Introduction to Process Querying

Artem Polyvyanyy

Abstract This chapter gives a brief introduction to the research area of process querying. Concretely, it articulates the motivation and aim of process querying, gives a definition of process querying, presents the core artifacts studied in process querying, and discusses a framework for guiding the design, implementation, and evaluation of methods for process querying.

1 Introduction

A business process is a plan and coordination of activities and resources of an organization that aim to achieve a business objective [22]. Business Process Management (BPM) is an interdisciplinary field that studies concepts and methods that support and improve the way business processes are designed, performed, and analyzed in organizations. BPM enables organizations to systematically control operational processes with the ultimate goal of reducing their costs, execution times, and failure rates through incremental changes and radical innovations [7, 22].

Over the last two decades, many methods, techniques, and tools have been devised to support BPM practices in organizations. Use cases addressed by BPM range from regulatory process compliance, via process standardization and reuse, to variant analysis, process instance migration, and process mining techniques for automated process modeling, enhancement, and conformance checking based on event data generated by IT systems. Despite being devised for different use cases, BPM approaches and tools often rely on similar underlying algorithms, process analysis

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

techniques, and constructed process analytics. For example, process compliance, standardization, reuse, and variant analysis methods rely on algorithms for retrieving processes that describe a case with conditions that capture a compliance violation or a process fragment suitable for standardization, variant identification, or reuse in fresh process designs. Process instance migration and process compliance may further rely on techniques for automatically augmenting processes, such as resolving the issues associated with the non-compliance of the processes or adaptation of a long-running process instance from the old to a new process design.

Process querying aims to identify core algorithms, analysis, and analytics over processes to promote their centralized development and improvement for later reuse when solving practical problems concerned with the management of processes and process-related artifacts, like resources, information, and data. We refer to such core process-related computations as *process querying methods*. Process querying multiplies the effect of process querying methods in different use cases and suppresses reinventions of such methods in different contexts.

The remainder of this chapter is organized as follows. The next section elaborates on the aim of process querying and gives definitions of process querying and a method for process querying. Then, Section 3 presents a framework for devising process querying methods. The framework consists of abstract components, each with well-defined interfaces and functionality, that, when instantiated, result in a concrete method for process querying. The chapter closes with conclusions that also shed light on the directions for future work in process querying.

2 Process Querying

This section discusses the objective and the definition of the research area of process querying and rigorously defines the concept of a process querying method.

2.1 Objective

Process querying aims to support systematic improvement and reuse of methods, techniques, and tools for manipulating and managing models of processes, either already executed or designed existing or envisioned processes, and process-related resources, information, and data. The need for scoping the area of process querying has emerged from numerous observations of non-coordinated efforts for developing approaches for automated management and manipulation of process-related artifacts in the research disciplines of BPM [7, 22] and process mining [1]. To name a few, examples of research problems studied in BPM that fall in the scope of process querying include process compliance, process standardization, process reuse, process migration, process selection, process variance management, process selection, process discovery, process enhancement, and correctness checking [16]. Existing

solutions to these problems often rely on techniques that share algorithmic ideas and principles. Hence, instead of conducting scattered, in silos, studies, with process querying we propose to identify and study such central ideas and principles to improve and reuse them when solving practical process-related problems. Though process querying emerges from the research disciplines of BPM and process mining, we envisage its application in other process-related fields, including software engineering, information systems engineering, computing, programming language theory, and process science.

2.2 Definition

Process querying emerges at the intersection of research areas concerned with modeling, analysis, and improvement of processes. Before giving our definition of process querying, we discuss several such areas and their relation to process querying.

Big data. Big data studies ways to analyze large datasets. Here, we usually speak about datasets that are too large to be analyzed using traditional techniques. Process querying also researches ways to analyze extremely large datasets but is primarily concerned with datasets that comprise event data, e.g., executions of IT systems, records of business processes, user interactions with information systems, and timestamped sensor data. In addition, process querying deals with descriptions of potentially infinite collections of processes.

Process modeling. A process model is a simplified representation of a real world or an envisioned process, or collection of processes, that serves a particular purpose for a target audience. Process modeling is a technique to construct a process model. Process querying studies techniques that support instructions for automated and semi-automated process modeling. Examples of such instructions include removing or inserting parts of a process model to ensure it represents the desired collection of processes for the envisaged purpose and audience.

Process analysis. Process analysis studies ways to derive insights about the quality of processes, including their correctness, validity, and performance characteristics. Process querying relies on existing and studies new process analysis techniques to retrieve existing and model new processes with desired quality profiles. For instance, a process query can specify an intent to retrieve all processes with duration in a given range or augment process designs to ensure their correct future executions under new constraints.

Process analytics. Process analytics studies techniques for computational and statistical analysis of processes and the event data they induce. It is also concerned with identifying meaningful patterns in event data and communication of these patterns. Process querying relies on and extends process analytics to apply it when retrieving and manipulating processes and related artifacts. An example of such synergy between process analytics and querying is an instruction to retrieve and replace process

patterns that lead to negative overall process outcomes with the patterns that were observed to result in positive outcomes.

Process intelligence. Process intelligence studies ways to infer insights from processes and the related resources, information, and data for their subsequent use. Examples of such insights include causes for poorly performing or unsuccessful process executions, while sample uses of the inferred insights include reporting and decision-making.

The definition of process querying is an evolving concept that is continuously refined via an iterative process of embracing and solving practical problems for retrieving and manipulating models of process and process-related artifacts. The current snapshot of this definition is given below:

> *Process querying combines concepts from Big data and process modeling and analysis with process analytics and process intelligence to study methods, techniques, and tools for retrieving and manipulating models of real world and envisioned processes, and the related resources, information, and data, to organize and extract process-related insights for subsequent systematic use.*

Therefore, the idea of process querying is to systematically extract insights from descriptions of processes, e.g., event logs of IT systems or process models, and the associated artifacts, e.g., resources used to support process executions, information capturing the domain knowledge, and data generated during process executions, stored in process repositories of organizations using effective instructions captured in process queries implemented using efficient techniques. Consequently, the task of process querying is to design those effective and efficient process queries over a wide range of inputs capable of delivering useful insights to the users.

2.3 Methods

Processes are properties of dynamic systems, where a *dynamic system* is a system that changes over time, for instance, a process-aware information system or a software system. A *process* is an ordering of events that collectively aim to achieve a goal state, where a *state* is a characteristic of a condition of the system at some point in time. In other words, a state of a process specifies all the information that is relevant to the system at a certain moment in time. An *event* is an instantaneous change of the current state of a system. An event can be distinguished from other events via its attributes and attribute values, for example, a timestamp, event identifier, process instance identifier, or activity that induced the event.

Let $\mathscr{U}_{an}$ be the universe of *attribute names*. We distinguish three special attribute names $time, act, rel \in \mathscr{U}_{an}$, where *time* is the "timestamp", *act* is the "activity", and *rel* is the "relationship" attribute name. Let $\mathscr{U}_{av}$ be the universe of *attribute values*.

Event. An *event* e is a mapping from attribute names to attribute values such that each attribute name that participates in the mapping is related to exactly one attribute value, i.e, $e : \mathscr{U}_{an} \nrightarrow \mathscr{U}_{av}$.

By $\mathscr{E}$, we denote the universe of events. Similar to attribute names, we identify three special classes of events. By $\mathscr{E}_{time}$, we denote the set of all events with timestamps, i.e., $\mathscr{E}_{time} = \{e \in \mathscr{E} \mid time \in dom(e)\}$. By $\mathscr{E}_{act}$, we denote the set of all events with activity names, i.e., $\mathscr{E}_{act} = \{e \in \mathscr{E} \mid act \in dom(e)\}$. Let $\mathscr{U}_{rel} = \mathscr{P}(\mathscr{E}) \times \mathscr{P}(\mathscr{E})$ be the set of all possible pairs of sets of events, where $\mathscr{P}(\mathscr{E})$ is the power set of $\mathscr{E}$, such that $\mathscr{U}_{rel}$ are possible attribute values, i.e., $\mathscr{U}_{rel} \subset \mathscr{U}_{av}$. Then, $\mathscr{E}_{rel}$ is the set of all events that assign a value from $\mathscr{U}_{rel}$ to its relationship attribute, i.e., $\mathscr{E}_{rel} = \{e \in \mathscr{E} \mid rel \in dom(e) \wedge e(rel) \in \mathscr{U}_{rel}\}$.

For example, $e = \{(case, 120328), (time, 2020\text{-}03\text{-}27T03{:}21{:}05Z), (act, \text{“Close claim”})\}$ is an event with three attributes. A possible interpretation of these attributes is that event e belongs to the process with case identifier $e(case) = 120327$, was recorded at timestamp $e(time) =$ 2019-10-22T11:37:21Z (ISO 8601), and was generated by the activity with name $e(act) =$ “Close claim”.

Process. A *process* π is a partially ordered set $(E, \leq)$, where $E \subseteq \mathscr{E}$ is a set of events and $\leq \subseteq E \times E$ is a partial order over E, i.e., $\leq$ is a reflexive, antisymmetric, and transitive binary relation over E.

A process describes that certain pairs of events are ordered. Note that every pair of related, by a process, events specifies that the first event precedes the second event, while for any two unrelated events, their order is unspecified. It is a common practice to interpret two unordered events as such that may be enabled simultaneously or occur in parallel, refer to [15] for details.

A process that is a total order over a set of events is a *trace*.

Trace. A *trace* τ is a process $(E, <)$, where $<$ is a total order over E.

A *behavior* is a collection of processes in which the same process may appear several times to denote the fact that it can be, or was, observed multiple times.

Behavior. A *behavior* b is a multiset of processes.

By $\mathscr{B}$, we denote the universe of behaviors.

Behaviors can be described in conceptual models. According to Lindland et al [13], a conceptual model consists of an explicit model component and an implicit model component. The *explicit component* is the set of all statements explicitly made using some modeling language, whereas the *implicit component* is the set of all statements that can be derived from the explicit component using deduction rules of the modeling language.

We refer to a conceptual model that describes behaviors as a *behavior model*. Let $\mathscr{A} \subset \mathscr{U}_{av}$ be the universe of *activities*. Let $\mathscr{U}_{ms}$ be the universe of explicit *model statements*. Then, $\mathscr{M} = \mathscr{P}(\mathscr{A}) \times \mathscr{P}(\mathscr{U}_{ms})$ is the universe of *activity models*, a special class of explicit components of behavior models, where each activity model is a pair composed of a set of activities and a set of model statements that compose

the activities into the model. By $\odot = (\varnothing, \varnothing)$, $\odot \in \mathscr{M}$, we denote the empty model, the model without activities and statements.

We define four classes of behavior models based on their explicit and implicit components. These four classes are due to the requirements identified in [17].

Behavior model. A *behavior model* is a pair (M, B), where $M \in \mathscr{M}$ is an activity model and $B \subseteq \mathscr{B}$ is a set of behaviors.

- An *event log* is a behavior model $(\odot, \{b\})$, where $b \in \mathscr{B}$ is a finite multiset of finite traces over $\mathscr{E}_{act}$, i.e., the activity model is empty and only one behavior is specified.
- A *simulation model* is a behavior model $(M, \{b\})$, where $M = (A, S) \in \mathscr{M}$ is a nonempty model and $b \in \mathscr{B}$ is a finite multiset of finite processes over $\mathscr{E}_{act}$ such that for every event e in a process in b it holds that $e(act) \in A$.
- A *process model* is a behavior model (M, B), where $M = (A, S) \in \mathscr{M}$ is a nonempty model and every $b \in B$ is a set of processes over $\mathscr{E}_{act} \setminus \mathscr{E}_{time}$ such that for every event e in a process in $b \in B$ it holds that $e(act) \in A$.
- A *correlation model* is a behavior model (M, B), where every $b \in B$ is a multiset of processes over $\mathscr{E}_{rel}$ such that if $M = (A, S)$ is a nonempty model, then for every event e in a process in $b \in B$ it holds that $act \in dom(e)$ and $e(act) \in A$.

Behavior models are immense information resources. A behavior model can characterize a dynamic system by describing potentially an infinite collection of processes it supports and suggesting the ways to lead the system to possible states that are not bounded by any finite collection of states [5].

We say that M and B are, respectively, the *explicit* component and the *implicit* component of the behavior model (M, B). That is, M models behaviors B. We also say that a behavior model $(M, \varnothing)$ is *informal*, i.e., the implicit component of an informal model is empty to indicate that no implicit model statement can be deduced from M. A behavior model $(M, \{b\})$, where $b \in \mathscr{B}$, is *formal*. An explicit component of a formal behavior model induces one behavior, i.e., all the implicit model statements are deduced from M deterministically to define one behavior. Finally, a behavior model (M, B), where $\mid B \mid > 1$, is *semi-formal*, i.e., an explicit component of a semi-formal behavior model can be interpreted as one of the behaviors in B reflecting that the deduction rules of the modeling language used to construct the explicit model are nondeterministic.

An event log is a collection of traces induced by some unknown explicit component of a behavior model; hence, the empty activity model is used as the first element in the pair that defines an event log. Each trace in a log describes some observed process execution. To reflect that the same trace can be observed multiple times, the implicit component of an event log contains a multiset of traces. The multiplicity of a trace in this multiset denotes the number of times this trace was observed. One of the core problems studied in process mining is automatically discovering the explicit component of a behavior model that induced a given log [1]. To support this use case, every event in a trace is required to have the *act* attribute. A simulation model is an activity model together with a finite imitation of its operations in the real world [6]. The implicit component of a simulation model contains a fraction of behavior that

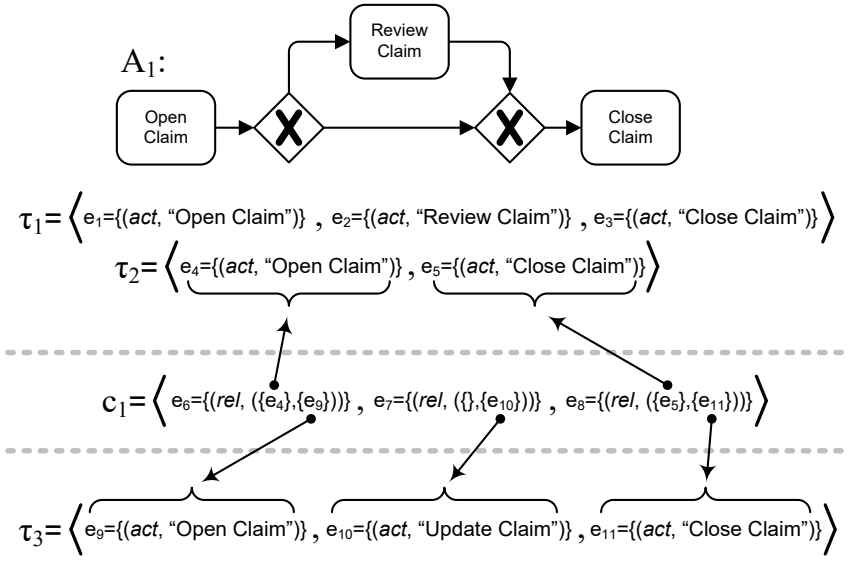

Fig. 1: Example behavior models.

can be induced by the explicit component, which constitutes the behavior imitated during some simulation exercise. To allow traceability, each event of the implicit component has the *act* attribute that refers to the activity that induced the event. A process model is an activity model together with a set of all possible behaviors that can be deduced from the statements the activity model is composed of [3, 19]. Note that a behavior induced by an explicit component of a process model can be infinite, for example, due to some cyclic process dependencies. To reflect the fact that events in the implicit components of process models are envisioned and were not observed in the real world, they do not have timestamps. Finally, a *correlation model* is a behavior model in which every event specifies a relation between two sets of events. Correlation models, for example alignments [2, 12], describe correspondences and discrepancies between the events in two compared processes. Thus, each event in the implicit component of a correlation model uses the *rel* attribute to specify the matching events from two compared processes.

The top of Fig. 1 shows a process model with activity model A_1 as the explicit component, captured in BPMN. According to BPMN semantics, the diagram describes two process instances τ_1 and τ_2, also shown in the figure. Hence, the process model is defined by the pair $(A_1, \{[\tau_1, \tau_2]\})$. The bottom of Fig. 1 shows trace τ_3, which is composed of three events e_9, e_{10}, and e_{11}, that describes a process that starts with activity "Open Claim", followed by activity "Update Claim", and concluded by activity "Close Claim". The pair $L = (\odot, \{[\tau_1^5, \tau_3^2]\})$ specifies a sample event log. Event log L specifies that trace τ_1 was observed five times, while trace τ_3 was observed two times. Note that this log contains traces that cannot be deduced from A_1.

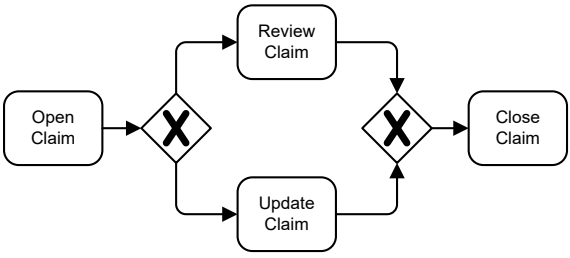

Fig. 2: A process model.

Finally, one can use trace c_1 from Fig. 1 to define a correlation model, for example, $(\odot, \{[c_1]\})$. Trace c_1 relates traces τ_2 and τ_3 and specifies that events e_4 and e_5 in trace τ_2 relate to events e_9 and e_{11} in trace τ_3, respectively, while event e_{10} in trace τ_3 does not correspond to any event in trace τ_2. Hence, c_1 captures minimal discrepancies between traces τ_2 and τ_3 and corresponds to the concept of optimal alignment between these two traces (assuming the use of the standard cost function) [2].

A process repository is an organized collection of behavior models. Let $\mathscr{U}_{re}$ be the set of all *repository elements* that are not behavior models, for example, folders for organizing the models, and names and values of meta-data attributes of the models.

Process repository. A *process repository* is a pair (P, R), where P is a collection of behavior models and $R \subseteq \mathscr{U}_{re}$ is a set of repository elements.

By $\mathscr{U}_{pr}$ and $\mathscr{U}_{pq}$, we denote the universe of *process repositories* and *process queries*, where a process query is an instruction that requests to retrieve artifacts from a process repository or to manipulate a process repository. For example, a process query may capture an instruction to replace a process in one of the behaviors of a process model in a repository with a fresh process. Note that, to ensure consistency between the explicit component and the implicit component, a realization of this query may require updates in the explicit part of the corresponding behavior model.

Finally, a *process querying method* is a computation that given a process repository and a process query systematically performs the query on the repository. The result of performing a query is, again, a process repository that implements the query on the input repository.

Process querying method. A *process querying method* m is a mapping from pairs, where each pair is composed of a process repository and a process query, to process repositories, i.e., it is a function $m : \mathscr{U}_{pr} \times \mathscr{U}_{pq} \to \mathscr{U}_{pr}$.

For example, a process querying method can support a process query that given a process repository that contains the process model captured in Fig. 1 and updates it to describe trace τ_3 instead of trace τ_2 to result in a process repository with a fresh model shown in Fig. 2, which represents all the traces in log L discussed above.

3 Process Querying Framework

In [17], we proposed the *Process Querying Framework* (PQF) for devising process querying methods. Schematic visualization of the framework is shown in Fig. 3.[1] We present the framework in Section 3.1. Then, Section 3.2 discusses decisions that one must take when designing a new process querying method. Finally, Section 3.3 elaborates on the challenges associated with the design decisions and how every process querying method is a compromise of the decisions taken.

3.1 Framework

The PQF is an abstract system of components that provide generic functionality and can be selectively replaced to result in a new process querying method. In Fig. 3, rectangles and ovals denote *active components* and *passive components*, respectively; note the use of an ad-hoc notation. An active component represents an *action* performed by the process querying method. In turn, a passive component is a (collection of) *data objects*. Passive components serve as inputs and outputs of actions. To show that a passive component is an input to an action, an arc is drawn to point from the component to the action, while an arc that points from an action to a passive component shows that the action produces the component. The dashed lines encode the aggregation relationships. A component that is used as an input to an action contains an adjacent component. For example, a *process repository* is an aggregation of event logs, process models, correlation models, or simulation models, refer to the figure.

The framework is composed of four parts. These parts are responsible for designing process repositories and process queries ("*Model, Simulate, Record and Correlate*"), preparing process queries ("*Prepare*"), executing process queries ("*Execute*"), and interpreting results of the process querying methods ("*Interpret*"). In the figure, the parts are enclosed in areas denoted by the dotted borders. Next, we detail the role of each of these parts.

Model, Record, Simulate, and Correlate. This part of the PQF is responsible for modeling or creating behavior models and process queries. Behavior models can be acquired in several ways. For example, they can be designed manually by an analyst or constructed automatically using process mining [1] or process querying, as a result of executing a query. Alternatively, an event log can be obtained by recording the traces of an executing IT system. Finally, a behavioral model can be a result of correlating steps of two different processes. Examples of behavior models include process models like computer programs, business process specifications captured

[1] © 2017 Elsevier B.V, Fig. 3 is reprinted from Decis. Support Syst. 100, Artem Polyvyanyy, Chun Ouyang, Alistair Barros, Wil M. P. van der Aalst, Process querying: Enabling business intelligence through query-based process analytics, pp 41–56, with permissions from Elsevier.

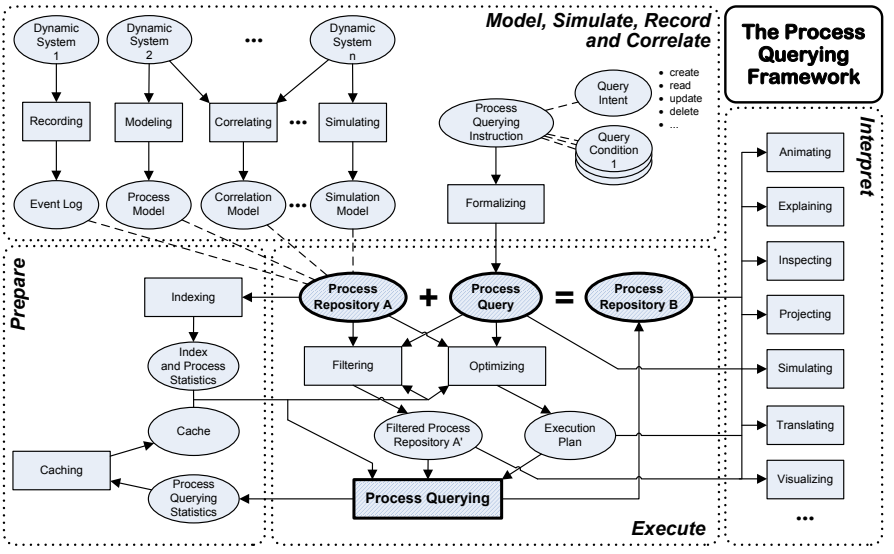

Fig. 3: A schematic view of the Process Querying Framework [17].

in BPMN, YAWL, and BPEL notation, and formal automata and Petri net models, event logs of IT systems [1], and correlation models like alignments [2, 12].

A *process querying instruction* specifies which processes, behaviors, or behavior models should be added to, removed from, replaced in, or retrieved from which processes, behaviors, or behavior models in a process repository. Such an instruction is composed of a *query intent* that specifies the aim of the instruction and a list of *query conditions* that parameterize the intent. The resulting process querying instruction should unambiguously specify how to execute it over the process repository. For example, an instruction can specify to retrieve, or *read*, processes from the repository, while its conditions detail which processes should be included in the query result and which should be left out. We refer to an instruction captured in some machine-readable, i.e., formal, language as a *process query*. The *Formalizing* active component of the framework is responsible for translating process querying instructions into process queries expressed in domain-specific programming languages or some other formalisms.

Prepare. The *Prepare* part of the framework is responsible for making process repositories ready for efficient querying. In its current version, the framework suggests two methods for preparing for querying, namely indexing and caching. In databases, *indexing* is a technique to construct a data structure, called an index, to efficiently retrieve data records based on some attributes. In computing, *caching* is a technique to store data in a cache so that future requests to that data can be served faster, where the stored data might be the result of an earlier computation.

An index is usually constructed offline and uses additional storage space to maintain an extra copy, or several copies, of the process repository. It is expected that

this additional storage will be used to speed up the execution of queries. A process querying can also collect statistics over properties of the repository and its indexes, referred to as *process statistics* in the figure. Process statistics should be used to guide the execution of process queries. For example, a process querying method can proceed by first executing queries over simple models to ensure that initial results are obtained early and proposed to the user.

Caching in process querying can rely on *process querying statistics* to decide which (parts of) query results should be stored for later prompt reuse. The statistics may include aggregate information on the execution of process queries and evaluation of process query conditions, e.g., frequencies of such executions and evaluations. The results of the most frequent queries and evaluated query conditions can then be put in the cache. Next time the user requests to evaluate a query or a condition of a query stored in the cache, its result can be retrieved from the cache instead of recomputed, which is usually more efficient. Caching decisions can rely on *process querying statistics* that aims to keep track of recent frequent query executions and query condition evaluations.

One can rely on other approaches to speed up the evaluation of process queries. The standard approaches that can be explored include parallel computing, e.g., map-reduce, algorithm redesign, e.g., stochastic and dynamic optimization, and hardware acceleration, e.g., in-memory databases and computing on graphics processing units. Note, however, that such optimization approaches are often inherent to the designs of techniques they optimize. Even though such optimizations are clearly useful, we request that future approaches impose as few restrictions as possible on the querying methods they are intended to be used with.

Execute. The *Execute* part of the framework is responsible for executing process queries over repositories. It comprises components for filtering process repositories, and optimizing and executing process queries.

Filtering is used before executing a query to tag those processes, behaviors, or models in the repository that are known to be irrelevant for the purpose of the process query. Then, the query execution routine can skip tagged artifacts to improve the efficiency of query processing. For instance, if a query requests to select all process models that describe a process with an event that refers to a given activity, it makes no sense to execute the query over models that do not contain the given activity. The filtering is performed by the eponymous active component of the framework that takes the repository and a query as input and produces a *filtered process repository* as output. A filtered repository is a projection of the original repository with some of its parts tagged as irrelevant for the purpose of the query processing. Clearly, to be considered useful, a concrete *Filtering* component must perform filtering decisions more efficiently than executing the query over the same parts of the repository.

The component responsible for query optimization, see the *Optimizing* component in the figure, takes as input a query and all the information that can help produce an efficient execution plan for the query. An *execution plan* is a list of commands that should be carried out to execute the query using the least amount of resources possible. Two types of optimizations are usually distinguished: logical and physical. A *logical* optimization entails reformulating a given query into an equivalent but

easier—which usually means a faster to execute—query. A *physical* optimization, in turn, consists of determining efficient means for carrying out commands in a given execution plan.

Finally, the *Process Querying* component takes as input an execution plan of a query, a corresponding filtered repository, as well as available index, process statistics, and cache, and produces a new process repository that implements the instruction specified by the query. As a side effect of executing a query, the component updates the *process querying statistics*. The filtered repository and the execution plan are the *critical inputs* of the querying component, as without these inputs the querying cannot take place, while all the other inputs are optional or can be empty.

Interpret. An outcome of a process query execution falls into two broad categories: successful or unsuccessful. The successful outcome signifies that the querying instruction captured in the query was successfully implemented in the repository. The latter situation may, for instance, arise when managing vast (possibly infinite) collections of processes described in a process model using scarce (finite) resources of a computer that processes the query.

When a process query fails to execute because of resource limitations, one can adopt at least two strategies to obtain a partial or the full query result. It may be possible to reformulate the original query to give up on the precision of its results. Alternatively, one may be able to optimize the querying method to allow handling the special case of the original query or the class of queries the original query falls into. The standard approaches for managing vast collections of processes include symbolic techniques, such as binary decision diagrams, and abstractions based on the structural model or behavior regularities.

It is often desirable to communicate the query results, successful or unsuccessful, to the user who issued the query. The *Interpret* part of the framework serves this purpose. All the active components of this part have a common goal: to contribute to the user's better comprehension of the querying results. The components listed in Fig. 3 are inspired by the various means for improving comprehension of conceptual models proposed by Lindland et al. [13]. As input, these components take the (filtered) process repository, the query and its execution plan, and the resulting process repository and aim to explain all the differences between the original and the resulting repository, and the reasons for the differences.

One can use several techniques to foster the understanding of process querying results. First, a user can understand a concept or phenomenon by inspecting, or reading, it. One can explore various approaches to facilitate the process of reading process query results. For instance, important aspects can be emphasized, while the secondary aspects downplayed. Besides, the inspection activities can be supported by a catalog of predefined explanation notes prepared by process analysts and domain experts. Second, by presenting process querying results diagrammatically rather than in text, their comprehension can be improved. Third, the visual representations of process query results can be further animated, e.g., to demonstrate the dynamics of the processes that comprise the query result. A common approach to animating the dynamics of a process is through a token game that demonstrates the process state evolution superposed over the diagrammatic process model. Fourth, the comprehen-

sion of the process querying results can be stimulated by projecting away their parts, allowing the user to focus on a limited number of aspects at a time. Fifth, one can simulate and demonstrate to the user the processes that constitute the query result captured in a static model. Finally, process querying results can be translated into notations that the user is more familiar with. The implementation of these practices is the task of the corresponding active components of the framework.

3.2 Design Decisions

A design decision is an explicit argumentation for the reasons behind a decision made when designing a system or artifact. When instantiating the PQF to define a concrete process querying method, one needs to take several design decisions imposed by the framework. Next, we discuss three decisions one needs to take when configuring the framework, namely which behavior models, which model semantics that induce processes, and which process queries should the process querying method support.

Which behavior models to consider? An author of a process querying method must decide which behavior models the method will support. Note that a method that addresses querying of event logs will most likely be composed of active and passive components of the PQF that are different from a method for querying correlation models. Besides, the choice of supported formalisms for capturing behavior models restricts the class of supported processes, or languages in the terminology of the theory of computation [20], supported by the process querying method. For instance, if behavior models are restricted to deterministic finite automata, the class of processes described by the models is limited to the class of regular languages [20, 21].

Which processes to consider? A behavior model can be interpreted as such that describes several behaviors, each induced by a different model semantics criterion. The choice of a semantics criterion determines the correspondence between the model and the collection of processes associated with this model. For instance, a process model can be interpreted according to the finite, infinite, or a fair process semantics. According to the finite process semantics, a model describes processes that lead to terminal goal states. In contrast, an infinite process semantics accommodates processes that never terminate, i.e., processes in which after every event there exists some next event that gets performed. A process in which, from some state onward, an event can get enabled for execution over and over again but never gets executed is an *unfair* process. A not unfair process, as per the stated principle, is a *strongly fair* process [11]. A fair process can be finite or infinite. There are different fairness criteria for processes. Several of them, including the strong fairness criterion, are discussed in [4]. The choice of the correspondence between models and collections of processes they are associated with defines the problem space of the process querying method, as it identifies the processes to consider when executing queries.

Which process queries to consider? An author of a process querying method must decide which queries the method will support. The design of a process query consists

of two sub-tasks of choosing the *intent* of the query and, subsequently, fixing its *conditions*. The choice of supported process queries determines the expressiveness of the corresponding process querying method, i.e., it defines the ability of the method to describe and solve various problems for managing process repositories. For example, a process querying method can support queries with the intent to *read* process-related information from the repository, i.e., to *retrieve* processes for which specific conditions hold. Alternatively, one can envision a process querying method that supports queries with intents that address all the CRUD operations over models, behaviors, or processes. To specify process queries formally, one can provide formal descriptions of their abstract syntax, concrete syntax, and notation [14].

3.3 Challenges and Compromise

The design decisions taken when instantiating the PQF into a concrete process querying method inevitably lead to challenges associated with their realization. Next, in Section 3.3.1, we discuss three challenges associated with the design decisions discussed in Section 3.2. After discussing the challenges, in Section 3.3.2, we conclude that every process querying method is a *compromise* between specific solutions taken to address the challenges.

3.3.1 Challenges

When implementing a process querying method, one inevitably faces three challenges: decidability, efficiency, and usefulness of the supported process queries.

Decidability. Process queries must be decidable. In other words, they must be solvable by algorithms on a wide range of inputs. Indeed, a process query that cannot be computed is of no help to the user. The decidability requirement poses a significant challenge, as certain process management problems are known to be undecidable over specific classes of inputs. For example, process queries can be expressed in terms of temporal logic formulas [5, 18]. However, temporal logic formulas can be undecidable over some classes of process models [9, 10].

Efficiency. Process querying aims to provide valuable insights into processes managed by organizations. As part of this premise, process querying should foster the learning of processes, behaviors, and behavior models contained in the repository by the novice users of the repository. In other words, it should support exploratory querying [23]. However, exploratory querying requires techniques capable of executing queries close to real-time. Therefore, another challenge of process querying is to develop process queries that can be computed efficiently, that is, fast and using small memory footprints. One can measure the efficiency of the process queries using well-known techniques in computational complexity theory, which study resources,

like computation time and storage space, required to solve computational problems with the goal of proposing solutions to the problems that use less resources.

Suitability. Process querying methods should offer a great variety of concepts and principles to capture and exercise in the context of process querying. Thus, the third challenge of process querying is concerned with achieving expressiveness in terms of capturing all the suitable (appropriate for the purpose envisioned by the users) process queries that specify instructions for managing process repositories. Authors of process querying methods should strive to propose designs that support all the useful (as perceived by the users) process queries. The suitability of process querying methods can be assessed empirically or, similar to [8], by identifying common reoccurring patterns in specifications of process querying problems.

3.3.2 Compromise

A process querying method is identified by a collection of process queries it supports. The selection of queries to support is driven by the considerations of decidability, efficiency, and suitability of queries. As these considerations often forbid the method to support all the desired queries, we refer to the phase of selecting which queries to support and not support as the *process querying compromise*.

The process querying compromise can be formalized as follows. Let D be the set of all decidable process queries. Some decidable queries can be computed efficiently; note that, in general, the decidability of certain process queries can be unknown. Let $E \subseteq D$ be the set of all process queries that are not only computable but are also efficiently computable. Finally, let S be the set of all process queries that the users perceive as suitable. Then, the queries in $E \cap S$ are the queries that one should aim to support via process querying methods. Fig. 4 demonstrates the relations between sets D, E, and S visually. In an ideal situation, it should hold that $S \subseteq E$, i.e., all the suitable queries are computable using some efficient methods. However, in practice, it is often impossible to fulfill the requirement of $S \subseteq E$, or even $S \subseteq D$. Then, one can strive to improve the efficiency of the techniques for computing queries in $(S \cap D) \setminus E$, which are the decidable and practically relevant queries for which no efficient computation procedure is known.

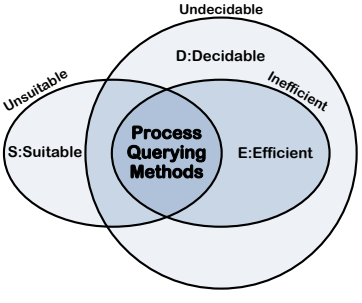

Fig. 4: Process querying compromise.

The existence of such compromise differentiates process querying from data querying. Note that data queries usually operate over finite datasets, making it possible to implement querying using, maybe not always efficient, but certainly effective methods.

4 Conclusion

This chapter presents and discusses the problem of process querying. Process querying aims to coordinate the efforts invested in the design, implementation, and application of techniques, methods, and tools for retrieving and manipulating models of processes, and the related resources, information, and data. Consequently, process querying supports centralized activities that improve process querying practices and suppresses reinventions of such practices in different contexts and variations. The chapter also presents an abstract framework for designing and implementing process querying methods. The framework consists of abstract components, each with a dedicated role and well-defined interface, which, when instantiated and integrated, result in a concrete process querying method. Finally, the chapter argues that every process querying method defines a compromise between efficiently decidable and practically relevant queries, which is unavoidably associated with challenges for designing and implementing such methods.

The concept of process querying emerged from the observations of theory and practice in the research discipline of BPM, and relates to other process-centric research fields like software engineering, information systems engineering, and computing. We envisage future applications, adaptations, and improvements of process querying techniques contributed from within these fields. Future endeavors in process querying will contribute to understanding the process querying compromise, including which queries are practically relevant for the users to justify the efforts for their design and use in practice.

Acknowledgements Artem Polyvyanyy wants to thank Chun Ouyang, Alistair Barros, and Wil van der Aalst with whom he worked together to shape the concept of process querying and to design and validate the Process Querying Framework.

References

1. van der Aalst, W.M.P.: Process Mining—Data Science in Action, 2nd edn. Springer (2016)
2. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying history on process models for conformance checking and performance analysis. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery **2**(2), 182–192 (2012). DOI 10.1002/widm.1045
3. van der Aalst, W.M.P., Stahl, C.: Modeling Business Processes—A Petri Net-Oriented Approach. MIT Press (2011)
4. Apt, K.R., Francez, N., Katz, S.: Appraising fairness in languages for distributed programming. Distrib. Comput. **2**(4), 226–241 (1988)
5. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
6. Banks, J., II, J.S.C., Nelson, B.L., Nicol, D.M.: Discrete-Event System Simulation, 5th edn. Pearson Education (2010)
7. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management, Second Edition. Springer (2018)
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE, pp. 411–420. ACM (1999)

9. Esparza, J.: Decidability and complexity of Petri net problems—an introduction. In: Petri Nets, *LNCS*, vol. 1491, pp. 374–428. Springer (1996)
10. Esparza, J., Nielsen, M.: Decidability issues for Petri nets—a survey. Bulletin of the EATCS **52**, 244–262 (1994)
11. Kindler, E., van der Aalst, W.M.P.: Liveness, fairness, and recurrence in Petri nets. Inf. Process. Lett. **70**(6), 269–274 (1999)
12. Leemans, S.J., van der Aalst, W.M., Brockhoff, T., Polyvyanyy, A.: Stochastic process mining: Earth movers' stochastic conformance. Information Systems p. 101724 (2021). DOI 10.1016/j.is.2021.101724
13. Lindland, O.I., Sindre, G., Sølvberg, A.: Understanding quality in conceptual modeling. IEEE Softw. **11**(2), 42–49 (1994)
14. Meyer, B.: Introduction to the Theory of Programming Languages. Prentice-Hall (1990)
15. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, Part I. Theor. Comput. Sci. **13**, 85–108 (1981)
16. Polyvyanyy, A.: Business process querying. In: Encyclopedia of Big Data Technologies. Springer (2019)
17. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. Decision Support Systems **100**, 41–56 (2017). DOI 10.1016/j.dss.2017.04.011
18. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems, chap. Business Process Compliance, pp. 297–317. Springer (2012)
19. Reisig, W.: Understanding Petri Nets—Modeling Techniques, Analysis Methods, Case Studies. Springer (2013)
20. Sipser, M.: Introduction to the Theory of Computation, 3rd edn. Cengage Learning (2012)
21. Valk, R., Vidal-Naquet, G.: Petri nets and regular languages. J. Comput. Syst. Sci. **23**(3), 299–325 (1981)
22. Weske, M.: Business Process Management - Concepts, Languages, Architectures, Third Edition. Springer (2019)
23. White, R.W., Roth, R.A.: Exploratory Search: Beyond the Query-Response Paradigm. Morgan & Claypool Publishers (2009)