tasks (3.a.3). The second group has four subgroups. The first two subgroups apply the set predicate `TaskInSetOfTasks` to a task set that is constructed using unary behavioral predicates (3.b.1) or binary predicates (3.b.2). The last two subgroups capture task set constructions using mixed behavioral predicates with set operations (3.b.3) or set comparisons (3.b.4).

Experiment 1.1: Impact of PQL bots on indexing time. Fig. 9 plots the indexing times (in seconds) for different parts of process model collections against different numbers of bots for the (a) industrial and (b) synthetic models.

Indexing the whole collection of industrial models with one bot took 6 hours and 54 minutes. Two bots managed to index 493 systems in 3 hours and 28 minutes (approximately two times faster than with one bot). Note that eight bots spent 1 hour and 12 minutes indexing the same collection (5.8 times faster than with one bot). A similar trend can be observed for the synthetic models. The relationship between the indexing time and the number of bots is best described by the power function $y = t \times x^k$, where x is the number of bots and y is the indexing time. For the industrial process models, the estimated values for constant t are 6,065.6, 11,241, 16,735, and 23,071 for 25%, 50%, 75%, and 100% of models in the collection, respectively. The estimated values for coefficient k are -0.845, -0.834, -0.826, and -0.839 for 25%, 50%, 75%, and 100% of models in the industrial collection, respectively. For the synthetic process models, the estimated values for constant t are 17,991, 35,494, 52,086, and 68,077 for 25%, 50%, 75%, and 100% of models in the collection, respectively. The estimated values for coefficient k are -0.833, -0.806, -0.827, and -0.834 for 25%, 50%, 75%, and 100% of models in the synthetic collection, respectively. The coefficient of determination R^2 ranges from 0.9901 to 0.9936 for the industrial process models and from 0.9834 to 0.9884 for the synthetic process models, indicating that the fitted functions can explain most of the variance in the indexing time.

This experiment shows that the indexing time grows linearly with the size of a process model collection. Using one PQL bot, 25% of models in the industrial collection were indexed in 1 hour and 48 minutes, 50% were indexed in 3 hours and 22 minutes, 75% in 5 hours and 2 minutes, and the whole collection was indexed in 6 hours and 54 minutes. This relationship between the indexing time and the collection size is best captured by the linear function $y = 49.549x + 158.65$, where x is the number of models in the collection and y is the indexing time. The coefficient of determination R^2 for the above example is 0.9985. The coefficients of determination R^2 for the fitted linear functions on four data points range from 0.9985 to 0.9997 (for different numbers of bots). We observed the same trend for the synthetic collection, with R^2 values ranging from 0.9949 to 0.9992.

Experiment 1.2: Impact of model size on indexing time. Fig. 10 plots the indexing times (in seconds) against different sizes of workflow systems for the industrial and synthetic models.

The relation between the indexing time and the model size in the industrial collection is best approximated by the power function $y = 0.8171 \times x^{1.7579}$, which results in a coefficient of determination R^2 of 0.9915. One obvious outlier workflow system in the industrial collection (a model with 25 observable transitions whose indexing took 858.7 seconds) can be explained by the much bigger size of its state space (2,097,422 reachable states, measured by the LoLA tool) compared to the sizes of state spaces of the other models in the collection (equal or less than 262,156 reachable states). For the synthetic models, the relation between the indexing time and the model size is best explained by the power function $y = 0.7423 \times x^{1.7775}$, which results in $R^2 = 0.99$.

Given a system, either industrial or synthetic, we noticed that it could be classified as an outlier using the value $f(x,y) = x^2 \times log(y)$, where x is the number of observable transitions of the system and y is the size of the state space of the system in terms of the number of its reachable states. The Pearson's correlation coefficient between the indexing times and the values of f is 0.9426 and 0.9692 for the industrial and synthetic models, respectively. The residual analysis has revealed that the four outliers in the industrial collection are among the six models with the highest values of f, while the seven outliers in the synthetic collection are among the eight models with the highest values of f. The outliers were identified as models with standardized residuals beyond ±3 units. The four outliers in the industrial collection have 25 (5 minutes and 38 seconds to index), 25 (14 minutes and 19 seconds), 32 (8 minutes and 48 seconds), and 35 (8 minutes and 48 seconds) observable transitions. Their state spaces comprise 114,717, 2,097,422, 390, and 882 reachable states, respectively. The seven outliers in the synthetic collection have 36 (11 minutes and 23 seconds to index), 36 (10 minutes and 30 seconds), 42 (14 minutes and 58 seconds), 46 (15 minutes and 23 seconds), 56 (27 minutes and 10 seconds), 63 (39 minutes and 59 seconds), 67 (41 minutes and 6 seconds) observable transitions. Their state spaces have 303,118, 131,108, 103,694, 1,602, 5,730, 311,306, and 6,818 reachable states, respectively.

Experiment 1.4: Impact of index size on indexing time. We performed four indexing runs to observe the impact on the indexing time. In the first run, Set 1 was indexed first, followed by Set 2, followed by Set 3, and finally concluded by indexing Set 4. The second run indexed the models in the order of Set 4, Set 1, Set 2, and finally, Set 3. The third run indexed the models in the order of Set 3, Set 4, Set 1, and finally, Set 2. The fourth run indexed the models in the order of Set 2, Set 3, Set 4, and finally, Set 1. The same label similarity threshold of 1.0 was used in all the runs. All four indexing runs were accomplished by one PQL bot.

For all the sets, we observed that the index size does not significantly affect the average indexing time. For models in Set 1, the recorded average indexing times are 53.63 seconds, 53.65 seconds, 54.09 seconds, and 54.26 seconds in the first, second, third, and fourth run, respectively. For models in Set 2, the recorded average indexing times are 53.29 seconds, 53.46 seconds, 53.70 seconds, and 53.80 seconds in the first, second, third, and fourth run, respectively. For models in Set 3, the recorded average indexing times are 44.75 seconds, 44.79 seconds, 44.93 seconds, and 45.15 seconds in the first, second, third, and fourth run, respectively. Finally, for models in Set 4, the recorded average indexing times are 49.6 seconds, 49.52 seconds, 49.9 seconds, and 49.92 seconds in the first, second, third, and fourth run, respectively. Given any set out of the four sets of models, the relation between the index size (used as a starting point to index the given set) and the average indexing time of a model in the set is best approximated by a polynomial function. The average indexing time for an industrial model in different sets at different positions in the indexing queue ranges from 44.75 to 54.26 seconds, with the difference between the maximum and minimum average indexing time for a given set at different indexing positions always being within 0.65 seconds. The measured coefficients of determination are equal to 0.9215, 0.9847, 0.9998, and 0.7366 for Set 1, Set 2, Set 3, and Set 4, respectively. The experiment demonstrates a general trend of a negligible increase in the indexing time with the growth of the index size. This observation can be explained by the fact that the introduced overhead is due to write operations on the PQL index, which modern database management systems can efficiently handle.

In the measurements of the average indexing times for the synthetic collection, we observed a similar trend as in the measurements for the industrial collection. In particular, given any set out of the four sets of models in the synthetic collection (note that each set of synthetic models is composed of 250 random models), the relation between the index size (used as a starting point to index the given set) and the average indexing time of a model in the set is best approximated by a polynomial function. The average indexing time for a synthetic model in different sets at different positions in the indexing queue ranges from 61.7 to 86.74 seconds, with the difference between the maximum and minimum average indexing time for a given set at different indexing positions always being within 0.75 seconds. The measured coefficients of determination are equal to 0.8456, 0.9947, 0.9999, and 0.8545 for Set 1, Set 2, Set 3, and Set 4, respectively.

Experiment 2.1: Impact of query threads on querying time. Fig. 11 plots the querying times (in seconds) for different parts of process model collections against different numbers of query threads. The relation between the querying time and the number of threads is best captured by the power function $y = t \times x^k$, where y is the querying time, and x is the number of query threads. For the industrial process models, the estimated values for constant t are 2.0793, 4.1112, 6.3066, and 8.3917 for 25%, 50%, 75%, and 100% of models in the collection, respectively. The estimated values for coefficient k are -0.582, -0.584, -0.595, and -0.594 for 25%, 50%, 75%, and 100% of models in the industrial collection, respectively. For the synthetic process models, the estimated values for constant t are 4.1811, 8.6902, 13.693, and 19.444 for 25%, 50%, 75%, and 100% of models in the collection, respectively. While the estimated values for coefficient k are -0.581, -0.597, -0.604, and -0.612 for 25%, 50%, 75%, and 100% of models in the synthetic collection, respectively. The coefficient of determination R^2 ranges from 0.9969 to 0.9982 for the industrial models and from 0.9983 to 0.9989 for the synthetic models, indicating that the fitted models can explain most of the variance in the querying time.

The measurements obtained in this experiment can be used to show that the querying time grows linearly with the size of a process model collection. For example, the measured average querying time with one query thread over 25% of the industrial models is 2.037 seconds, over 50% is 4.037 seconds, over 75% is 6.109 seconds, and over the whole collection is 8.259 seconds. This trend is best described by the linear relation $y = 0.0168 \times x - 0.0658$, where x is the number of models in the collection and y is the querying time. The coefficient of determination R^2 for the above example is 0.9998. The coefficients of determination R^2 for the fitted linear functions on four data points range from 0.9988 to 1.0 (for different numbers of query threads). We observed the same trend for the synthetic collection, with R^2 values ranging from 0.9967 to 0.9998.

Experiment 2.2: Impact of query types on querying time. Figs. 12(a) and 12(b) show the average querying times for the collection of industrial models for queries in Categories 1 and 2, while Figs. 12(c) and 12(d) show the average querying times for queries in Category 3. The average querying times for the synthetic collection are shown in Figs. 12(e) and 12(f) (Categories 1 and 2) and Figs. 12(g) and 12(h) (Category 3). Queries in Category 1 only comprise atomic predicates and, thus, are the fastest. The measured average querying time for the Category 1 queries using one query thread is 1.75 seconds for the 493 models in the industrial collection (approximately 3.5ms per one model-query check) and 4.84 seconds for the 1,000 models in the synthetic collection (approximately 4.8ms per one model-query check). With eight query threads, the Category 1 queries were, on average, accomplished in 0.47 seconds for the 493 models in the industrial collection and 1.61 seconds for the 1,000 models in the synthetic collection. Queries in Category 2 comprise atomic predicates and logical connectives. Thus, they require more time to accomplish than queries in Category 1. The measured average querying time for Category 2 queries with one thread is 2.8 seconds for the industrial models and 8.2 seconds for the synthetic models. With eight query threads, it is 0.84 and 2.52 seconds for the industrial and synthetic models, respectively. Queries in Category 3 comprise macros and, hence, are the lengthiest. The average querying time for Category 3 queries with one query thread is 10.89 seconds for the industrial models (approximately 11ms per one model-query check) and 24.76 seconds for the synthetic models (approximately 25ms per one model-query check). With eight query threads, it is 3.23 and 6.96 seconds for the industrial and synthetic models, respectively.

Figs. 12(a), 12(c), 12(e) and 12(g) demonstrate the linear dependency between the number of models in a collection and querying time for different query types. The coefficients of determination R^2 for the fitted linear functions for different query subgroups range from 0.9718 to 1.0 for the industrial models and from 0.9592 to 0.9992 for the synthetic models. Finally, Figs. 12(b), 12(d), 12(f), and 12(h) show that for all the query subgroups the relation between the querying time and the number of query threads follows the trend observed in Experiment 2.1. The coefficients of determination R^2 for the fitted power functions for different query subgroups range from 0.9724 to 0.998 for the industrial models and from 0.9329 to 0.9955 for the synthetic models.

Appendix G. PQL Queries

This appendix contains 150 PQL query templates used in the evaluation reported in Section 7. Each template is a PQL query with placeholders for activity labels. The templates can be instantiated with specific labels and used as a benchmark to evaluate the performance of PQL tools. The query templates were developed to exploit the various features of the PQL grammar. According to the PQL features they support, these query templates are divided into three categories and further subdivided into groups and subgroups; refer to Section 7.1 for details. The PQL query templates are listed in Table G.13.

Table G.13: PQL query templates.

No.	ID	PQL template
1	1.a.1	`SELECT * FROM * WHERE CanOccur("L1");`
2	1.a.2	`SELECT * FROM * WHERE AlwaysOccurs("L1");`
3	1.b.1	`SELECT * FROM * WHERE Cooccur("L1","L2");`
4	1.b.2	`SELECT * FROM * WHERE Conflict("L1","L2");`
5	1.b.3	`SELECT * FROM * WHERE TotalCausal("L1","L2");`
6	1.b.4	`SELECT * FROM * WHERE TotalConcurrent("L1","L2");`
7	2.a.1.1	`SELECT * FROM * WHERE NOT CanOccur("L1");`
8	2.a.1.2	`SELECT * FROM * WHERE NOT AlwaysOccurs("L1");`
9	2.a.1.3	`SELECT * FROM * WHERE NOT Cooccur("L1","L2");`
10	2.a.1.4	`SELECT * FROM * WHERE NOT Conflict("L1","L2");`
11	2.a.1.5	`SELECT * FROM * WHERE NOT TotalCausal("L1","L2");`
12	2.a.1.6	`SELECT * FROM * WHERE NOT TotalConcurrent("L1","L2");`
13	2.a.2.1	`SELECT * FROM * WHERE CanOccur("L1") AND CanOccur("L2");`
14	2.a.2.2	`SELECT * FROM * WHERE CanOccur("L1") AND CanOccur("L2") AND CanOccur("L3");`
15	2.a.2.3	`SELECT * FROM * WHERE CanOccur("L1") AND CanOccur("L2") AND CanOccur("L3") AND CanOccur("L4");`
16	2.a.2.4	`SELECT * FROM * WHERE AlwaysOccurs("L1") AND AlwaysOccurs("L2");`

17	2.a.2.5	`SELECT * FROM * WHERE AlwaysOccurs("L1") AND AlwaysOccurs("L2") AND AlwaysOccurs("L3");`
18	2.a.2.6	`SELECT * FROM * WHERE AlwaysOccurs("L1") AND AlwaysOccurs("L2") AND AlwaysOccurs("L3") AND AlwaysOccurs("L4");`
19	2.a.2.7	`SELECT * FROM * WHERE Cooccur("L1","L2") AND Cooccur("L3","L4");`
20	2.a.2.8	`SELECT * FROM * WHERE Cooccur("L1","L2") AND Cooccur("L3","L4") AND Cooccur("L5","L6");`
21	2.a.2.9	`SELECT * FROM * WHERE Cooccur("L1","L2") AND Cooccur("L3","L4") AND Cooccur("L5","L6") AND Cooccur("L7","L8");`
22	2.a.2.10	`SELECT * FROM * WHERE Conflict("L1","L2") AND Conflict("L3","L4");`
23	2.a.2.11	`SELECT * FROM * WHERE Conflict("L1","L2") AND Conflict("L3","L4") AND Conflict("L5","L6");`
24	2.a.2.12	`SELECT * FROM * WHERE Conflict("L1","L2") AND Conflict("L3","L4") AND Conflict("L5","L6") AND Conflict("L7","L8");`
25	2.a.2.13	`SELECT * FROM * WHERE TotalCausal("L1","L2") AND TotalCausal("L3","L4");`
26	2.a.2.14	`SELECT * FROM * WHERE TotalCausal("L1","L2") AND TotalCausal("L3","L4") AND TotalCausal("L5","L6");`
27	2.a.2.15	`SELECT * FROM * WHERE TotalCausal("L1","L2") AND TotalCausal("L3","L4") AND TotalCausal("L5","L6") AND TotalCausal("L7","L8");`
28	2.a.2.16	`SELECT * FROM * WHERE TotalConcurrent("L1","L2") AND TotalConcurrent("L3","L4");`
29	2.a.2.17	`SELECT * FROM * WHERE TotalConcurrent("L1","L2") AND TotalConcurrent("L3","L4") AND TotalConcurrent("L5","L6");`
30	2.a.2.18	`SELECT * FROM * WHERE TotalConcurrent("L1","L2") AND TotalConcurrent("L3","L4") AND TotalConcurrent("L5","L6") AND TotalConcurrent("L7","L8");`
31	2.a.3.1	`SELECT * FROM * WHERE CanOccur("L1") OR CanOccur("L2");`
32	2.a.3.2	`SELECT * FROM * WHERE CanOccur("L1") OR CanOccur("L2") OR CanOccur("L3");`
33	2.a.3.3	`SELECT * FROM * WHERE CanOccur("L1") OR CanOccur("L2") OR CanOccur("L3") OR CanOccur("L4");`
34	2.a.3.4	`SELECT * FROM * WHERE AlwaysOccurs("L1") OR AlwaysOccurs("L2");`
35	2.a.3.5	`SELECT * FROM * WHERE AlwaysOccurs("L1") OR AlwaysOccurs("L2") OR AlwaysOccurs("L3");`
36	2.a.3.6	`SELECT * FROM * WHERE AlwaysOccurs("L1") OR AlwaysOccurs("L2") OR AlwaysOccurs("L3") OR AlwaysOccurs("L4");`
37	2.a.3.7	`SELECT * FROM * WHERE Cooccur("L1","L2") OR Cooccur("L3","L4");`
38	2.a.3.8	`SELECT * FROM * WHERE Cooccur("L1","L2") OR Cooccur("L3","L4") OR Cooccur("L5","L6");`
39	2.a.3.9	`SELECT * FROM * WHERE Cooccur("L1","L2") OR Cooccur("L3","L4") OR Cooccur("L5","L6") OR Cooccur("L7","L8");`
40	2.a.3.10	`SELECT * FROM * WHERE Conflict("L1","L2") OR Conflict("L3","L4");`
41	2.a.3.11	`SELECT * FROM * WHERE Conflict("L1","L2") OR Conflict("L3","L4") OR Conflict("L5","L6");`
42	2.a.3.12	`SELECT * FROM * WHERE Conflict("L1","L2") OR Conflict("L3","L4") OR Conflict("L5","L6") OR Conflict("L7","L8");`
43	2.a.3.13	`SELECT * FROM * WHERE TotalCausal("L1","L2") OR TotalCausal("L3","L4");`
44	2.a.3.14	`SELECT * FROM * WHERE TotalCausal("L1","L2") OR TotalCausal("L3","L4") OR TotalCausal("L5","L6");`

45	2.a.3.15	`SELECT * FROM * WHERE TotalCausal("L1","L2") OR TotalCausal("L3","L4") OR TotalCausal("L5","L6") OR TotalCausal("L7","L8");`
46	2.a.3.16	`SELECT * FROM * WHERE TotalConcurrent("L1","L2") OR TotalConcurrent("L3","L4");`
47	2.a.3.17	`SELECT * FROM * WHERE TotalConcurrent("L1","L2") OR TotalConcurrent("L3","L4") OR TotalConcurrent("L5","L6");`
48	2.a.3.18	`SELECT * FROM * WHERE TotalConcurrent("L1","L2") OR TotalConcurrent("L3","L4") OR TotalConcurrent("L5","L6") OR TotalConcurrent("L7","L8");`
49	2.b.1.1	`SELECT * FROM * WHERE CanOccur("L1") AND Conflict("L2","L3");`
50	2.b.1.2	`SELECT * FROM * WHERE AlwaysOccurs("L1") AND Cooccur("L2","L3") AND TotalConcurrent("L4","L5");`
51	2.b.1.3	`SELECT * FROM * WHERE Cooccur("L1","L2") AND TotalCausal("L3","L4") AND TotalConcurrent("L5","L6");`
52	2.b.2.1	`SELECT * FROM * WHERE AlwaysOccurs("L1") OR Cooccur("L2","L3");`
53	2.b.2.2	`SELECT * FROM * WHERE CanOccur("L1") OR AlwaysOccurs("L2") OR Conflict("L3","L4");`
54	2.b.2.3	`SELECT * FROM * WHERE Conflict("L1","L2") OR TotalCausal("L3","L4") OR TotalConcurrent("L5","L6");`
55	2.b.3.1	`SELECT * FROM * WHERE AlwaysOccurs("L1") OR (Cooccur("L2","L3") AND (NOT TotalCausal("L4","L5")));`
56	2.b.3.2	`SELECT * FROM * WHERE (CanOccur("L1") AND (NOT Conflict("L2","L3"))) OR (TotalConcurrent("L4","L5") AND AlwaysOccurs("L6"));`
57	3.a.1.1	`SELECT * FROM * WHERE CanOccur({"L1","L2"},ALL);`
58	3.a.1.2	`SELECT * FROM * WHERE CanOccur({"L1","L2","L3"},ALL);`
59	3.a.1.3	`SELECT * FROM * WHERE CanOccur({"L1","L2","L3","L4"},ALL);`
60	3.a.1.4	`SELECT * FROM * WHERE CanOccur({"L1","L2"},ANY);`
61	3.a.1.5	`SELECT * FROM * WHERE CanOccur({"L1","L2","L3"},ANY);`
62	3.a.1.6	`SELECT * FROM * WHERE CanOccur({"L1","L2","L3","L4"},ANY);`
63	3.a.1.7	`SELECT * FROM * WHERE AlwaysOccurs({"L1","L2"},ALL);`
64	3.a.1.8	`SELECT * FROM * WHERE AlwaysOccurs({"L1","L2","L3"},ALL);`
65	3.a.1.9	`SELECT * FROM * WHERE AlwaysOccurs({"L1","L2","L3","L4"},ALL);`
66	3.a.1.10	`SELECT * FROM * WHERE AlwaysOccurs({"L1","L2"},ANY);`
67	3.a.1.11	`SELECT * FROM * WHERE AlwaysOccurs({"L1","L2","L3"},ANY);`
68	3.a.1.12	`SELECT * FROM * WHERE AlwaysOccurs({"L1","L2","L3","L4"},ANY);`
69	3.a.2.1	`SELECT * FROM * WHERE Cooccur("L1",{"L2","L3"},ALL);`
70	3.a.2.2	`SELECT * FROM * WHERE Cooccur("L1",{"L2","L3","L4"},ALL);`
71	3.a.2.3	`SELECT * FROM * WHERE Cooccur("L1",{"L2","L3","L4","L5"},ALL);`
72	3.a.2.4	`SELECT * FROM * WHERE Cooccur("L1",{"L2","L3"},ANY);`
73	3.a.2.5	`SELECT * FROM * WHERE Cooccur("L1",{"L2","L3","L4"},ANY);`
74	3.a.2.6	`SELECT * FROM * WHERE Cooccur("L1",{"L2","L3","L4","L5"},ANY);`
75	3.a.2.7	`SELECT * FROM * WHERE Conflict("L1",{"L2","L3"},ALL);`
76	3.a.2.8	`SELECT * FROM * WHERE Conflict("L1",{"L2","L3","L4"},ALL);`
77	3.a.2.9	`SELECT * FROM * WHERE Conflict("L1",{"L2","L3","L4","L5"},ALL);`
78	3.a.2.10	`SELECT * FROM * WHERE Conflict("L1",{"L2","L3"},ANY);`
79	3.a.2.11	`SELECT * FROM * WHERE Conflict("L1",{"L2","L3","L4"},ANY);`
80	3.a.2.12	`SELECT * FROM * WHERE Conflict("L1",{"L2","L3","L4","L5"},ANY);`
81	3.a.2.13	`SELECT * FROM * WHERE TotalCausal("L1",{"L2","L3"},ALL);`
82	3.a.2.14	`SELECT * FROM * WHERE TotalCausal("L1",{"L2","L3","L4"},ALL);`

83	3.a.2.15	`SELECT * FROM * WHERE TotalCausal("L1",{"L2","L3","L4","L5"},ALL);`
84	3.a.2.16	`SELECT * FROM * WHERE TotalCausal("L1",{"L2","L3"},ANY);`
85	3.a.2.17	`SELECT * FROM * WHERE TotalCausal("L1",{"L2","L3","L4"},ANY);`
86	3.a.2.18	`SELECT * FROM * WHERE TotalCausal("L1",{"L2","L3","L4","L5"},ANY);`
87	3.a.2.19	`SELECT * FROM * WHERE TotalConcurrent("L1",{"L2","L3"},ALL);`
88	3.a.2.20	`SELECT * FROM * WHERE TotalConcurrent("L1",{"L2","L3","L4"},ALL);`
89	3.a.2.21	`SELECT * FROM * WHERE TotalConcurrent("L1",{"L2","L3","L4","L5"},ALL);`
90	3.a.2.22	`SELECT * FROM * WHERE TotalConcurrent("L1",{"L2","L3"},ANY);`
91	3.a.2.23	`SELECT * FROM * WHERE TotalConcurrent("L1",{"L2","L3","L4"},ANY);`
92	3.a.2.24	`SELECT * FROM * WHERE TotalConcurrent("L1",{"L2","L3","L4","L5"},ANY);`
93	3.a.3.1	`SELECT * FROM * WHERE Cooccur({"L1","L2"},{"L3","L4"},ALL);`
94	3.a.3.2	`SELECT * FROM * WHERE Cooccur({"L1","L2","L3"},{"L4","L5","L6"},ALL);`
95	3.a.3.3	`SELECT * FROM * WHERE Cooccur({"L1","L2","L3","L4"},{"L5","L6","L7","L8"},ALL);`
96	3.a.3.4	`SELECT * FROM * WHERE Cooccur({"L1","L2"},{"L3","L4"},ANY);`
97	3.a.3.5	`SELECT * FROM * WHERE Cooccur({"L1","L2","L3"},{"L4","L5","L6"},ANY);`
98	3.a.3.6	`SELECT * FROM * WHERE Cooccur({"L1","L2","L3","L4"},{"L5","L6","L7","L8"},ANY);`
99	3.a.3.7	`SELECT * FROM * WHERE Cooccur({"L1","L2"},{"L3","L4"},EACH);`
100	3.a.3.8	`SELECT * FROM * WHERE Cooccur({"L1","L2","L3"},{"L4","L5","L6"},EACH);`
101	3.a.3.9	`SELECT * FROM * WHERE Cooccur({"L1","L2","L3","L4"},{"L5","L6","L7","L8"},EACH);`
102	3.a.3.10	`SELECT * FROM * WHERE Conflict({"L1","L2"},{"L3","L4"},ALL);`
103	3.a.3.11	`SELECT * FROM * WHERE Conflict({"L1","L2","L3"},{"L4","L5","L6"},ALL);`
104	3.a.3.12	`SELECT * FROM * WHERE Conflict({"L1","L2","L3","L4"},{"L5","L6","L7","L8"},ALL);`
105	3.a.3.13	`SELECT * FROM * WHERE Conflict({"L1","L2"},{"L3","L4"},ANY);`
106	3.a.3.14	`SELECT * FROM * WHERE Conflict({"L1","L2","L3"},{"L4","L5","L6"},ANY);`
107	3.a.3.15	`SELECT * FROM * WHERE Conflict({"L1","L2","L3","L4"},{"L5","L6","L7","L8"},ANY);`
108	3.a.3.16	`SELECT * FROM * WHERE Conflict({"L1","L2"},{"L3","L4"},EACH);`
109	3.a.3.17	`SELECT * FROM * WHERE Conflict({"L1","L2","L3"},{"L4","L5","L6"},EACH);`
110	3.a.3.18	`SELECT * FROM * WHERE Conflict({"L1","L2","L3","L4"},{"L5","L6","L7","L8"},EACH);`
111	3.a.3.19	`SELECT * FROM * WHERE TotalCausal({"L1","L2"},{"L3","L4"},ALL);`
112	3.a.3.20	`SELECT * FROM * WHERE TotalCausal({"L1","L2","L3"},{"L4","L5","L6"},ALL);`
113	3.a.3.21	`SELECT * FROM * WHERE TotalCausal({"L1","L2","L3","L4"},{"L5","L6","L7","L8"},ALL);`
114	3.a.3.22	`SELECT * FROM * WHERE TotalCausal({"L1","L2"},{"L3","L4"},ANY);`
115	3.a.3.23	`SELECT * FROM * WHERE TotalCausal({"L1","L2","L3"},{"L4","L5","L6"},ANY);`
116	3.a.3.24	`SELECT * FROM * WHERE TotalCausal({"L1","L2","L3","L4"},{"L5","L6","L7","L8"},ANY);`

117	3.a.3.25	`SELECT * FROM * WHERE TotalCausal({"L1","L2"},{"L3","L4"},EACH);`

118	3.a.3.26	`SELECT * FROM * WHERE`
		`TotalCausal({"L1","L2","L3"},{"L4","L5","L6"},EACH);`

119	3.a.3.27	`SELECT * FROM * WHERE`
		`TotalCausal({"L1","L2","L3","L4"},{"L5","L6","L7","L8"},EACH);`

120	3.a.3.28	`SELECT * FROM * WHERE TotalConcurrent({"L1","L2"},{"L3","L4"},ALL);`

121	3.a.3.29	`SELECT * FROM * WHERE`
		`TotalConcurrent({"L1","L2","L3"},{"L4","L5","L6"},ALL);`

122	3.a.3.30	`SELECT * FROM * WHERE`
		`TotalConcurrent({"L1","L2","L3","L4"},{"L5","L6","L7","L8"},ALL);`

123	3.a.3.31	`SELECT * FROM * WHERE TotalConcurrent({"L1","L2"},{"L3","L4"},ANY);`

124	3.a.3.32	`SELECT * FROM * WHERE`
		`TotalConcurrent({"L1","L2","L3"},{"L4","L5","L6"},ANY);`

125	3.a.3.33	`SELECT * FROM * WHERE`
		`TotalConcurrent({"L1","L2","L3","L4"},{"L5","L6","L7","L8"},ANY);`

126	3.a.3.34	`SELECT * FROM * WHERE TotalConcurrent({"L1","L2"},{"L3","L4"},EACH);`

127	3.a.3.35	`SELECT * FROM * WHERE`
		`TotalConcurrent({"L1","L2","L3"},{"L4","L5","L6"},EACH);`

128	3.a.3.36	`SELECT * FROM * WHERE`
		`TotalConcurrent({"L1","L2","L3","L4"},{"L5","L6","L7","L8"},EACH);`

129	3.b.1.1	`SELECT * FROM * WHERE`
		`"L1" IN GetTasksCanOccur({"L2","L3","L4"});`

130	3.b.1.2	`SELECT * FROM * WHERE`
		`"L1" IN GetTasksAlwaysOccurs({"L2","L3","L4"});`

131	3.b.2.1	`SELECT * FROM * WHERE`
		`"L1" IN GetTasksCooccur({"L2","L3"},{"L4","L5","L6"},ALL);`

132	3.b.2.2	`SELECT * FROM * WHERE`
		`"L1" IN GetTasksCooccur({"L2","L3"},{"L4","L5","L6"},ANY);`

133	3.b.2.3	`SELECT * FROM * WHERE`
		`"L1" IN GetTasksConflict({"L2","L3"},{"L4","L5","L6"},ALL);`

134	3.b.2.4	`SELECT * FROM * WHERE`
		`"L1" IN GetTasksConflict({"L2","L3"},{"L4","L5","L6"},ANY);`

135	3.b.2.5	`SELECT * FROM * WHERE`
		`"L1" IN GetTasksTotalCausal({"L2","L3"},{"L4","L5","L6"},ALL);`

136	3.b.2.6	`SELECT * FROM * WHERE`
		`"L1" IN GetTasksTotalCausal({"L2","L3"},{"L4","L5","L6"},ANY);`

137	3.b.2.7	`SELECT * FROM * WHERE`
		`"L1" IN GetTasksTotalConcurrent({"L2","L3"},{"L4","L5","L6"},ALL);`

138	3.b.2.8	`SELECT * FROM * WHERE`
		`"L1" IN GetTasksTotalConcurrent({"L2","L3"},{"L4","L5","L6"},ANY);`

139	3.b.3.1	`SELECT * FROM * WHERE`
		`"L1" IN GetTasksAlwaysOccurs({"L2","L3","L4"}) UNION`
		`GetTasksTotalCausal({"L5","L6"},{"L7","L8","L9"},ALL);`

140	3.b.3.2	`SELECT * FROM * WHERE`
		`"L1" IN GetTasksCanOccur({"L2","L3","L4"}) INTERSECT`
		`GetTasksConflict({"L5","L6"},{"L7","L8","L9"},ANY);`

141	3.b.3.3	`SELECT * FROM * WHERE`
		`"L1" IN GetTasksCooccur({"L2","L3"},{"L4","L5","L6"},ANY) EXCEPT`
		`GetTasksTotalConcurrent({"L7","L8"},{"L9","L10","L11"},ANY);`

142	3.b.3.4	`SELECT * FROM * WHERE "L1" IN (GetTasksCooccur({"L2","L3"},{"L4","L5","L6"},ANY) EXCEPT GetTasksTotalConcurrent({"L7","L8"},{"L9","L10","L11"},ANY)) UNION (GetTasksCanOccur({"L12","L13","L14"}) INTERSECT GetTasksConflict({"L15","L16"},{"L17","L18","L19"},ALL));`
143	3.b.3.5	`SELECT * FROM * WHERE "L1" IN (GetTasksCooccur({"L2","L3"},{"L4","L5","L6"},ANY) UNION GetTasksTotalConcurrent({"L7","L8"},{"L9","L10","L11"},ALL)) INTERSECT (GetTasksCanOccur({"L12","L13","L14"}) EXCEPT GetTasksConflict({"L15","L16"},{"L17","L18","L19"},ALL));`
144	3.b.4.1	`SELECT * FROM * WHERE ({"L1","L2","L3"} EXCEPT GetTasksConflict({"L4","L5"},{"L6","L7","L8"},ALL)) EQUALS GetTasksTotalCausal({"L9","L10"},{"L11","L12","L13"},ALL);`
145	3.b.4.2	`SELECT * FROM * WHERE GetTasksCooccur({"L1","L2"},{"L3","L4","L5"},ANY) NOT EQUALS GetTasksTotalConcurrent({"L6","L7"},{"L8","L9","L10"},ANY);`
146	3.b.4.3	`SELECT * FROM * WHERE ({"L1","L2","L3"} EXCEPT GetTasksCooccur({"L4","L5"},{"L6","L7","L8"},ALL)) OVERLAPS WITH GetTasksConflict({"L9","L10"},{"L11","L12","L13"},ANY);`
147	3.b.4.4	`SELECT * FROM * WHERE ({"L1","L2","L3"} EXCEPT GetTasksAlwaysOccurs({"L4","L5","L6"}) IS SUBSET OF GetTasksCanOccur({"L7","L8","L9"}));`
148	3.b.4.5	`SELECT * FROM * WHERE GetTasksTotalCausal({"L1","L2"},{"L3","L4","L5"},ALL) IS PROPER SUBSET OF ({"L6","L7","L8"} EXCEPT GetTasksTotalConcurrent({"L9","L10"},{"L11","L12","L13"},ALL));`
149	3.b.4.6	`SELECT * FROM * WHERE (GetTasksCooccur({"L1","L2"},{"L3","L4","L5"},ALL)) EQUALS GetTasksTotalConcurrent({"L6","L7"},{"L8","L9","L10"},ALL) OR ((GetTasksCanOccur({"L11","L12","L13"}) OVERLAPS WITH GetTasksConflict({"L14","L15"},{"L16","L17","L18"},ALL)) AND (({"L19","L20","L21"} EXCEPT GetTasksTotalCausal({"L22","L23"},{"L24","L25","L26"},ANY)) IS SUBSET OF GetTasksAlwaysOccurs({"L27","L28","L29"})));`
150	3.b.4.7	`SELECT * FROM * WHERE (NOT (GetTasksCooccur({"L1","L2"},{"L3","L4","L5"},ANY) OVERLAPS WITH GetTasksTotalConcurrent({"L6","L7"},{"L8","L9","L10"},ANY))) AND ((GetTasksConflict({"L11","L12"},{"L13","L14","L15"},ALL) IS PROPER SUBSET OF GetTasksCanOccur({"L16","L17","L18"})) AND (({"L19","L20","L21"} EXCEPT GetTasksTotalCausal({"L22","L23"},{"L24","L25","L26"},ALL)) NOT EQUALS GetTasksAlwaysOccurs({"L27","L28","L29"})));`