Querying Structural and Behavioral Properties of Business Processes ⋆

Daniel Deutch ⋆⋆ and Tova Milo

School of Computer Science, Tel Aviv University, {danielde,milo}@post.tau.ac.il

Abstract. BPQL is a novel query language for querying business process specifications, introduced recently in [5, 6]. It is based on an intuitive model of business processes as rewriting systems, an abstraction of the emerging BPEL (Business Process Execution Language) standard [7]. BPQL allows users to query business processes visually, in a manner very analogous to the language used to specify the processes. The goal of the present paper is to study the formal model underlying BPQL and investigate its properties as well as the complexity of query evaluation. We also study its relationship to previously suggested formalisms for process modeling and querying. In particular we propose a query evaluation algorithm of polynomial data complexity that can be applied uniformly to queries on the *structure* of the process specification as well as on the potential *behavior* of the defined process. We show that unless P=NP the efficiency of our algorithm is asymptotically optimal.

1 Introduction

A Business Process (BP for short) consists of a group of business activities undertaken by one or more organizations in pursuit of some goal. It usually depends upon various business functions for support (e.g. personnel, accounting, inventory), and interacts with other BPs/activities carried out by the same or other organizations. Consequently, the implementations of such BPs typically operate in a cross-organization, distributed environment.

It is a common practice to use XML for data exchange between BPs, and *Web Services* for interaction with remote processes [26]. Complementarily, the recent BPEL standard (Business Process Execution Language [7]) allows description not only of the interface between the participants in a process, but also of the *full operational logic* of the process and its *execution flow*.

Since BPEL has a fairly complex syntax, commercial vendors offer systems that allow design of BPEL specifications via a visual interface. These systems use a conceptual, intuitive representation of the process, as a graph of activity nodes, connected by control and data flow edges. The designs are automatically converted to BPEL specifications, which in turn can be automatically compiled into executable code implementing the BP [23].

Already in 2002, the importance of query languages for business processes had been recognized by BPMI (the Business Process Management Initiative) [8], yet no draft standard has been published since.

⋆ The research has been partially supported by the European Project EDOS and the Israel Science Foundation

⋆⋆ Supported by the Deutsch Institute

To answer this need, we have recently developed BPQL, a novel query language for querying business process specifications [5, 6]. BPQL is based on the same graph-based view of processes, used by vendors for the specification of BPs. It allows users to query BPs visually, in an intuitive manner, very analogous to how such processes are typically specified. In this paper we present a thorough study of the formal model underlying BPQL, suggest a generic algorithm for query evaluation on BPs, and analyze its complexity and relationship with common formalisms for processes modeling and querying.

Next, we give the intuition behind our formalisms. The exact definitions are given in the next section.

Data Model Intuitively, we model the specification of a BP system as a set of directed, possibly recursive *nested graphs*, including a unique *root* graph that serves as the entry point for the specification. Each graph has a single 'start' and 'end' nodes and represents the execution flow of some function (i.e. a process). A graph may contain (possibly recursive) calls to other functions (processes), which in turn are also represented by flow graphs. Upon an invocation of a call to a function f, appearing in the graph of a function g, the graph of (the implementation of) f is 'plugged-in' into g's graph, replacing the node that represents the call to f. Each graph obtained from g's graph by a sequence of such replacements is called a *refinement* of g.

Query language At the core of the BPQL language are *BP patterns*, which generalize the tree patterns of XML to nested BP graphs and enable users to describe the patterns of activities/data flow that are of interest. In particular, the patterns allow navigation along two axes: (1) the standard path-based axis, which navigates paths in process graphs, and (2) a novel zoom-in axis, that allows to navigate (transitively) inside the process functions, at any depth of nesting, and query their refinements. Many data models which are all equivalent[1] to this simple model of nested graphs appear in the literature. Among them one can find restricted versions of Rewriting Systems (e.g. [25]), Recursive State Machines (RSMs) [4], Context Free Graph Grammars [14], and others. Each of these works relates to some query language which is evaluated over the data model. We identify two main branches of query languages, as follows. In the Databases area, the query languages are *structural*. Namely, they allow users to ask questions about the structure of a specification (graph). In contrast, in the Verification area, the query languages are *behavioral*. These queries relate to the possible runs of the process defined by specification, and are used to identify invariants, execution patterns, etc. The models considered for the structural Database queries are typically 'flat' graph models, without nesting. Verification-related works query include both flat and nested graph models.

While our model for BP specifications is quite standard, we emphasize the uniqueness of our *query language* with respect to common query languages (see Section 3 for an overview). The main features of the query language are given next.

[1] The definition of *models equivalence* is given in section 3

1. BPQL is a unified environment for querying structural as well as behavioral properties of business processes. Specifically, this work is the first to suggest a query language for structural queries over nested graphs.
2. BPQL allows queries with flexible granularity. Users can ask *coarse-grained* queries that consider certain process components as black boxes and allow a high level abstraction, as well as *fine-grained* queries that "zoom-in" on all (or some of) the process components, possibly recursively.
3. BPQL is a graphical query language, with the query being similar to the specification, thus allowing intuitive formulation of the queries parallel to the specification development.

BPQL enables a flexible and intuitive formulation of queries on BPs. We will see, however, that this makes the evaluation of queries somewhat intricate. First, the nested shape of the BP graphs/patterns causes the query evaluation to be computationally more expensive than that of similar queries on flat graphs. Indeed, we show that while the data complexity of BPQL queries is polynomial, the combined complexity is NP-complete w.r.t. the size of the query, even for simple classes of queries that can be evaluated on flat graphs in polynomial time (combined complexity). Second, the need to support both structural and behavioral interpretations for BP patterns required the design of a query evaluation algorithm which can be parameterized by the desired semantics. We propose here such a query evaluation algorithm and show that, unless $P = NP$, a more efficient algorithm does not exist. Moreover, thanks to the modular nature of our algorithm, the complexity of query evaluation over nested BPs is parameterized by the complexity of query evaluation for flat graphs. This allows identification of restricted classes of queries and specifications for which the performance can be further improved.

The BPQL query language was originally introduced in [5] where a first prototype of the BPQL system was demonstrated. There, and in a follow-up work [6] the focus was on the graphical query interface and the system implementation. The model that had been considered was limited to structural queries. The formalization presented here is new, and so are the results. Since BPs in general, and BPEL specifications in particular, are promised such a brilliant future, we believe it is very important to develop a formal foundation for modeling and querying such specifications, so that this technology can be better understood and used. Querying the behavior of a system is essentially a *verification problem* [12] and is typically of very high complexity (from NP-hard for very simple specifications to undecidable in the general case [12]). To guaranty a complexity that is polynomial in the size of the data, BPQL ignores the run-time semantics of certain BPEL constructs such as conditional execution and variable values, and focuses on the given specification flow. We believe that this approach offers a reasonable balance between expressibility and complexity. Clearly, the general problem is more complex, and further work is needed.

The paper is organized as follows. Section 2 describes the BPQL data model and query language and its semantics. Section 3 compares BPQL to related models. Section 4 describes the query evaluation algorithm and Section 5 studies its complexity. We conclude in Section 6.

2 Preliminaries

In this section we present the formal model underlying BPQL. We start with the motivation for our work, and then proceed to the formal definitions.

2.1 Motivation

The following questions may rise from the introduction: Why are structural queries over nested graphs interesting? What are the advantages of a generic framework for multiple query semantics? Why is it important to have a graphical query language, similar to the specification? We give here intuitive answers to these questions, using some examples.

Figure 1 depicts a partial specification of a travel agency system. The rectangle-shaped nodes represent function calls. $BP1$ is the root BP and contains a single node, AlphaTours, that serves as an entry point for the travel agency. $BP2$ describes the implementation of the AlphaTours function, where a user can choose between searching for a trip and reserving one. $BP3$ is the implementation of the $SearchTrip$ function used in $BP2$. A user can request for a specific search (for flights, cars, etc.) or can go back to the AlphaTours trip reservation process. Note that this definition establishes recursive dependencies between the processes, as $BP2$ may call $BP3$, which in turn, if the user decides to reset (implemented in the BP as a call to AlphaTours), calls $BP2$.

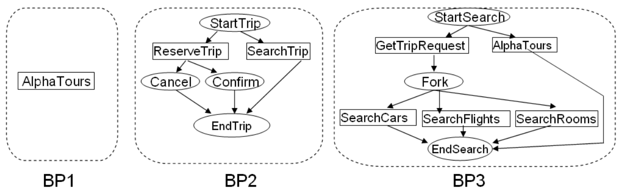

Fig. 1. A BPQL Specification.

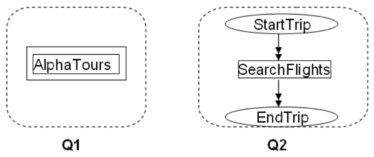

Fig. 2. A BPQL Query.

An example query is depicted in Figure 2. It is formulated graphically in a manner very similar to the specification. This is an important feature of the query language, as (a) it allows faster learning curve of the language and (b) it allows simultaneous formulation, by the specification designer, of a specification and verification queries over it.

To answer a query, we seek for occurrences of the described patterns within the specification. Intuitively, the query in Figure 2 searches the AlphaTours BP, and the processes that it uses, for execution paths leading to/from a SearchFlights operation. $Q2$ here describes an *implementation pattern* for the AlphaTours function. The double-headed arrows indicate that we are looking for execution paths of arbitrary length. The double bounding of the AlphaTours rectangle denotes an *unbounded zoom-in*; we search for the $Q2$ pattern inside the implementation of AlphaTours and (recursively) the functions that it invokes. In general, when matching a (double-bounded) function node n of the query to a function node n' in the specification, we require that the implementation pattern of n, as given in the query, is matched to (a refinement of) the implementation of n' in the specification. Such matching is called an *embedding*.

Some variants of the answer to a query are suggested. The first distinction is between *boolean* and *explanatory* answers. The former answers whether or not some embedding exists, while the latter is a new BP, consisting of the specification parts that contributed to some possible embedding. To continue with our example, the explanatory answer for the query in Figure 2 when applied on the system in Figure 1 is depicted in Figure 3. The answer here is a 'projection' of the travel agency system over the parts relevant to the query, and so it contains the SearchTrip function in $BP2$ and the path in its implementation, $BP3$, that leads to SearchFlights. It also contains the AlphaTours function call node in $BP3$, as this call allows to invoke $BP2$ and recursively reach (by calling SearchTrip) $BP3$ and SearchFlights, via another execution path (in fact, an infinite number of such recursive calls, hence paths, are possible).

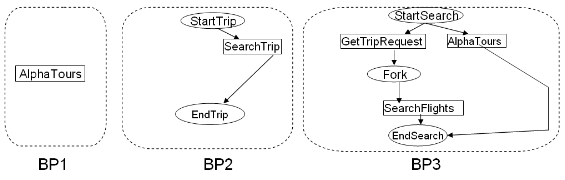

Fig. 3. Explanatory Query Answer.

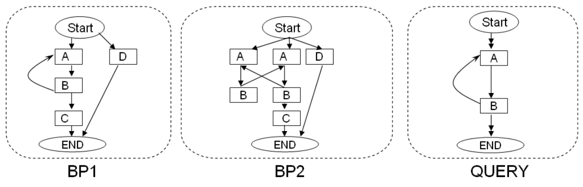

Fig. 4. Structural vs. Behavioral

Another distinction concerns the type of embedding (of the query in the specification) sought for. We look at two common approaches for such embeddings, referred to as *structural* and *behavioral*. Consider the query (BP pattern) depicted in Figure 4. Interpreted as a query over the *structure* of a process specification, this query searches for BPs whose "code" contains a loop of the shape depicted by the query. BP1 in Figure 4 is an example for such BP. The same query, interpreted as a query over the *behavior* of the BPs, will look for processes containing execution paths of form similar to the one specified in the query, namely an unbounded sequence of A,B's. This is satisfied by both BP1 and BP2. The key point is that here, unlike the structural interpretation, the use of distinct occurrences of A and B is allowed.

In previous query languages for querying process specifications, typically only the behavioral approach was taken, with modal (and specifically temporal) logics being used as the basis for the query language. The dichotomy between the two approaches is established by the fact that subgraph isomorphism/homomorphism cannot be expressed by any bisimulation-invariant language [12], and thus, in particular, by any temporal logic (as these are bisimulation-invariant [12]). Thus, structural queries cannot be formulated using the previous works, but are still of great interest, as explained next. Continuing with the example above, code reuse is a common programming policy. This policy would probably impose loops of the structure depicted in BP1 rather than the structure in BP2. The query in Figure 4, when interpreted as structural query, enforces this policy, in a manner not possible using behavioral queries. In general, structural queries are of high importance for any purpose that is interested also in the code itself, and not only in its executions. Such purposes may include coding conventions, profiling and optimizations.

2.2 Definitions

We now give the formal definitions of the specification and query languages. To simplify the presentation we first consider a basic data model and query language, and then enrich them to obtain the full fledged model.

BPs and BP systems We assume the existence of two infinite domains: a domain $\mathcal{N}$ of nodes and a domain $\mathcal{L}$ of node labels, containing a sub-domain $\mathcal{F}$ of function names. We model a BP as a directed labeled graph. Formally,

Definition 1. *A* business process *(BP) is a quadruple* $p = (G, \lambda, \texttt{start}, \texttt{end})$*, where* $G = (N, E)$ *is a connected directed graph in which* $N \subset \mathcal{N}$ *is a finite set of nodes,* E *is a set of edges with endpoints in* N*;* $\lambda : N \to \mathcal{L}$ *is a labeling function for the nodes;* $\texttt{start}, \texttt{end}$ *are two distinguished nodes in* G *and every node in* G *resides on a path from* $\texttt{start}$ *to* $\texttt{end}$*. Nodes labeled by function names from* $\mathcal{F}$ *are called* function calls.

A *system* is a collection of BPs, and a mapping of function names to implementations.

Definition 2. *A* system s *of BPs is a triple* (S, s_0, τ)*, where* S *is a finite set of BPs,* $s_0 \in S$ *is a distinguished BP, called the* root *process, and* $\tau : \mathcal{F} \to 2^S$ *is a (possibly partial) function, called the* implementation function*, mapping function names in* S *to sets of BPs in* S*.*

W.l.o.g we assume that the nodes in the graphs have distinct identifiers. This will be utilized below in the construction of the explanatory answer to a query. A function name can be mapped, through the implementation function, to a set of BPs. These represent alternative possible implementations for the function (one of which will be chosen at run time as the actual implementation). The implementation function is partial if the internal implementation structure of some functions is unknown (e.g. since their providers do not wish to expose their specification). Given a BP p and a function call n in p, a more detailed description of p can be obtained by replacing n by one of the function's possible implementations. A result of such replacements is called a *refinement*.

Definition 3. *Given a system $s = (S, s_0, \tau)$, a BP p, and a node n in p labeled by a label l for which τ is defined, we say that $p \xrightarrow{n} p'$ (w.r.t. τ) if p' is obtained from p by replacing n in p by one of its possible implementations $g \in \tau(l)$. [Namely, n is deleted from p, and a copy of g is plugged in its place, with the incoming/outgoing edges of n now connected to the start/end node of g, resp.]*

If $p \xrightarrow{n_1} p_1 \xrightarrow{n_2} p_2 \ldots \xrightarrow{n_k} p_k$, we say that p_k is a refinement *of p, and name the sequence of node replacements a* refinement sequence.

We say that a node $v \in p_k$ depends *on a node n_i in the sequence if $v \in p_i$ but $v \notin p_{i-1}$. v* depends transitively *on n_i if it either depends on n_i or depends on some node n_j transitively depending on n_i.*

Queries We now consider queries and their answers. Queries are modeled using *BP patterns.* These generalize BPs similarly to the way tree patterns generalize XML trees. Formally,

Definition 4. *A BP* pattern *is a tuple $\hat{p} = (p, I_e, I_f)$, where p is a BP and I_e, I_f are distinguished sets of edges and function names in p, resp. These are the* indirect *edges and functions of $\hat{p}$.*

A query *q is a system of* BP patterns *(Q, q_0, τ), where Q is a set of BP patterns, q_0 is the root BP pattern, and τ is an implementation function.*

Embeddings To evaluate a query, its patterns are embedded into the system BPs. Generally speaking, every type of relation over (finite) flat graphs may be generalized to an embedding type. We suggest here the usage of three main types of graph relations - homomorphism, isomorphism, and bisimulation. These are generalized to *homomorphic-* and *isomorphic-embeddings* (which capture the structural query interpretation) and *bisimilar-embedding* (capturing behavioral interpretation). We define these next. We consider first the embedding of a single BP pattern, then of full queries.

Definition 5. *Let $\hat{p}$ be a BP pattern and let p be a BP. An* homomorphic (resp. isomorphic)-embedding *of $\hat{p}$ into p is a homomorphism (isomorphism) ψ from the nodes of $\hat{p}$ to the nodes of p s.t.*

1. `(nodes)` *each node of $\hat{p}$ is mapped to a node of p having the same label; the start (resp. end) node of $\hat{p}$ is mapped to the start (resp. end) node of p.*
2. `(edges)` *for each (indirect) edge of $\hat{p}$ from a node m to a node n there is an edge (path) in p from $\psi(m)$ to $\psi(n)$.*

Definition 6. *Let $\hat{p}$ be a BP pattern and let p be a BP. A* bisimilar-embedding *of $\hat{p}$ into p is a binary relation R between the nodes of $\hat{p}$ and the nodes of some subgraph p' of the transitive closure* [2] *of p s.t.*

1. `(nodes')` *for each node $n \in \hat{p}$ [resp. each $n' \in p'$] there exists some node $n' \in p'$ [$n \in \hat{p}$] s.t. $R(n, n')$ holds; whenever $R(n, n')$ holds, n and n' have the same label and if one is a start/end node then so is the other.*

[2] The transitive closure of a graph is obtained by adding edges (specially marked as 'indirect') between any two nodes n, m such that m is reachable from n.

2. `(edges')` *for each (indirect) edge from a node n to a node m in $\hat{p}$, [resp. from n' to m' in p'] there exists a (indirect) edge from some node n' to some m' in p' [resp. from some n to some m in $\hat{p}$] s.t. $R(m, m')$ and $R(n, n')$ hold.*

In the sequel, when some definition/result applies to all homomorphic, isomorphic, and bisimilar embeddings we will denote all by *X-embedding*.

We now consider the embedding of a query consisting of a set of such BP patterns into a specification.

Definition 7. *Let $q = (Q, q_0, \tau_q)$ be a query and let $s = (S, s_0, \tau_s)$ be a system of BPs. An* X-embedding *of q into s consists of*

1. *An homomorphism h from the BP patterns in Q to the BPs in S and their refinements that (i) maps the root pattern q_0 of q to the root BP s_0 of s, and (ii) maps, for each (indirect) function name c in q, the BPs in $\tau_q(c)$ to (refinements of) the BPs in $\tau_s(c)$.*
2. *An X-embedding for each $\langle BPpattern, BP\rangle$ pair in the homomorphism.*

To conclude, we need to define the query semantics. We distinguish between *boolean* and *explanatory* answers for a query. The boolean X-answer to a query q on a system s is positive if such X-embedding exists and is negative otherwise. The explanatory X-answer consists of s's components participating in such X-embeddings, as defined formally below.

Definition 8. *The nodes and edges of a system s that are* relevant *to a given X-embedding include*

1. *the nodes of s in the ranges of the mappings (ψ or R, depending on the embedding type)*
2. *the edges and nodes of s appearing on paths between these nodes and which could be used to verify requirement* `(edges)` *(resp.* `(edges)'`*) for the embedding.*
3. *the nodes on which any of the above depend on, transitively (see Definition 3).*

The explanatory *X-answer of a query q on a system s, denoted $q_X(s)$, is a restriction of s to those nodes and edges that are relevant to some X-embedding of q in s. (Empty BPs are removed and the domain of τ is restricted to the relevant functions).*

In the sequel, we will refer to BPQL, under isomorphic, homomorphic, and bisimilar embeddings, as isoBPQL, homBPQL, and bisBPQL, resp. One may also consider combinations, allowing the user to specify different interpretations for various BP patterns in the query, and our results will still hold.

3 Related Models & Languages

Before presenting our query evaluation algorithm, we first set the background by looking at some closely related models and languages. We compare our work to relevant works in three areas, namely Model Checking, Formal Models and Databases. We classify the works according to the structural/behavoiural dichotomy, and discuss their relationships. Due to space constraints, we cannot give the formal definitions of each model we discuss, and the reader is referred to the literature. In the following, we use $BPQL_{spec}$, and $BPQL_{query}$ to denote the specification and query parts of BPQL, respectively. We start by formally defining the notion of model and query languages containment for models that define sets of finite graphs. In the following, $\equiv$ denotes graph isomorphism, and $\simeq$ denotes query equivalence (where two boolean queries Q_1, Q_2 over graphs are considered equivalent if a graph satisfies Q_1 iff it also satisfies Q_2).

Definition 9. *For two models M_1, M_2, $M_1 \subseteq M_2$ if for all $m_1 \in M_1$ there exists $m_2 \in M_2$ s.t. m_1, m_2 represent respectively (possibly infinite) sets of graphs S_1, S_2, and $\forall G_1 \in S_1$ $\exists G_2 \in S_2$ s.t. $G \equiv G'$. Also, $|m_2|$ is required to be linear in $|m_1|$.*

$M_1 \sim M_2$ if $M_1 \subseteq M_2$ and $M_2 \subseteq M_1$.

For two (boolean) query languages L_1, L_2 over some domain D, $L_1 \subseteq L_2$ if for all $Q_1 \in L_1$ there exists $Q_2 \in L_2$ s.t. $Q_1 \simeq Q_2$, and $|Q_2|$ is linear in $|Q_1|$.

$L_1 \sim L_2$ if $L_1 \subseteq L_2$ and $L_2 \subseteq L_1$.

Model Checking Several models similar to our model of nested graphs appear in works related to model checking. A common model that captures this semantics is named Recursive State Machines (RSM) [4]. This model naturally extends Finite State Machines (FSM), by allowing some states to be *call* states, invoking other FSMs. A call is simulated by replacing the call state by its implementation. Each FSM has some entry and exit states. The simplest form of RSM is Single Entry Single Exit RSM (SERSM), where each FSM has unique start and exit nodes. It is straightforward to prove the following proposition.

Proposition 1. *$BPQL_{spec} \sim SERSM$*

By their nature, works in the area of Model Checking use *behavioral* query languages, being interested in properties of the process's possible executions rather than its exact structure. *Temporal logics* are used to capture such properties. The most common of these are LTL, CTL*, and the more powerful alternation-free μ-calculus. These logics consider the behavior of programs over time, and differ in their quantifiers. CTL^* allows queries over branching execution paths, and supply corresponding quantifiers; LTL considers the time as linear, and does not allow branching; μ-calculus is the most general temporal logic, containing fix-point operators (μ and ν), that allow recursive iterations over the queried process. A particular fragment of μ-calculus, called 'alternation-free', is the one consisting of formulas that contains no μ operator depending on ν or vice versa. The exact definitions can be found in [12]. We can show the following:

Proposition 2. *$bisBPQL_{query} \subset$ alternation-free μ-calculus*

Specifically, using [10] one can easily obtain an evaluation algorithm for alternation-free μ-calculus over SERSM, of complexity $poly(|spec|)*2^{|query|}$. This algorithm can be used to answer $bisBPQL$ queries with the same complexity.

Formal Languages There is rich literature on Context Free Processes in the area of Formal Models. A main branch of this research concerns *Context Free Graph Grammars*, first introduced in early works such as [24]. These grammars generalize the common model of context free grammars over strings. Similarly, the grammar consists of a set of non-terminals and derivation rules. Each non-terminal derives labeled graphs, which in turn contain objects (nodes, edges, etc.) labeled by non-terminals. The rules are accompanied by instructions on how to connect the new graphs to the original graph. These instructions are called the *connection relation.* The literature (e.g. [20]) considers mainly two particular cases of context free graph grammars: Hyperedge Replacement (HR) grammars, where the non-terminals in the graph are hyperedges, and Vertex Replacement (VR) grammars, where the non-terminals are graph *nodes.* By [20], $HR \subset VR$.

The following proposition establishes the connection between our specification model and context free graph grammars.

Proposition 3. $BPQL_{spec} \subset HR \subset VR$

The work on these models is mostly theoretic, and uses, for query formalism, formal logics such as $FO(TC)$[3] or MSO[4]. The following theorem from [14] shows decidability of MSO over HR graph grammars.

Theorem 1. *[14] It is decidable whether a given MSO formula is satisfied by any graph generated by a given HR graph grammar*

As FO(TC) and MSO are *structural* query languages, it is suitable to compare the structural variants of BPQL to these logics. It is easy to see that

Proposition 4. $isoBPQL_{query}$,$homBPQL_{query} \subset FO(TC) \subset MSO$

Using Prop. 3, 4 and Thm. 1, we obtain:

Theorem 2. *isoBPQL, homBPQL are decidable*

Where bisBPQL is decidable as well, as implied from the discussion above. However, though the proof of theorem 1 is constructive, i.e. provide a decision procedure, it is unfeasible for practical use, as its complexity is non-elementary in the size of the query.
Databases Works in the Database world typically consider the representation of data as *flat* graphs (e.g. [13, 1]). Models that consider nested relations [3], actually consider flat trees . One exception that does consider nested graphs (trees) is *Active XML* (AXML) [2]. AXML is an extension of XML where the XML trees may contain nodes that represent calls to Web services. When invoked, the calls return new AXML trees that replace the call element. AXML data is queried using standard XML query languages like XQuery [11]. However, the semantics relates only to the full (possibly infinite) refinement of the document and does not allow queries of finer granularity. For this purpose, the model presented here can be adapted.

4 Query evaluation for BPQL

To evaluate a query q on a system s, we need to embed the BP patterns in q within (refinements) of the BPs in s. We assume first the existence of some oracle, denoted *X-match*, that given a single BP pattern $\hat{p}$ and some BP p, computes

[3] First Order logic augmented by a Transitive Closure operator
[4] Monadic Second Order Logic

the X-embeddings of $\hat{p}$ into p. (We will consider the implementation of such an oracle later). We start by showing how to use this oracle to find X-embeddings of $\hat{p}$ into *refinements* of p. Later, we use this to derive an evaluation algorithm for the full query.

Our algorithm is inspired by the original BPQL query evaluation algorithm presented in [6]. However, unlike that algorithm, which is applicable only to structural queries, the present algorithm is designed in a modular manner that can be parameterized by the required type of embedding. This is achieved by modeling the queries as logic formulas – FO(TC) formulas for structural queries and μ-calculus formulas for behavioral ones – and using a similar formula decomposition method for both, as described below. We sketch here an intuitive description for the boolean version of our algorithm, and then explain how to obtain its explanatory version. A full description of the algorithm, as well as its correctness proof, can be found in the full version of the paper [17].

Embedding a single pattern We start by explaining how to find, given a system s, a BP p and a BP pattern $\hat{p}$, X-embeddings of $\hat{p}$ into refinements of p. Our algorithm first constructs (1) a graph grammar G_p that describes the possible refinements of p (w.r.t s), and (2) an FO(TC) or μ-calculus formula, depending on the embedding type, $F_{\hat{p}}$ that represents the pattern $\hat{p}$. It then uses the two to compute a new graph grammar that encodes the X-embeddings of $\hat{p}$ into refinements of p. The boolean query answer will be positive iff the constructed grammar is not empty. We explain each of these steps below.

Grammar We first construct a graph grammar for the system s, as explained in the previous section. We use the result of [22] stating that an HR graph grammar can be translated into a normal form, where each graph includes only two non-terminals. We assign to the normal-formed grammar a new root non-terminal R that derives the BP p, and denote the resulting grammar by G_p. It is easy to see that the set of graphs derived from R in G_p corresponds precisely to the set of possible refinements of p w.r.t s.

Formula The formula for $\hat{p}$ uses two types of predicates: $L_A(n)$ holds iff the given BP contains a node n having a label A. $Path(n, m)$ holds iff there is a path from node n to node m. In general, each pattern $\hat{p}$ can be expressed as a conjunction of these predicates.

The distinction between the different embeddings sought for is expressed in the formula construction: For homBPQL and isoBPQL, variables are interpreted over individual nodes, while for bisBPQL they are interpreted over sets of nodes. Also, $isoBPQL$ formulas contain additional clauses representing inequalities between the node variables.

Algorithm We use the graph grammar G_p and the formula $F_{\hat{p}}$ described above to construct a new graph grammar that encodes the embeddings of $\hat{p}$ in refinements of p. The basic idea is similar to the one used in verification algorithms, e.g. [4]. We try all splits of the formula $F_{\hat{p}}$ up into 3 parts, each of which is 'not larger' then the original formula. Each part is then handled separately, as follows. The first part is embedded directly within p, where the other two parts are embedded recursively within the implementations of p's function call nodes. To capture this recursive embedding, we replace within (the grammar representation of) p its two non-terminals N_1,N_2, that represent the function calls, by (N_1, F_{N_1}) and (N_2, F_{N_2}), (where F_{N_1}, F_{N_2} are the above mentioned parts of $F_{\hat{p}}$) and we continue recursively to finding embeddings of F_{N_1} (F_{N_2}) within the implementation of N_1 (N_2). Intuitively, we find the fix-point of the set of constraints generated.

Formula Decomposition To complete the algorithm description, we only need to describe the split of a formula F. For a BP g with two function call nodes (grammar non-terminals) N_1, N_2, we define the *split* F into three formulas denoted F_g, F_{N_1} and F_{N_2}. This is done by considering all possible splitting of the node predicates of F into three sets [5] f_g, f_{N_1}, f_{N_2} (representing the nodes to be embedded in g, N_1, and N_2, resp.) and then splitting the remainder of F based on this nodes split. The node predicates in F_g, F_{N_1}, F_{N_2} are trivially f_g, f_{N_1}, f_{N_2}, respectively. We further need to consider the paths connecting the nodes. The splitting of the path formulas depends upon the nodes split - path predicates with both end-nodes in f_{N_1} (resp. f_{N_2}) are added [6] to F_{N_1} (resp. F_{N_2}). The treatment of path predicates with one end-node in f_{N_1} and the other in f_{N_2} is similar: these are split into three parts s.t. one describes the sub-path to be embedded in N_1 (the corresponding path predicate is added to F_{N_1}), the second describes the sub-path to be embedded in g, connecting N_1 to N_2 (added to F_g), and the third describes the sub-path to be embedded in N_2 (added to F_{N_2}). We can show the following theorem, used in the algorithm correctness proof.

Theorem 3. *(informal) A pattern $\hat{p}$ can be X-embedded within a BP p, containing call nodes N_1 and N_2 labeled l_{N_1} and l_{N_2} resp., if and only if there exists a split of $F_{\hat{p}}$ into F_1, F_2, F_3 (as described in the algorithm) such that F_1 can be X-embedded into p without matching N_1 and N_2, F_2 and F_3 can be X-embedded into the implementations of l_{N_1} and l_{N_2} respectively.*

Evaluating a full query The algorithm above constructs a graph grammar that encodes the embedding of a single BP Pattern. Extending it to handle a full BPQL query is fairly straightforward. For each indirect function call node in the query, we use the algorithm above to compute the graph grammar rules representing the embeddings of the function's implementation into refinements of the corresponding call node in the system. If any of the computed grammars happens to be empty, we stop and return an empty graph grammar. For the direct call nodes in the query, as well as for the query root BP pattern, we use directly the X-match oracle to obtain grammar rules describing their possible (direct) embedding into the corresponding system BPs. Here again, if any of these embeddings fail, we stop and return an empty grammar.

The following theorem states the correctness of the algorithm. The proof appears in the full version of the paper [17].

Theorem 4. *The grammar constructed by the algorithm is not empty iff an embedding exists*

[5] For structural queries the sets are required to be disjoint.

[6] Note that all formulas are conjunctive, so whenever we refer to 'adding' a formula f_1 into a formula f_2 we mean generating the conjunction $f_1 \bigwedge f_2$.

The explanatory query answer can also be easily obtained from the above algorithm, as it maintains the unique identifiers of all nodes and edges being used. These can be extracted from the constructed graph grammar and used to generate the explanatory answer.

5 Complexity

The complexity of the algorithm presented in the previous section depends on the complexity of the X-match oracles. We first examine the complexity of such oracles for isomorphic, homomorphic and bisimilar embeddings. Next we analyze the complexity of the full algorithm, parameterized by the oracle's complexity.

X-match oracles Given a BP pattern $\hat{p}$ and some BP p, X-match computes the X-embeddings of $\hat{p}$ into p. For the three types of embedding, the problem of testing for the existence of an embedding is NP-complete w.r.t the size of the query pattern, but polynomial in the data size. (The proof follows immediately from the NP-completeness of subgraph isomorphism/homomorphism/bisimulation [16, 18]). A worst case complexity for the oracles is thus $O(p^{\hat{p}})$. However, using optimization techniques, this is typically much lower in practice [21].

The overall algorithm For a given X-match oracle, we use O(X-match(n,m)) to denote the worst case time complexity of the oracle when embedding a query pattern of size m into a BP of size n.

Theorem 5. *Given a BP system s and a query q, the time complexity of (the Boolean and Explanatory versions of) the query evaluation algorithm presented in the previous section is* $O(\mid s\mid^2 \times c^{\mid q\mid} \times O(X\text{-}match(\mid s\mid, \mid q\mid)))$*, where c is a constant.*

Thus, the algorithm is polynomial in the size of the system s [7] and in the complexity of the X-match oracle, but is exponential in the size of the query. Since testing for the existence of isomorphic-, homomorphic-, and bisimilar-embeddings is NP-hard in the size of the query, it is evident that testing if the answer to a iso-,hom-, and bisBPQL is empty is also NP-hard in the query size. Interestingly, we can expose an additional type of hardness that comes from the nested shapes of the system and query graphs, as follows.

Theorem 6.
1. *Boolean hom-, iso-, and bisBPQL are NP-hard in the size of the query even when the system BPs and the query patterns belong to a restricted class of graphs for which the X-match can be computed in polynomial time.*
2. *For homBPQL and bisBPQL, the above holds if, furthermore, all the call nodes in the system and the query have only one possible implementation.* [8]

It is open if (2) holds also for isoBPQL. The proof (appearing in the full version) is by reduction from the problem of testing if a 3NF formula is satisfiable, known to be NP-complete. The graphs used in the proof have very simple, almost tree-shaped structure, where all nodes besides the end nodes have a single parent.

To give a lower bound we can show that

[7] The complexity is quadratic in the size of the system because of the mapping to normal form grammars, resulting in a quadratic size grammar

[8] In general, the implementation function allows to map each function name to a set of BPs, representing alternative possible implementations for the function.

Theorem 7. *The Boolean versions of homBPQL and isoBPQL are in NP (combined complexity).*

The main lemma required in order to supply an NP algorithm is the following.

Lemma 1. *For every BPQL system s and homBPQL (isoBPQL) query q, exactly one of the following holds:*

1. *There is no homomorphic (isomorphic) embedding of q into s.*
2. *There is at least one homomorphic (isomorphic) embedding that maps nodes of q only to nodes of refinements obtained by a* polynomial *number of refinement steps.*

The correctness of this lemma stems from the correctness of the analogous lemma for context free *string* grammars. (The proof is in the full version). Interestingly, when the query is viewed as a logic formula, this property can also be viewed as an instance of the *Small Model Property.* It is open if the same holds for bisBPQL.

A different kind of analysis is obtained through *parameterized complexity*, where the size of one of the inputs which is typically small (the query size, in our case) is considered as a parameter k, and the size of the rest of the input is denoted n. A parameterized complexity class corresponding to $PTIME$ is FPT [19], which is the class of all problems solved with time complexity $O(f(k) * P(n))$, P being a polynomial and f being any computable function. An important hardness class, namely W[1]-hard [19], contains problems which are likely not to be in FPT, and thus is analogous to the class of NP-hard problems. We can show the following proposition.

Proposition 5. *If X-match is in FPT (resp. is W[1]-hard) then so is X-BPQL.*

Note the difference from conventional complexity analysis, where even for X-matches that are in $PTIME$, X-BPQL is NP-hard (See theorem 6(1)).

6 Conclusion

This paper studied the formal model underlying BPQL, a novel query language for BP specifications. We investigated its properties as well as the complexity of query evaluation, showed how queries on the structure and behavior of BPs can be processed in a uniform manner, and analyzed the relationship to previously suggested formalisms for processes modeling and querying. Because of space constraints, we have discussed only parts of the full BPQL model and query language, which include extensions such as regular path expressions, joins, and negation. Our results extend to this setting as well, as shown in [17].

To guaranty a complexity that is polynomial in the size of the data, BPQL ignores the run-time semantics of certain BPEL constructs such as conditional execution and variable values. Identifying semantic constructs that can nevertheless be incorporated without increasing the complexity is a challenging future research task. It would be interesting, following e.g. [15], to consider the data manipulated by BPs and the messages passed from one process to another. One may also consider a setting where calls are possibly asynchronous, or where the knowledge of the implementation of some (remote) processes may be partial [9]. It would also be interesting to combine our algorithm with some existing verification techniques, e.g. [21].

References

1. S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of asynchronous discrete event systems: datalog to the rescue! In *Proc. of PODS '05*, 2005.
2. S. Abiteboul, O. Benjelloun, and T. Milo. Positive active xml. In *Proc. of PODS '04*, 2004.
3. S. Abiteboul, P. C. Fischer, and H.J. Schek. Nested relations and complex objects. *LNCS*, 361, 1989.
4. R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4), 2005.
5. C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying Business Processes with BP-QL (demo). In *Proc. of VLDB*, 2005.
6. C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes. In *Proc. of VLDB*, 2006.
7. Business Process Execution Language for Web Services. http://www.ibm.com/developerworks/library/ws-bpel/.
8. BPMI. Business process management initiative: Business process: Business process query language (bpql). http://www.service-architecture.com/web-services/articles/business_process_query_language_bpql.html.
9. P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis. Using partial evaluation in distributed query evaluation. In *Proc. of VLDB*, 2006.
10. O. Burkart and B. Steffen. Model checking for context-free processes. In *Proc. of CONCUR'92*, 1992.
11. D. Chamberlin. Xquery: a query language for xml. In *Proc. of SIGMOD*, 2003.
12. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking.* MIT Press, 1999.
13. M. Consens and A. Mendelzon. The g+/graphlog visual query system. In *Proc. of SIGMOD*, 1990.
14. B. Courcelle. The monadic second-order logic of graphs. *Inf. Comput.*, 85(1), 1990.
15. A. Deutsch, L.Sui, V.Vianu, and D.Zhou. Verification of communicating data-driven web services. In *Proc. of PODS*, 2006.
16. A. Dovier and C. Piazza. The subgraph bisimulation problem. *IEEE Trans. Knowl. Eng.*, 15(4), 2003.
17. Querying structural and behavioral properties of business processes - full version. `http://www.cs.tau.ac.il/~danielde/BPQLFull.pdf/`.
18. M.R. Garey and D.S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.
19. M. Grohe. Parameterized complexity for the database theorist. *SIGMOD Rec.*, 31(4), 2002.
20. D. Janssens and G. Rozenberg. Graph grammars with node-label controlled rewriting and embedding. In *Proc. of COMPUGRAPH*, 1983.
21. M. S. Lam, J.Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proc. of PODS*, 2005.
22. T. Lengauer and E. Wanke. Efficient decision procedures for graph properties on context-free graph languages. *J. ACM*, 40(2), 1993.
23. Oracle BPEL Process Manager 2.0 Quick Start Tutorial. http://www.oracle.com/technology/products/ias/bpel/index.html.
24. T. Pavlidis. Linear and context-free graph grammars. *J. ACM*, 19(1), 1972.
25. A. Schurr. Logic based programmed structure rewriting systems. *Fundam. Inf.*, 26(3-4), 1996.
26. The World Wide Web Consortium. `http://www.w3.org/`.