# Process Query Language: Design, Implementation and Evaluation

ARTEM POLYVYANYY, ARTHUR H.M. TER HOFSTEDE, MARCELLO LA ROSA, and CHUN OUYANG, Queensland University of Technology

Organisations can derive significant benefits from the use of practices, techniques, and tools from the area of business process management. Through the concomitant focus on processes they may acquire a substantial number of process models and such large collections of models require management, including support for versioning, merging, conformance checking, and retrieval. Model retrieval thus far has mostly focussed on querying models using syntactic properties of their control flow rather than on exploiting semantic properties capturing aspects of model execution. While the latter is (much) more challenging, it is also more effective, especially in a context where process models serve as a basis for subsequent automation. The focus of this paper is to overcome the challenges associated with semantic querying of process model collections and thus unlocking its benefits. The first challenge concerns determining decidability of the building blocks of the semantic query language, i.e. semantic relations between tasks. This is important as not all such relations can necessarily be computed, especially if one allows them to be based on unrestricted temporal logic statements. The second challenge concerns useability, specifically, selecting those semantic relations among those that are computable that are also considered relevant and intuitive by domain experts. The third and final challenge is concerned with achieving acceptable performance of query evaluation. The evaluation of a query may require expensive checks on all process models (of which there may be thousands) and to deal with this suitable index structures should be put in place. The effectiveness of these structures should be demonstrated empirically.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; *Parallel programming*; D.2.0 [**Software Engineering**]: General; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; *Syntax*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Process models*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Query formulation*; *Relevance feedback*; *Search process*

General Terms: Design, Languages, Management, Theory

Additional Key Words and Phrases: Process, process model, process instance, process model collection, process model repository, querying, process querying, searching, retrieving

**ACM Reference Format:**
Artem Polyvyanyy, Arthur H.M. ter Hofstede, Marcello La Rosa, and Chun Ouyang. 2014–2015. Process Query Language: Design, Implementation and Evaluation. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2010), 13 pages.
DOI : http://dx.doi.org/10.1145/0000000.0000000

## 1. INTRODUCTION

Through the application of methods and techniques from the field of business process management, organisations can identify, model, analyse, deploy, and diagnose their business processes. This process-oriented thinking provides great benefits as making processes explicit through formal high-level representations, i.e. process models, allows them 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 organisation's environment, both internal and external. As a consequence, some organisations have collected large numbers of process models and their maintenance poses significant challenges.

---

Authors' address: A. Polyvyanyy, A.H.M. ter Hofstede, M. La Rosa, and C. Ouang, Business Process Management Discipline, Information Systems School, Science & Engineering Faculty, Queensland University of Technology, Brisbane, Australia.

DOI : http://dx.doi.org/10.1145/0000000.0000000

Process models tend to evolve over time, new models may merge, which may need to be informed by existing models, and models may need to be merged. To support these activities, it should be possible to query a potentially large process model repository to retrieve models with certain characteristics. Current process model query languages, of which perhaps the most prominent proponent is BPMN-Q, predominantly focus on *syntactic* aspects of process models. Hence queries are based on paths, which are formed by direct succession relations between model elements.

More powerful than a syntactic approach to querying would be an approach based on *semantic* relations between tasks, e.g. relations that capture that certain tasks need to be executed in order or in parallel or can never be executed as part of the same process instance. Semantic relations are essential when one needs to explore whether certain process behaviours are possible or not.

The added retrieval power of a semantic query language comes at a price. Semantic relations cover a broad spectrum of inter-task dependencies and may be expressed through temporal logic. Temporal logic is a form of modal logic that includes temporal operators and which can be used to reason over the behaviour of systems over time. Temporal logic is powerful enough to be able to express properties that are undecidable [Esparza and Nielsen 1994]. Hence a semantic query language needs to be careful in terms of the semantic inter-task dependencies that it supports.

Decidability of semantic inter-task relations is a factor in choosing which relations to support in a query language, but it is not the one. While some relations are decidable, their use may not be very intuitive to stakeholders, in this case experts that are likely to end up formulating queries over process model collections. It is important that a relation occurs frequently enough in queries to warrant support 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 as it is anticipated that stakeholders may wish to see the answers to their queries in (almost) real-time.

In this paper a specific process model query language is proposed that is based on a selection of semantic inter-task relations. The Process Query Language (PQL) is a special-purpose programming language for managing process models based on information about process instances that they describe. PQL programs are also called queries. The first version of the PQL language, proposed in this paper, allows formulating search intents for retrieving process models from collections or repositories thereof based on information held in process instances.

The selected semantic inter-task relations supported by PQL, termed *predicates* in PQL, are shown to be decidable through the application of model checking, an automated technique which, given a finite-state model of a system (e.g., a process model) and a formal property (e.g., a temporal logic formula), systematically checks whether this property holds for (a given state in) that model [Baier and Katoen 2008]. In addition, these predicates are validated with domain experts in terms of their perceived usefulness and intuitiveness. To further facilitate query formulation, PQL is not only provided with an abstract syntax but also a concrete one, which is inspired by SQL. The runtime environment for PQL makes uses of indexes to enhance the performance of query evaluation. Indexes are special data structures that improve the speed of computations of behavioral operators trading off time for their construction with space for their storage. Performance of query evaluation is demonstrated through a set of experiments with a real-life process model collection.

## 2. PQL SYNTAX

### 2.1. Abstract Syntax

This section discusses the syntax of the PQL language in the form of an *abstract syntax*, which is also often referred to as an *(abstract) grammar*. The grammar of the PQL language is defined using the notation introduced in [Meyer 1990]. In this notation, the abstract grammar of a programming language consists of a finite set of names of *constructs* and a finite set of *productions*, each associated with a construct. Each construct describes the structure of a set of objects, also called *specimens* of the language, using productions of three types; these are *aggregate*, *choice*, and *list* productions.

The top construct of the abstract grammar of the PQL language is `Query`. It captures the core structure of all PQL programs, i.e., queries.

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

The `Query` construct (see above, on the left hand side) is defined as an *aggregate* production composed of four components (see above, on the right hand side); in general, an aggregate production defines a construct that is made of a fixed number of components. The components are separated by semicolons, each preceded by a *tag* indicating its *role* within the construct. Thus, every PQL query is composed of variables, attributes, locations, and a predicate, which are distinguished via tags *vars*, *atts*, *locs*, and *pred*, respectively. Intuitively, a PQL query specifies a search intent to discover the attributes of *all* process models in the collection of models identified by the locations that satisfy the predicate, where the evaluation of the predicate relies on information stored in the variables. The order in which the various specimens are listed in aggregate productions is irrelevant for the sake of the abstract grammar specification. This order is important in the context of the next section, in which the concrete syntax of the PQL language is proposed.

The `Query` construct seen above defines a class of PQL queries. One can specify an instance of this class using abstract syntactic expressions. For example, the statement $\texttt{q} \triangleq \texttt{Query}(\textit{vars} : \textit{vs}; \textit{atts} : \textit{as}; \textit{locs} : \textit{ls}; \textit{pred} : p)$ defines a query having *vs*, *as*, *ls*, and *p*, as variables, attributes, locations, and a predicate, respectively (assuming that all the specimens, i.e., *vs*, *as*, *ls*, and *p*, are provided).

In PQL, variables, attributes, and locations are defined as *list* productions, where a list production defines a sequence of zero, one, or more specimens of another construct.

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

Therefore, a PQL query defines a sequence of zero, one, or more variables, denoted by $\texttt{Variable}^{*}$; the asterisk symbol stands for the *Kleene star*—its standard language theory meaning. Every sequence of attributes must contain at least one attribute, denoted by $\texttt{Attribute}^{+}$; note that the asterisk symbol is replaced by a plus sign to signify that the list of locations cannot be empty. Similarly, every sequence of locations must contain at least one location specimen.

PQL introduces a dedicated construct, denoted by `Variable`, to define *variables*.

$$\texttt{Variable} \triangleq \textit{name} : \texttt{VariableName}; \textit{tasks} : \texttt{SetOfTasks}$$

A PQL variable associates a symbolic name with a set of *tasks*, or to be more precise, with a collection of abstract concepts that represent tasks. Tasks are introduced in the PQL language to refer to atomic units of observable behavior that are captured in process models. Each variable is an aggregate of two constructs: a variable name ($\textit{name} : \texttt{VariableName}$), and a collection of tasks ($\textit{tasks} : \texttt{SetOfTasks}$). Such a separation of the variable name from its associated content allows the name to be used independently of the exact information it represents. Thus, a variable name can be bound to a set of tasks during run time, and the content of the set may change during evaluation of the query. When a predicate of some PQL query gets evaluated, every variable name that is mentioned in the predicate is replaced by the corresponding set of tasks.

The PQL language introduces the `Attribute` construct to allow specifying those process model properties that must be retrieved in a response to a successful query matching exercise.

$$\texttt{Attribute} \triangleq \texttt{Universe} \mid \texttt{AttributeID} \mid \texttt{AttributeName} \mid \texttt{AttributeModel}$$

A PQL attribute can be anything that identifies a single property or a collection of properties of a process model. In the first version of the PQL language, the `Attribute` construct is associated with a *choice* production that allows for four alternatives; in general, a choice production defines a construct as a set of alternatives. The alternatives are separated by vertical bar symbols. Every attribute is either the *universe* attribute, denoted by `Universe`, the *identifier* property, denoted by `AttributeID`, the *name* property, denoted by `AttributeName`, or a formal *specification* property,

denoted by AttributeModel. The universe attribute has a special meaning. It refers to a collection of all properties that are associated with process models in the process model repository.

In the PQL language, locations are used to address process models that are of interest to the search intents of queries, i.e., those models that should be matched against the queries.

$$\texttt{Location} \triangleq \texttt{Universe} \mid \texttt{LocationID} \mid \texttt{LocationDirectory}$$

A location can generally be anything that identifies a single process model or a collection of process models. It is defined as a *choice* production that allows for three alternatives. A location is either the *universe* location, denoted by Universe, an *identifier* location, denoted by LocationID, or a *directory* location, denoted by LocationDirectory. The universe location is designed to address all process model in the scope of the query (usually, all process models in the repository). Identifier locations are introduced in the PQL language to allow fine-grained targeting of models based on their unique identifiers. We assume that repositories do indeed tag models with unique identifiers, e.g., universally unique identifiers (UUIDs) or integer identifiers. Finally, a directory location allows addressing models that are stored in a particular directory of the repository. For example, a directory location can be a name of a directory, specified as a character string, an URI [URI Planning Interest Group 2001], or an XPath expression [W3C XSL/XML Query Working Groups 2007].

PQL provides several alternatives for specifying a set of tasks. A set of tasks can be defined as an enumeration of tasks, a result of standard operations on sets of tasks, information stored in a variable, a construction macro, or a dynamically-valued constant. These various possibilities are captured in the SetOfTasks construct of the PQL grammar.

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

The SetOfTasks construct is defined as a choice production. One can use the VariableName construct to refer to the set of tasks associated with a name of some variable. Alternatively, one can specify a set of tasks using the Universe construct. The Universe construct, when used in the context of a reference to a set of tasks, constitutes a dynamically-valued constant that refers to the set of all tasks of the process model currently being matched to the query. The content of this set should be created at initialization time, freshly for every new process model that gets matched to the query.

The PQL language proposes a notation to specify set of tasks literals, i.e., a notation for representing sets of tasks as fixed values. Set of tasks literals can be defined using the SetOfTasksLiteral construct, which is specified as a list production of zero, one, or more tasks.

$$\texttt{SetOfTasksLiteral} \triangleq \texttt{Task}^*$$

As mentioned above, tasks are abstract representations of atomic units of observable behavior.

$$\texttt{Task} \triangleq \textit{label} : \texttt{Label}; \textit{sim} : \texttt{Similarity}$$

A PQL task is defined as an aggregate of two components: a *label*, denoted by *label* : Label, and a *similarity* degree threshold, denoted by *sim* : Similarity. The idea is that given a label of a task one may be interested in all the tasks of which the label has (at least) a certain degree of similarity to the given label.

Another way to specify a set of tasks is to construct it. For this purpose, one can rely on the SetOfTasksConstruction construct, which is defined as a choice production below.

$$\texttt{SetOfTasksConstruction} \triangleq \texttt{UnaryPredicateConstruction} \mid \texttt{BinaryPredicateConstruction}$$

Given a set of tasks and a unary behavioral primitive, the UnaryPredicateConstruction construct can be used to compose a set of tasks that contains every task from the given set and for which the given behavioral primitive holds. The given behavioral primitive must be evaluated in the context of the process model that is being matched to the query. Similarly, the BinaryPredicateConstruction construct is introduced in the PQL language to allow selecting those tasks from a given set of tasks for which certain binary behavioral primitive holds, either with at least one or with all tasks taken from another given set of tasks. The choice of a quantifier type,

either the existential or universal, to be used during the above described selections is implemented via the AnyAll construct.

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

Both, the UnaryPredicateConstruction construct and the BinaryPredicateConstruction construct, are associated with aggregate productions. The AnyAll construct is specified as a choice between the Any qualifier versus the All qualifier, where Any and All stand for the existential quantifier type and the universal quantifier type, respectively. The PQL language uses the UnaryPredicateName construct and the BinaryPredicateName construct to refer to unary behavioral primitives and binary behavioral primitives, respectively. The first edition of the PQL language supports two unary and six binary behavioral primitives. These are the CanOccur and AlwaysOccurs unary behavioral primitives, and the CanConflict, CanCooccur, Conflict, Cooccur, TotalCausal, and TotalConcurrent binary behavioral primitives.

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

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

The PQL language proposes several ways to specify predicates; all the options are captured in the choice production that is associated with the Predicate construct.

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

For instance, predicates can be captured as specimens of UnaryPredicate or BinaryPredicate.

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

The UnaryPredicate construct and the BinaryPredicate construct are introduced in the PQL language to allow checking the unary behavioral primitives and the binary behavioral primitives, respectively. Both these constructs are aggregations of a name (specified by the UnaryPredicateName construct or the BinaryPredicateName construct) and a respective number of Task constructs; one for the UnaryPredicate construct and two for the BinaryPredicate construct.

The PQL language utilizes a well-known mechanism of macros for combining results of several UnaryPredicate or BinaryPredicate checks into a result of a single statement.

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

The aggregate production associated with the UnaryPredicateMacro construct is composed of a reference to a unary behavioral primitive (*name* : UnaryPredicateName), a set of tasks (*tasks* : SetOfTasks), and a quantifier ($q$ : AnyAll). Intuitively, a single macro statement $p \triangleq \texttt{UnaryPredicateMacro}(\textit{name} : n;\ \textit{tasks} : ts;\ q : x)$ is equivalent to a complex 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 $\texttt{UnaryPredicate}(p.\textit{name};\ \textit{task} : t)$ evaluates to *true*. Similarly, one can rely on the

BinaryPredicateMacro construct to combine results of multiple BinaryPredicate checks.

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

The BinaryPredicateMacroTaskSet construct is designed to allow checking whether a certain binary behavioral primitive (*name* : BinaryPredicateName) holds between a given task (*task* : Task) and either at least one (if the AnyAll construct is instantiated with the Any specimen) or every (if the AnyAll construct is instantiated with the All specimen) task in a given set of tasks (*tasks* : SetOfTasks). Similarly, the BinaryPredicateMacroSetSet construct can be used to check whether a binary behavioral primitive of interest evaluates to *true* for certain pairs of tasks in the Cartesian product of two given sets of tasks. Note for the option to use the Each qualifier as a specimen of the AnyEachAll construct in the respective production above. When employed, this option induces a check of whether for every task in one given set of tasks the specified behavioral relation holds with some task from the other given set of tasks.

The PQL language supports checks of basic binary relations between sets of tasks. These are captured by the choice production associated with the SetPredicate construct.

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

The PQL language allows checking if a task is a member of a given set of tasks. This can be accomplished using the TaskInSetOfTasks construct, which is specified as an aggregation of a task (*task* : Task) and a set of tasks (*tasks* : SetOfTasks). Moreover, the PQL language allows checking several binary relations between sets of tasks using the SetComparison construct. The SetComparison construct is composed of two sets of tasks ($tasks_1$ : SetOfTasks and $tasks_2$ : SetOfTasks) and a reference to a comparison operator (*oper* : SetComparisonOperator). The PQL language supports five comparison operations. They specify 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 the other set of tasks.

As PQL is designed to utilize three-valued reasoning, it operates with three truth values: *true*, *false*, and *unknown*. This is reflected in the three literals of the choice production associated with the TruthValue construct, which is proposed below.

$$\texttt{TruthValue} \triangleq \texttt{True} \mid \texttt{False} \mid \texttt{Unknown}$$

To allow complex logical statements on atomic propositions, PQL supports standard logical operations. These are negation (Negation), conjunction (Conjunction), and disjunction (Disjunction).

To permit for a three-valued logic that is used with PQL to be functionally complete, the language includes a test of whether a given three-valued logic value is *unknown*. This check is reflected in the IsUnknown option of the LogicalTest construct proposed below, which for the sake of completeness allows for the total of six different tests of whether a three-valued logic value is or is not equal to a certain truth value.

$$\texttt{LogicalTest} \triangleq \texttt{IsTrue} \mid \texttt{IsNotTrue} \mid \texttt{IsFalse} \mid \texttt{IsNotFalse} \mid \texttt{IsUnknown} \mid \texttt{IsNotUnknown}$$

For a grammar of a language to be complete, all its constructs must be specified in terms of well-defined components, called the *terminal* constructs. The following constructs are the terminal constructs of the PQL grammar: Any, All, Each, Universe, AttributeID, AttributeName, AttributeModel, Identical, Different, OverlapsWith, SubsetOf, ProperSubsetOf, True,

`False`, `Unknown`, as well as all the constructs that are parts of the choice productions associated with the `UnaryPredicateName` and `BinaryPredicateName` constructs. All the above mentioned constructs do not have an internal structure and, thus, are atomic constructs of the PQL language.

Several of the proposed PQL constructs can be defined in terms of well-known sets. For instance, in the discussion above, we hint at the fact that `LocationID` can be specified as an integer. This can be captured rigorously in the production $\texttt{LocationID} \triangleq \mathit{value} : \mathbb{Z}$, where $\mathbb{Z}$ is the symbol often used in mathematics to denote the set of all integers. Similarly, we specify `LocationDirectory`, `VariableName`, and `Similarity`, as $\texttt{LocationDirectory} \triangleq \mathit{value} : \mathbb{S}$, $\texttt{VariableName} \triangleq \mathit{id} : \mathbb{V}$, and $\texttt{Similarity} \triangleq \mathit{value} : [0\,..\,1]$, respectively; here, $\mathbb{S}$ and $\mathbb{V}$ are the set of all character strings and the set of all legal variable names, respectively. Note that set $\mathbb{V}$ is defined in the next section.

Some of the PQL constructs are still not defined in terms of terminal constructs. The PQL language trivially defines the `Negation` construct and all the six options associated with the `LogicalTest` construct in terms of a single `Predicate` component, e.g., $\texttt{Negation} \triangleq \mathit{pred} : \texttt{Predicate}$, $\texttt{IsTrue} \triangleq \mathit{pred} : \texttt{Predicate}$, etc. Finally, for the sake of space considerations, at this stage we omit rigorous definitions of five PQL constructs: `Conjunction`, `Disjunction`, `Union`, `Intersection`, and `Difference`. Intuitively, `Conjunction` and `Disjunction` can be defined as sets of predicates, whereas `Union`, `Intersection`, and `Difference` can be specified as collections of sets of tasks. However, any definition of priorities for the operations that the above stated constructs represent in terms of grammar rules is rather lengthy and is driven by semantic, rather than syntactic, rules. In the next section, we discuss priorities of various operations that are supported in PQL, whereas missing rigorous specifications of the five mentioned constructs can be found in Appendix A.

### 2.2. Concrete Syntax

The abstract syntax of PQL is independent of any particular representation. This section proposes a mapping from the abstract syntax of PQL to its specific encoding. This encoding constitutes one possible concrete syntax of the PQL language, i.e., its machine- and human-readable representation.

The first concrete syntax of the PQL language proposed in this section is inspired by SQL—a programming language for managing data stored in a relational database management system (DBMS) [Date and Darwen 1997]. Being inspired by SQL, we intent to keep the core structure of concrete PQL queries as similar as possible to that of SQL queries and to reuse SQL keywords in PQL, given that the contexts are similar. The reason for this is threefold:

- Despite addressing different domains, i.e., dynamic processes versus static data, both languages serve the same purpose—the purpose of querying for information. Note that SQL was originally proposed to *retrieve* data stored in quasi-relational DBMS [Chamberlin and Boyce 1974].
- SQL is a widely used standard that is supported by just about every DBMS on the market. As a result, its syntax is well-recognized by technical specialists and analysts. By closely following the concrete syntax of SQL, PQL becomes readily usable by a wide range of stakeholders.
- As suggested by several interviewees, it would be beneficial for the syntax of the envisioned query language to resemble that one of SQL. For example, one interviewee commented: "From an overall strategic point of view it'll bring a lot of benefits because different parts of the organization will be able to work together by using some kind of a structured query language (SQL)".

Given a construct of an abstract grammar, one can specify its concrete syntax as a function that yields all its specific forms. In this section, PQL is defined as a textual language. Hence, for each PQL construct, its concrete syntax is given as a function that takes a specimen of the respective abstract construct as input and returns a collection of character strings that are accepted as its concrete encodings. We shall denote such a function by the name of the respective construct with subscript `c`.

For example, the concrete syntax of a specimen of the `Query` construct is defined as follows.

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

We use regular expressions [Aho and Ullman 1992] to define the concrete syntax of PQL specimens. Hence, as per the above definition, 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, i.e., ‘;’. There can be an arbitrary number of whitespace characters between any two subsequent components of a query string. The order of various components is fixed. Note that the presence of the `WHERE` clause in a PQL query is optional, i.e., the `WHERE` keyword and the specification of the predicate can be skipped.

The reader might have already noticed that the core structure of a PQL query is similar to that one of an SQL query that is signified with the declarative `SELECT` statement and is used to formulate an intent for retrieving data from one or more database tables or expressions.

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

$$\texttt{Variables}_\texttt{c}(vs:\texttt{Variables}) \triangleq \mathit{isEmpty}(vs)\ ?\ \text{‘’} : \texttt{Variable}_\texttt{c}(vs.\mathit{FIRST})\ \texttt{Variables}_\texttt{c}(vs.\mathit{TAIL})$$

That is, the encoding of the empty list of variables is the empty string. However, if a list of variables contains at least one element, its encoding is constructed as a concatenation of a concrete form of its first element, denoted by $vs.\mathit{FIRST}$, and an encoding of the list of its all other elements, denoted by $vs.\mathit{TAIL}$. The concrete syntax of a PQL variable is defined below.

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

The concrete syntax of all other specimens of PQL constructs that are associated with list productions is defined similar to that one of the `Variables` construct seen above. However, all these encodings expect to include special symbols between every two subsequent components. 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. Additionally, 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 instance, the character string ‘{"Buy item","Purchase product"}’ is a valid encoding of a specimen of the `SetOfTasksLiteral` construct that contains two elements; here, we follow the standard notation for specifying fixed sets. In the example above, strings `"Buy item"` and `"Purchase product"` are valid encodings of tasks. In general, the concrete encoding of a PQL task is defined as follows.

$$\begin{aligned}\texttt{Task}_\texttt{c}(t:\texttt{Task}) \triangleq\ & \text{‘~’}\ \text{‘"’}\ \texttt{Label}_\texttt{c}(t.\mathit{label})\ \text{‘"’} \\ & \mid \text{‘"’}\ \texttt{Label}_\texttt{c}(t.\mathit{label})\ \text{‘"’}\ (\text{‘[’}\ \texttt{Similarity}_\texttt{c}(t.\mathit{sim})\ \text{‘]’})?\end{aligned}$$

Labels of PQL tasks are always 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 the fact that one is interested in all the tasks of which the label has a degree of similarity to the specified label that is equal or 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.

Every specimen of a construct that is associated with a choice production is a specimen of one of the constructs from the list of alternatives of the choice production. Hence, a concrete syntax of an abstract grammar can (and often does) omit special encodings to signify choice productions, which is the case for the concrete syntax of PQL that is being proposed here. Thus, in the sequel, we only propose concrete encodings for the remaining aggregate productions of the PQL grammar.

A specimen of the `Attribute` construct is a specimen of one out of four terminal constructs. These are `Universe`, `AttributeID`, `AttributeName`, and `AttributeModel`. In PQL, they are denoted by character strings ‘*’, ‘id’, ‘name’, and ‘model’, respectively. Similarly, a specimen of the `Location` construct is a specimen of either `Universe`, or `LocationID`, or `LocationDirectory`.

A specimen of `LocationID` is an integer, whereas a specimen of `LocationDirectory` is a character string; PQL character strings are sequences of characters enclosed in double quotes.

A name $x \in \mathbb{V}$ of a PQL variable, i.e., a concrete encoding of a specimen of the `VariableName` construct, may contain lower case 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; digit at start are not allowed. Subsequent characters may be letters, digits, or underscore symbols.

Next, we propose the concrete syntax for specimens of the `UnaryPredicateConstruction` construct and the `BinaryPredicateConstruction` construct.

$$\texttt{UnaryPredicateConstruction}_\texttt{c}(upc : \texttt{UnaryPredicateConstruction}) \triangleq$$
$$\texttt{`GetTasks'UnaryPredicateName}_\texttt{c}(upc.name)\ \texttt{`('}\ \texttt{SetOfTasks}_\texttt{c}(upc.tasks)\ \texttt{`)'}$$

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

The concrete encodings of specimens of these constructs follow the syntax for specifying function calls that is used in many well-celebrated programming languages, i.e., a name of a function to be called is followed by a comma-separated list of parameters which is enclosed in parentheses. Here, the names of functions are obtained by prefixing `'GetTasks'` to names of unary and binary predicates. The remaining components are specified as parameters of the respective function calls.

PQL exercises similar principles when specifying the concrete syntax of predicates and macros, both for unary and binary cases. The concrete syntax for specifying predicates proceeds as follows.

$$\texttt{UnaryPredicate}_\texttt{c}(up : \texttt{UnaryPredicate}) \triangleq$$
$$\texttt{UnaryPredicateName}_\texttt{c}(up.name)\ \texttt{`('}\ \texttt{Task}_\texttt{c}(up.task)\ \texttt{`)'}$$

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

The only difference is that the names of these imitated function calls are not prefixed, but are solely composed of the concrete encodings of the respective predicate names.

It is proposed that the concrete syntax for denoting the PQL macros overloads the syntax for specifying function calls which encode the PQL predicates, i.e., names of functions and types of outputs are the same, both for a predicate and the respective macro, but types of inputs differ.

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

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

$$\texttt{BinaryPredicateMacroSetSet}_\texttt{c}(bpm : \texttt{BinaryPredicateMacroSetSet}) \triangleq$$
$$\texttt{BinaryPredicateName}_\texttt{c}(bpm.name)$$
$$\texttt{`('}\ \texttt{SetOfTask}_\texttt{c}(bpm.tasks_1)\ \texttt{`,'}\ \texttt{SetOfTask}_\texttt{c}(bpm.tasks_2)\ \texttt{`,'}\ \texttt{AnyEachAll}_\texttt{c}(bpm.q)\ \texttt{`)'}$$

The above proposed syntax rules rely on the concrete encodings of `AnyAll` and `AnyEachAll`, which when instantiated are specified as specimens of `Any`, `Each`, or `All`. These are denoted by the PQL keywords **`ANY`**, **`EACH`**, and **`ALL`**, respectively.

A specimen of the `TaskInSetOfTasks` construct can be typed in as follows.

$$\texttt{TaskInSetOfTasks}_\texttt{c}(in : \texttt{TaskInSetOfTasks}) \triangleq \texttt{Task}_\texttt{c}(in.task)\ \texttt{`IN'}\ \texttt{SetOfTask}_\texttt{c}(in.tasks)$$

That is, every character string that starts with an encoding of a task, which is followed by the PQL keyword `IN`, and ends with an encoding of a set of tasks, specifies a specimen of `TaskInSetOfTasks`.

A specimen of the SetComparison construct can be specified in this concrete syntax of PQL as two encodings of sets of tasks with a representation of a comparison operator in between.

$$\texttt{SetComparison}_{\texttt{c}}(\textit{comp}:\texttt{SetComparison}) \triangleq$$
$$\texttt{SetOfTask}_{\texttt{c}}(\textit{comp.tasks}_1)\ \texttt{SetComparisonOperator}_{\texttt{c}}(\textit{comp.oper})\ \texttt{SetOfTask}_{\texttt{c}}(\textit{comp.tasks}_2)$$

A set comparison operator is instantiated in PQL via a choice between specimens of terminal constructs Identical, Different, OverlapsWith, SubsetOf, and ProperSubsetOf, which are defined in this concrete syntax of PQL via the keywords **EQUALS**, **NOT EQUALS**, **OVERLAPS WITH**, **IS SUBSET OF**, and, **IS PROPER SUBSET OF**, respectively. The specimens of the terminal constructs True, False, and Unknown get encoded as the keywords **TRUE**, **FALSE**, and **UNKNOWN**, respectively.

Predicate names in PQL are encoded as names of the respective terminal constructs, e.g., the unary predicate construct CanOccur and the binary predicate construct TotalCausal have concrete encodings CanOccur and TotalCausal, respectively.

Finally, the concrete encodings of the specimens of the Negation and the six logical test constructs are given below.

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

These encodings rely on the use of the PQL keywords **NOT**, **IS**, **TRUE**, **FALSE**, **UNKNOWN**, and the concrete encoding of the Predicate construct, which is defined by concrete encodings of all the alternatives associated with the corresponding choice production.

We envision introduction of other specific encodings of the abstract syntax of the PQL language, e.g., a visual encoding of PQL queries. We believe that availability of different concrete encodings will make the PQL language accessible to a wider audience.

## A. PQL GRAMMAR

This appendix specifies the complete grammar of the PQL language captured in the ANTLR notation. ANTLR (ANother Tool for Language Recognition) is a parser generator for reading and translating structured text or binary files [Parr and Quong 1995; Parr 2013]. ANTLR can take a grammar of a language as input and generate source code for a parser that can build and walk syntax trees [Meyer 1990]. The language must be specified using a context-free grammar which is expressed using extended Backus-Naur Form [Hopcroft et al. 2006].

```
1  // A PQL v1 grammar for ANTLR v4
2
3  // [The "BSD licence"]
4  // Copyright (c) 2014-2015 Artem Polyvyanyy
5  // All rights reserved.
6
7  // A PQL v1 grammar for ANTLR v4
8
9  grammar PQL;
10
11 query      : variables
12            SELECT attributes
13            FROM locations
14            (WHERE predicate)? EOS ;
15
16 variables  : variable* ;
17 variable   : varName ASSIGN
18              setOfTasks EOS ;
19
20 varName    : VARIABLE_NAME ;
21
22 attributes : attribute (SEP attribute)* ;
23 attribute  : universe
24            | attributeID
25            | attributeName
26            | attributeModel ;
27
28 locations  : location (SEP location)* ;
29 location   : universe
30            | locationID
31            | locationDirectory ;
32
```

```
universe           : UNIVERSE ;
attributeID        : ATTRIBUTE_ID ;
attributeName      : ATTRIBUTE_NAME ;
attributeModel     : ATTRIBUTE_MODEL ;
locationID         : INTEGER ;
locationDirectory  : STRING ;

setOfTasks : tasks
           | union
           | intersection
           | difference ;

tasks      : ( varName | universe )
           | setOfTasksLiteral
           | setOfTasksConstruction
           | setOfTasksParentheses ;

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 ;

anyEachAll  : ANY | EACH | ALL ;

setPredicate     : taskInSetOfTasks
           | setComparison ;

taskInSetOfTasks : task IN setOfTasks ;

setComparison    : setOfTasks
           setComparisonOperator
           setOfTasks ;

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

truthValue : TRUE
           | FALSE
           | UNKNOWN ;

logicalTest: isTrue
           | isNotTrue
           | isFalse
           | isNotFalse
           | isUnknown
           | isNotUnknown ;

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

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

```
159
160 difference   : tasks DIFFERENCE tasks
161              | tasks DIFFERENCE
162                difference ;
163
164 negation     : NOT proposition ;
165
166 isTrue       : proposition IS TRUE ;
167 isNotTrue    : proposition IS NOT TRUE ;
168 isFalse      : proposition IS FALSE ;
169 isNotFalse   : proposition IS NOT FALSE ;
170 isUnknown    : proposition IS UNKNOWN ;
171 isNotUnknown : proposition IS NOT UNKNOWN ;
172
173 disjunction  : (proposition | logicalTest |
174          conjunction) OR (proposition |
175          logicalTest | conjunction) (OR
176          (proposition | logicalTest
177          | conjunction))* ;
178
179 conjunction  : (proposition | logicalTest)
180          AND (proposition | logicalTest)
181          (AND (proposition
182          | logicalTest))* ;
183
184 parentheses  : LP proposition RP
185         | LP conjunction RP
186         | LP disjunction RP
187         | LP logicalTest RP ;
188
189 setOfTasksParentheses : LP varName RP
190         | LP universe RP
191         | LP setOfTasksLiteral RP
192         | LP setOfTasksConstruction RP
193         | LP union RP
194         | LP difference RP
195         | LP intersection RP
196         | LP setOfTasksParentheses RP ;
197
198 UNIVERSE         : '*' ;
199 ATTRIBUTE_ID     : 'id' ;
200 ATTRIBUTE_NAME   : 'name' ;
201 ATTRIBUTE_MODEL  : 'model' ;
202
203 STRING       : DQ ( ESC_SEQ
204              | ~('\\'|'"') )* DQ ;
205 INTEGER      : '0' | '1'..'9' '0'..'9'* ;
206 VARIABLE_NAME: ('a'..'z'|'_')
207                ('a'..'z'|'0'..'9'|'_')*;
208 SIMILARITY   : '1' | '0' ('.' '0'..'9'+)?
209              | '.' '0'..'9'+ ;
210
211 LP           : '(' ;
212 RP           : ')' ;
213 LB           : '{' ;
214 RB           : '}' ;
215 LSB          : '[' ;
216 RSB          : ']' ;
217 DQ           : '"' ;
218 EOS          : ';' ;
219 SEP          : ',' ;
220 ASSIGN       : '=' ;
221 TILDE        : '~' ;
222
223 ESC_SEQ      : '\\' ('\"'|'\\'|'/'|'b'|
224              'f'|'n'|'r'|'t')
225              | UNICODE_ESC ;
226 UNICODE_ESC : '\\' 'u' HEX_DIGIT
227             HEX_DIGIT HEX_DIGIT HEX_DIGIT ;
228 HEX_DIGIT   : ('0'..'9'|
229              'a'..'f'|'A'..'F') ;
230 WS           : [ \r\t\n]+ -> skip ;
231 LINE_COMMENT: '--' ~[\r\n]* -> skip ;
232
233 SELECT       : 'SELECT' ;
234 FROM         : 'FROM' ;
235 WHERE        : 'WHERE' ;
236 EQUALS       : 'EQUALS' ;
237 OVERLAPS     : 'OVERLAPS' ;
238 WITH         : 'WITH' ;
239 SUBSET       : 'SUBSET' ;
240 PROPER       : 'PROPER' ;
241 GET_TASKS    : 'GetTasks' ;
242
243 NOT          : 'NOT' ;
244 AND          : 'AND' ;
245 OR           : 'OR' ;
246
247 ANY          : 'ANY' ;
248 EACH         : 'EACH' ;
249 ALL          : 'ALL' ;
250
251 IN           : 'IN' ;
252 IS           : 'IS' ;
253 OF           : 'OF' ;
254
255 TRUE         : 'TRUE' ;
256 FALSE        : 'FALSE' ;
257 UNKNOWN      : 'UNKNOWN' ;
258
259 identical       : EQUALS ;
260 different       : NOT EQUALS ;
261 overlapsWith    : OVERLAPS WITH ;
262 subsetOf        : IS SUBSET OF ;
263 properSubsetOf  : IS PROPER SUBSET OF ;
264
265 UNION           : 'UNION' ;
266 INTERSECTION    : 'INTERSECT' ;
267 DIFFERENCE      : 'EXCEPT' ;
268
269 CAN_OCCUR       : 'CanOccur' ;
270 ALWAYS_OCCURS   : 'AlwaysOccurs' ;
271 CAN_CONFLICT    : 'CanConflict' ;
272 CAN_COOCCUR     : 'CanCooccur' ;
273 CONFLICT        : 'Conflict' ;
274 COOCCUR         : 'Cooccur' ;
275 TOTAL_CAUSAL    : 'TotalCausal' ;
276 TOTAL_CONCUR    : 'TotalConcurrent' ;
```

## REFERENCES

Alfred Aho and Jeffrey Ullman. 1992. *Foundations of Computer Science*. W.H. Freeman and Company.

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT Press.

D. D. Chamberlin and R. F. Boyce. 1974. SEQUEL: A Structured English Query Language. In *ACM SIGMOD*. 249–264.

C.J. Date and H. Darwen. 1997. *A Guide to the SQL Standard: A User's Guide to the Standard Database Language SQL*. Addison-Wesley.

Javier Esparza and Mogens Nielsen. 1994. Decidability Issues for Petri Nets - A survey. *Bulletin of the EATCS* 52 (1994), 244–262.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Bertrand Meyer. 1990. *Introduction to the Theory of Programming Languages*. Prentice-Hall.

Terence John Parr. 2013. *The Definitive ANTLR 4 Reference*. Pragmatic Programmers, LLC.

Terence John Parr and Russell W. Quong. 1995. ANTLR: A Predicated- *LL(k)* Parser Generator. *Software — Practice and Experience (SPE)* 25, 7 (1995), 789–810.

URI Planning Interest Group. 2001. *URIs, URLs, and URNs: Clarifications and Recommendations 1.0*. Technical Report. W3C. http://www.w3.org/TR/uri-clarification/

W3C XSL/XML Query Working Groups. 2007. The XPath 2.0 Standard. (2007). http://www.network-theory.co.uk/w3c/xpath/

Received February 2007; revised March 2009; accepted June 2009