Journal of Computer and System Sciences 78 (2012) 583–609

Contents lists available at SciVerse ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

A structural/temporal query language for Business Processes

Daniel Deutch [a,*], Tova Milo [b]

[a] Ben Gurion University, Israel
[b] Tel Aviv University, Israel

ARTICLE INFO

Article history:
Received 29 August 2010
Received in revised form 30 August 2011
Accepted 12 September 2011
Available online 14 September 2011

Keywords:
Business Processes
Query languages

ABSTRACT

A Business Process consists of multiple business activities, which, when combined in a flow, achieve some particular goal. These processes usually operate in a distributed environment and the software implementing them is fairly complex. Thus, effective tools for analysis of the possible executions of such processes are extremely important for companies (Beeri et al., 2006, 2007 [4,5]); (Deutch and Milo, 2008 [13]); these tools can allow to debug and optimize the processes, and to make an optimal use of them. The goal of the present paper is to consider a formal model underlying Business Processes and study query languages over such processes. We study in details the relationship of the proposed model with previously suggested formalisms for processes modeling and querying. In particular we propose a query evaluation algorithm of polynomial data complexity that can be applied uniformly to two kind of common queries over processes, namely queries on the structure of the process specification as well as temporal queries 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 particular 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. Complementarily, the 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 [29]. The declarative, yet complex, nature of BPEL specifications call for the design of a query language, that will allow to effectively analyze the possible executions of a given process. To answer this need, we have developed BPQL [3,4], a query language for querying business process specifications. We have then continued to extend the query language [12–14] to account for various analysis needs that rise in the context of Business Processes, and studied query evaluation in each context. However, what is missing from these previous works is the formal, fundamental positioning of the model and query evaluation algorithm within the context of other common formalisms for modeling

* Corresponding author.
E-mail addresses: deutchd@cs.bgu.ac.il (D. Deutch), milo@cs.tau.ac.il (T. Milo).

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2011.09.004

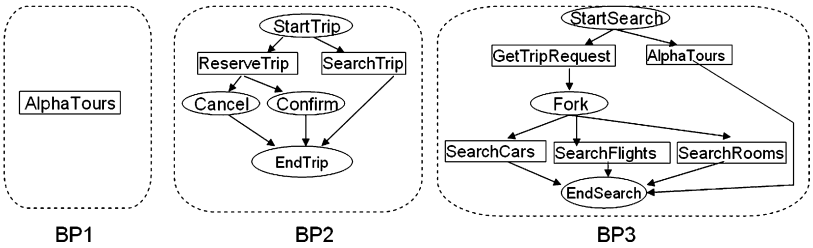

Fig. 1. A BPQL specification.

and querying processes. This positioning is the goal of the present paper. We note that some of the results presented here appeared also in [12], but only in a high-level form and without detailed proofs. Specifically, we show here that many data models are in fact equivalent (see Section 3 for a formal definition of model equivalence, and proofs that such equivalences hold) to our BP model. In particular, this means that our query evaluation results apply to these models as well. Among these commonly used models one can find restricted versions of Rewriting Systems (e.g. [32]), Recursive State Machines (RSMs) [1,6], Context Free Graph Grammars [10], 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 temporal [19]. Namely, the queries relate to the possible runs of the process defined by specification, and are used to identify invariants, execution patterns, etc. We provide here a unified environment for querying structural as well as temporal properties of business processes. We study the expressive power of our query language with respect to common languages, and explain how analysis tasks that cannot be expressed using temporal logic, are easily and intuitively formed using our query language. We then study the complexity of query evaluation over Business Processes, and provide query evaluation algorithms that may be applied uniformly with either the temporal or structural semantics. We show that these algorithms are practically feasible: they incur a worst case complexity that is polynomial in the size of the process specification, with the exponent dependent on the query size. We further show that the exponential dependency over the query size is unavoidable, unless $P = NP$.

Note. To guarantee 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. Appendix A provides additional formal details on the query evaluation algorithm.

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.

Fig. 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.

An example query is depicted in Fig. 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 Fig. 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

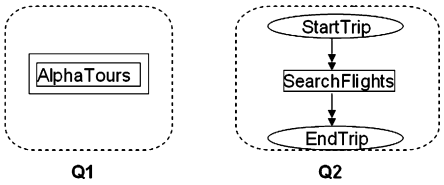

Fig. 2. A BPQL query.

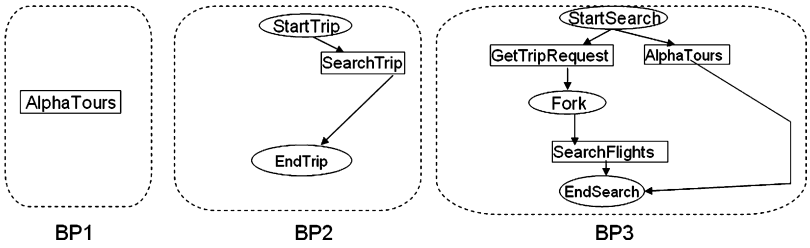

Fig. 3. Explanatory query answer.

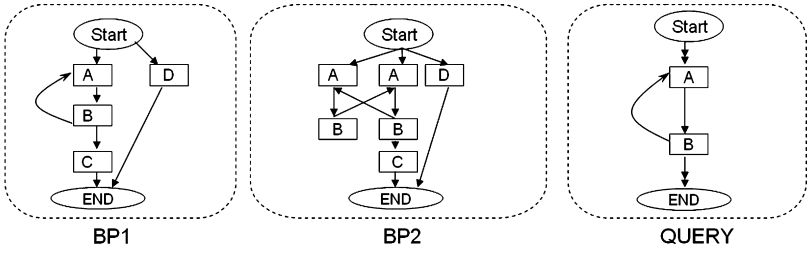

Fig. 4. Structural vs. behavioral.

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. An occurrence of the query pattern in the specification 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 Fig. 2 when applied on the system in Fig. 1 is depicted in Fig. 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). In general, a user may wish to focus on a particular part of the query and view only those system components that are relevant to this specific part. All the results presented in the paper can be easily generalized to this context. We omit this here.

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 simple query (BP pattern) depicted in Fig. 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 Fig. 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 simple fact that subgraph isomorphism/homomorphism cannot be expressed by any bisimulation-invariant language [9], and thus, in particular, by any temporal logic (as these are bisimulation-invariant [9]). Thus, structural queries cannot be formulated using the previous works. However, structural queries are 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 Fig. 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: imposing coding conventions, debugging, profiling, code optimizations, etc.

To conclude, we re-state that our framework is uniform and generic, allowing behavioral queries as well as structural queries. This forms a unification of important fragments of the common query languages over a simple common abstraction of BPs, using a simple, intuitive and graphical query language.

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 2.1. 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, along with a mapping of function names to their implementations.

Definition 2.2. 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 2.3. 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 $p \xrightarrow{n_1} p_1 \xrightarrow{n_2} p_2 \xrightarrow{n_3} \cdots \xrightarrow{n_k} p_k$, 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. For simplicity we consider first simple positive queries without negation and joins. These, and other extensions, are considered later. Queries are modeled using *BP patterns*. These generalize BPs similarly to the way tree patterns generalize XML trees. Formally,

Definition 2.4. 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 2.5. 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 2.6. 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[1] 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. (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 use the notation *X-embedding* to denote all.

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

Definition 2.7. 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. A 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 ⟨BP pattern, BP⟩ 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 2.8. 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 2.3).

The *explanatory* X-answer of a query q on a system s, denoted by $q_X(s)$, 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. As this does not affect the results presented in the paper we ignore it in the sequel.

Extensions. To simplify the presentation, we used above a very simple model and query language. The full BPQL model includes several useful extensions that enhance the expressive power, and facilitate the querying of real life business processes, without affecting the complexity of query evaluation. We discuss them below.

Regular path expressions. Indirect edges in the query may be annotated by regular expressions. The annotation of a regular expression T restricts the search to paths where the sequence of labels of the nodes on the path form a word in the regular language defined by T. The notions of the various X-embeddings extend naturally to this setting, with conditions (edges) and (edges′) of Definitions 2.5 and 2.6, resp., being refined. It now matches an edge in the BP pattern, that is annotated by a regular path expression T, to paths whose node labels form words in T.

As the term 'Path' may be slightly ambiguous, we start with an explicit definition of its intended meaning, as follows.

Definition 2.9. Given a graph $G = (V, E)$ and two unique nodes $start, end \in V$, a *path* from *start* to *end* is a sequence of nodes $P = (V_0, V_1, \ldots, V_k)$ such that $\forall i = 0, \ldots, k-1$. $V_i \to V_{i+1} \in E$, $\forall i, j$, $i \neq j$. $(V_i, V_{i+1}) \neq (V_j, V_{j+1})$ (distinct edges), $V_0 = start$, $V_k = end$.

[1] 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.

A finite *traversal* over P is a sequence $T = (U_0, U_1, \ldots, U_t)$ such that $U_0 = start$, $U_t = end$, and $\forall i,\ i = 0, \ldots, t-1$, $\exists k$ $(U_i = V_k \wedge U_{i+1} = V_{k+1})$.

Note that between any two nodes in a graph G there exists a finite number of paths (as the number of sequences of distinct edges in G is finite). A traversal over the path can only use the same nodes and edges as the path, but may repeat edges in case of cycles. For each path P there thus may be an infinite number of traversals over P (as we can traverse over each cycle any number of times).

Definition 2.10. For a path P with its nodes annotated with labels belonging to some alphabet Σ, we associate with P a regular expression $Reg(P)$ with every string $S \in Reg(P)$ constructed out of the sequence of labels obtained by some finite traversal over P.

$Reg(P)$ can be though of as the regular expression that represents the same language as the finite state automaton represented by the path.

Definition 2.11. A path P *conforms* to a regular expression T over Σ if $Reg(P) \cap L(T) \neq \phi$ where $L(T)$ is the language of all strings generated by T.

We are now ready to refine the definition of X-embeddings to patterns with regular path expressions. We start by homomorphic- and isomorphic-embeddings, then consider bisimilar-embeddings.

Definition 2.12. 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 in $\hat{p}$ to the nodes of some subgraph p' of p s.t.

1. (`nodes`) each node in $\hat{p}$ is mapped to a node of p' with the same label; the start (resp. end) node in $\hat{p}$ is mapped to the start (resp. end) node of p'.
2. (`edges`) for each (indirect) edge from a node m to a node n (marked by a regular expression T) in $\hat{p}$ there is an edge (path) P in p' from $\psi(m)$ to $\psi(n)$ (such that P conforms to T).

In order to adapt the definition of bisimilar-embedding to handle regular expressions, we define the *annotated transitive closure* of a graph G to be the transitive closure of G, where each indirect edge e that was placed as replacement for a (finite) set of paths $P = \{P_1, \ldots P_k\}$ being marked with $Reg(P) = Reg(P_1) \cup Reg(P_2) \cup \cdots \cup Reg(P_k)$. The annotation of an edge e is denoted $Reg(e)$.

Definition 2.13. Let $\hat{p}$ be a BP pattern and let p be a BP. A *bisimilar-embedding* of $\hat{p}$ into p is binary relation R between the nodes of $\hat{p}$ and the nodes of some subgraph p' of the annotated transitive closure 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. (`edges'`) for each (indirect) edge from a node n to a node m in $\hat{p}$, marked by T [resp. from n' to m' in p'] there exists an (indirect) edge e' 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 (and $Reg(e') \cap T \neq \phi$).

Negation. In a query with *negation*, the patterns include some nodes and edges that are distinguished as *negative*. The intuitive interpretation is that the query searches for occurrences of the positive portions of the patterns, for which none of the negative parts co-occur. More formally, to define the semantics of queries with negation we extend the notion of X-embedding: Given a BP pattern $\hat{p}$ with negation, the *positive part of* $\hat{p}$, denoted by $positive(\hat{p})$, is the pattern obtained from $\hat{p}$ by deleting all the negative edges and nodes, and all the edges incident on these nodes. An X-embedding of $\hat{p}$ into a BP p is an X-embedding of $positive(\hat{p})$ which cannot be extended, by adding nodes corresponding to the negative part, to an embedding of $\hat{p}$. Now, the X-embedding of a query q is defined as before, with this refined embedding being used for the BP patterns in q that contain negation.

Label predicates, variables and joins. Nodes in query patterns may be annotated with *predicates* that can be satisfied by more than one label. This is useful when users are interested in a set of nodes whose name share some common property (e.g. contains the string "search"), rather than a specific node. To support this, when looking for an X-embedding, a query node annotated with a predicate P can be mapped to any node whose label satisfies P.

Together with label predicates, one can also attach label *variables* to nodes and test for (in)equality of the assigned labels. Label variables, and joins based on (in)equality of such variables, are incorporated within our framework in a natural fashion. Note however that the complexity of the query evaluation algorithm becomes $O(|s|^{|q|} \times O(\text{X-match}(|s|, |q|)))$ (compared to $O(|s|^2 \times c^{|q|} \times O(\text{X-match}(|s|, |q|)))$ in the simpler model, see Section 5). On the other hand, Beeri et al. [4] have

introduced *path variables*, namely the expressive power to require that paths occurring in different parts of the pattern will bear the same sequence of node labels. It was shown in [4] that this added expressive power renders the problem of testing emptiness of a query answer undecidable (or PSPACE-hard, for non-recursive systems). Joins on path variables are therefore not supported by BPQL.

Data elements and process properties. Nodes in the BP graph may represent process properties such as its provider, capabilities, etc. They can also represent data elements that serve as input/output to BPs. Similarly, BPQL queries may restrict the search to processes having certain properties (e.g. provided by a given provider) or inquire about the flow of data (e.g. which data elements serve as input, possibly transitively, to a certain BP). Note, however, that BPQL queries cannot inquire about the potential run time *value* of the data.

Distributed systems. In a distributed setting, each peer holds a set of BPs and may provide (use) processes to (of) remote peers. To support this in the BPQL model, we associate a peer id with each BP and node. For queries, we allow annotation of BP patterns and nodes with a peer id (or a predicate on the peer identifiers), having the semantics of restricting the search to BPs supplied by the specified peers. The BPQL query engine [4,5] supports distributed query evaluation.

3. Related models and languages

Before presenting our query evaluation algorithm, let us first set the background by looking at some closely related models and languages. We will thus obtain a comparison point for the expressivity of BPQL and for the complexity of our evaluation algorithm and its properties. In the discussion we shall address two angles of BPQL: (1) the model used to describe the business processes, and (2) the query language used to specify the properties of interest. These two angles are denoted below, resp., by BPQL_{spec} and BPQL_{query}.

In principle, one could derive a query evaluation algorithm for BPQL by adapting an existing evaluation algorithm for some language equivalent or more powerful than BPQL_{lang} on a model equivalent or more powerful than BPQL_{spec}. We will see that while such languages and models do exist, the adaptation of existing algorithms does not prove to be practical. We will focus on models that we believe are most relevant for our work. We first consider data models, then query languages, and finally existing evaluation algorithms for the queries in these models. For both categories (data/query languages) we start by defining notions of languages relations (i.e. inclusion and equivalence). These notions will be utilized when comparing different models.

3.1. Specification models

On comparing the expressive power of specification languages. As mentioned in the introduction, specifications can be interpreted in two different ways – structural or run-time. A choice of either of the approaches implicate on the comparison between models, as we explain next. The models we are concerned with, generate an infinite set of finite state machines (i.e. finite directed labeled graphs) which corresponds to the set of all expansions (re-writings), obtained by replacing a node with a label l by some graph G' which depends only on l. These models are categorized as *context free processes*. Two approaches exist, when tackling these models. The *structural* approach considers all possible structures of the specification graph expansions (re-writings), and states that $L_1 \subseteq L_2$ if for each $S_1 \in L_1$, $S_2 \in L_2$, and for each finite state machine graph G_1 generated by S_1 there exists an isomorphic finite state machine graph G_2 generated by S_2. The runtime approach concerns all possible executions over these structures, and thus the structures need not be 'the same' (i.e. isomorphic) but rather representing the same set of runs (i.e. bisimilar). As two isomorphic graphs are also bisimilar, but the converse is not necessarily true, the structural requirement is stronger. We consider structural queries as well as executional queries, and thus we use the stronger notion of structural inclusion (and induced by it, the notion of equivalence: $L_1 \sim L_2$ iff $L_1 \subseteq L_2$ and $L_2 \subseteq L_1$) of specification languages. As always in reduction, the question of the result size rises. In all of our specification reductions, the size of the resultant model is linear in the size of the original model. We typically give a syntactical translation and show that the semantics remains intact.

Common models. Common models used in the literature to describe business processes, and software specifications in general, include finite state machines, recursive state machine, graph grammars (in various flavors), and equational sets. We describe them below.

Finite and recursive state machines. A *finite state machine* (*FSM*) [33] is an edge-labeled directed graph, where the nodes represent *states*, and the edges represent transitions. The labeling of the edges may represent conditions for the transition to occur, actions to execute upon transitions, or both. FSMs can be used to describe simple "flat" processes. A *recursive state machine* (*RSM*) [6,1] is an extension of FSM, introducing a definition of a node *implementation*. In a way very similar to BPQL_{spec}, an implementation of a label of an RSM node is itself an RSM. An expansion of the state machine is defined as replacing a node with an implementation of its label, and connecting the implementation as a subgraph into the original graph. Some variants exist on the connection pattern, where the simplest version requires each implementation RSM to have

a single entry and a single exit nodes. This restriction of the model is called *Single Entry Single Exit RSM* (SRSM). BPQL_{spec} is equivalent to *SRSM*, as we shall prove. We start by giving the formal definitions of *RSM* and *SRSM*.

Definition 3.1. A recursive state machine (RSM) A over a finite alphabet Σ is a set $M = \{M_1, \ldots, M_k\}$ where each $M_i = (N_i, B_i, En_i, Ex_i, X_i, \delta_i)$ is called a *component structure*, and contains:

1. A set N_i of nodes and a set B_i of boxes (corresponds to regular and composite nodes in BPs).
2. An indexing $Y : \{B_1, \ldots, B_m\} \mapsto \{1, \ldots, k\}$ that assigns for every box an index of one component structure.
3. A set of entry nodes En_i and a set of exit nodes Ex_i.
4. A labeling function $X_i : N_i \mapsto \Sigma$.
5. A transition relation δ_i with transitions of the form (u, σ, v) where u is either a node in N_i or a pair (b, x) with b being a box in B_i and x being an exit node in Ex_j for $j = Y_i(b)$, the label σ is in $\Sigma \cup \{\varepsilon\}$, and v is either a node in N_i or a pair (b, x) with b being a box in B_i and x being an entry node in En_j for $j = Y_i(b)$.

An RSM is single-entry if $\forall i \ |En_i| = 1$, and it is single-exit if $\forall i \ |Ex_i| = 1$. We denote a single-entry single-exit RSM by SRSM.

The semantics of SRSM is given through the following definition of Kripke structures.

Definition 3.2. A (infinite) Kripke structure over a set of atomic propositions P is a tuple (S, R, L) where S is a possibly infinite set of states, R partial to $S \times S$ is the transition relation, and $L : S \to 2^P$ is the labeling function, associating with each state the set of atomic propositions that is true in this state.

Kripke structures are used to formalize *expansions* of the SRSM, in a similar manner to the manner *refinements* relate to BPs. Specifically, when considering an expansion of *SRSM*, it corresponds exactly to a Kripke structure. The nodes of the expansion are interpreted as the states, its edges as the transition relation, and the labels of the expansions are interpreted as the labels of the Kripke structure. Usually the term 'Kripke structure' relates to finite state machines. Here, following [6], the number of states may be infinite.

Each of the above-mentioned terms (BP refinements and SRSM expansions) represents the semantics of the evolving processes, in the corresponding model. We next show that the model of SRSMs is equivalent to the model of BPs. As part of the proof, we show that the terms of *expansion* and *refinement* are indeed equivalent.

Theorem 3.3. *Single Entry Single Exit Recursive State Machines* (*SRSM*) *and Business Processes model are of the same expressive power.*

Proof. We start with a syntactical translation between SRSM and BP. The names of the ingredients are different but their semantics is the same. The technical details follow.

Recall that an RSM is a set of component structures $M = \{M_1, \ldots, M_n\}$. M is equivalent to the set of BPs S. M_1 is called the top-level structure of M and is equivalent to the root system process S_0 by the terminology of BPs. We next show that a component structure is equivalent to a BP.

A BP can be represented by a tuple (G, L, u, v), where: G is a graph, L is a labeling function, labeling each node with one label out of a set of concrete labels, or assigning it an external function label, which means that it is a composite node. u and v are the start and end nodes.

On the other hand, a component structure consists of:

- A finite set of nodes N_i. This set corresponds to the set of nodes of the BP graph.
- A finite set of boxes B_i. A box corresponds to a composite node.
- A non-empty subset of the nodes N_i, called the entry nodes and denoted by I_i. If the RSM is a single-entry one, then each set I_i is a singleton, and thus corresponds to the start node of the BP.
- A non-empty subset of the nodes N_i, called the exit nodes O_i. If the RSM is a single-exit one, then each set O_i is a singleton, and thus corresponds to the end node of the BP.
- A labeling function $X_i : N_i \to 2^P$, that labels each node with a subset of P. This corresponds to the labeling function of BP.

So far we've showed the equivalence of the static structures of the two models. We also need to consider expansions (re-writings) of SRSMs (resp. BPs), and to show that these two concepts are indeed equivalent. We do this by showing a syntactical translation between the primitives that concern expansions and re-writings, as we continue listing the components of SRSMs and comparing them to the components of BPs.

- An Indexing function $Y_i : B_i \to \{1, \ldots, n\}$ that maps each box to the index j of some structure M_j. This corresponds to the implementation function for composite nodes.

- A set C_i (call nodes) of pairs of the form (b, e) where b is a box in B_i, and e is an entry-node of M_j. This attaches a box to a start point of the callee.
- A set R_i (return nodes) of pairs of the form (b, x) where b is a box in B_i, and x is an exit-node of M_j, $j = Y_i(b)$. This attaches a box to an end point of the caller.

Note that the two definitions above are redundant in case of single-entry single-exit model as $Y_i(b)$ determines $C_i = (b, S(M_i))$, and $R_i = (b, E(M_i))$ and thus it can be neglected.

- An edge relation E_i, with every edge being a pair (u, v) s.t.
  1. u is either a node in N_i or a return node in R_i.
  2. v is either a node in N_i or a call node in C_i.

  When considering the single-entry single-exit model, this corresponds exactly to the terms of re-writings in BPs, as for any edge in a (rewriting of) BP one of the following is true:
  1. Connecting two nodes in the original graph or
  2. The target of the edge can be a start node of an expansion (if the edge's target was rewritten) or
  3. The origin of the edge can be an end node of an expansion (if the edge's target was rewritten) or
  4. Both ends can be the result of such re-writings.

  These cases correspond exactly to the 4 combinations of (1) and (2) above.
- Substitution of a box b is done by inserting the structure M_j s.t. $j = Y_i(b)$ and connecting it to the nodes in M_i according to the edge relation E. This is equivalent to re-writings. In the single-entry single-exit model E is enforced to connect any node that is connected to an expansion of B, to the only entry (or exit in the opposite case of the edge) of B. This completes the equivalence to BP, where composite nodes $\sim$ boxes.
- The expansion (which of course can be infinite) $K(M)$ of an RSM M is the Kripke structure (S, R, L) defined as follows: A state of the structure is defined by a node, and a description of how it was created, namely a finite sequence of boxes. This sequence is called the node's *context*.

  R is the set of transitions $((v, w), (v', w'))$ that satisfy any of the following:
  - $(v, v') \in E_i$, $v, v' \in N_i$, $w = w'$ (same context, original edges),
  - $(v, (b', e')) \in E_i$, $v \in E_i$, $v' = e'$, and $w' = wb'$ (applying rewriting on b'),
  - $((b, x), v) \in E_i$, $v = x$, $v' \in N_i$, $w = w'b$ (again, applying rewriting, but on the other side of the edge),
  - $((b, x), (b', e')) \in E_i$, $v = x$, $v' = e'$, and $w' = w''b'$ with $w = w''b$ (applying one rewriting on one side, another on the other).

  This is just a formal definition of the expansion process, which complies with rewriting in BPs, along with keeping track of contexts.
- Finally, the labeling function $L : S \to 2^P$ is defined by $L((v, w)) = X_i(v)$, for i s.t. $v \in M_i$. This just means that every node keeps its original label, as in BPs. □

Context free graph grammars. The notion of context free graph grammars was introduced in early works such as [31]. The idea is that similar to the notion of string grammars where non-terminals appear in strings and are associated with derivation rules that allow to replace them with sub-strings. With graph grammars, the non-terminals may be associated with some objects in the graph (nodes, edges, paths, k-size cliques, etc.) and rewriting rules specify for each non-terminal, the subgraphs that it can be replaced with. A difficulty that is new here (with respect to string grammars) lies in how the newly derived subgraphs are connected to the original graph (where the non-terminal appeared). Thus, the rules are accompanied by connection instructions on how to connect these new graphs to the original graph. These instructions are called the *Connection Relation*. The literature (e.g. [24,11]) considers mainly two particular cases of context free graph grammars: Hyperedge Replacement (HR) grammars, where the non-terminals in the graph are hypergraphs, and Vertex Replacement (VR) grammars, where the non-terminals are graph *nodes*. For the former, the connection relation is implied by the nodes residing on the hyperedge. These nodes are connected, upon replacement, to the corresponding nodes of the inserted graph. The latter allows any connection relation to be defined, hence its expressive power is stronger [11].

There is a tight connection between context free grammars and equational sets, to be defined next. An equational set [27] is the set of least solutions to an equations system $\langle U_1 = p_1, \ldots, U_n = p_n \rangle$ with the U_i's being variables and each p_i, called *polynomial*, is of the form $t_1 + \cdots + t_k$, each t_i being either a variable (one of the U_i-s) or a constant. There are different interpretations of the U_i variables in the above definition that affect their meaning. When the variables are interpreted over graph vertices (hyperedges), the defined set of graphs is named *VR-equational* (respectively *HR-equational*).

The tight relationship between equational sets and context free graph grammars was established in [10], as follows.

Proposition 3.4. *(See [10].) A set of graphs is VR-equational* (*HR-equational*) *if and only if it is generated by some VR* (*HR*) *graph grammar.*

Consequently the set of process refinements defined by a BPQL$_{spec}$ system is *VR-equational* (and thus also *HR-equational*) as well. From now on we shall refer to the more general notion of *VR-equational* simply as *equational*. From the above discussion it follows that,

Proposition 3.5. $\text{BPQL}_{spec} \sim SRSM \subset HR \subset VR \sim Equational.$

The proposition above constitutes that $\text{BPQL}_{spec} \subset$ HR, through the equivalence of BPQL_{spec} and *SRSM*. We can also show a direct and constructive reduction from BPQL_{spec} to HR grammars. This reduction will be used in our algorithm given in Section 4, where we use some results on HR grammars. Hence the importance of a direct construction, given as the proof of the following lemma.

Lemma 3.6. *Every BP system can be translated into an equivalent HR graph grammar with a size linear in the size of the original BP.*

Proof. The simple transformation transforms the labeled nodes into labeled edges, and thus obtain a particular case of HR graph grammar where the hyper-edges are actually just plain edges. More formally, for each vertex v labeled by l we generate a directed edge $e(v)$ labeled l. For every edge pointing to v in the original graph, we generate a new edge pointing to the origin of $e(v)$. For every edge with origin v, we create an edge whose origin is the target of $e(v)$. The replacement rules now apply to the edge $e(v)$ rather than to the node v. □

3.2. Query languages

On comparing the expressive power of query languages. When we compare the expressive power of two query languages we must set some ground rules for reductions, otherwise one can always conduct the trivial reduction of solving the query, encoding the solution within the structure, and then reduce the query to any logic. For all of the following definitions, denote *Graphs* as the (infinite) domain of all graphs.

Definition 3.7. A reduction R is a function $Graphs \times L_1 \to Graphs \times L_2$ where L_1, L_2 are two different logics over graphs.

Definition 3.8. R is defined as a structure-independent reduction if it can be decomposed into two reductions $R_1 : Graphs \to Graphs$, $R_2 : L_1 \to L_2$ such that for all graphs g and formulas f, $R(g, f) = (R_1(g), R_2(f))$. If there exists such decomposition where $R_1 = I$ (the identity function), then the reduction is structure-preserving.

Definition 3.9. If a logic L_1 can be reduced using a structure preserving reduction to a logic L_2, we say that L_1 is partial to L_2, or equivalently that L_1 is expressible by L_2.

A simple example of a reduction that is structure-independent but not structure-preserving can be seen when handling the concept of transitive edges. Receiving as an input to the reduction both the query and the specification, one can generate a 'transitive closure' of the specification, namely for every two nodes (u, v), if a path from u to v exists, then a specially labeled edge is generated, connecting u to v. We can then replace the transitive edges in the query by an ordinary edge, labeled by the new special label. Similar construction can handle regular expressions over the edges. However, we shall not use such reductions; all of our query languages reductions are structure-preserving. I.e., we show *expressibility* relations between languages.

Common models. When considering query languages, there exist two main groups, categorized by their invariance to bisimulation. All modal logics, and specifically the temporal logics commonly used for program verification, such as *LTL* and *CTL**, are bisimulation invariant, whereas other languages such as First and Second Order logics are not. We consider them in turn below.

As a basis for comparison, we will consider in this subsection *Boolean* BPQL queries over *flat* BPs (graphs) and compare them to several common query languages/logics over finite graphs. So, whenever we use here the term "BPQL query" we mean it in the above sense. This will prove useful in the following section, when considering general BPQL queries over nested BPs.

***CTL** and μ-calculus.** As mentioned above, the temporal logics commonly used for program verification are invariant to bisimulation. They thus fail to capture isoBPQL and homBPQL, but are natural candidates to express bisBPQL. All temporal logic formalisms [19] contain, in addition to the regular connectives, modalities that refer to time. The formalisms differ in their concept of time (either linear or branching), and in the specific modalities. *LTL* (Linear Temporal Logic) regards time as a straight line, and contains modalities such as N (next), F (Finally), U (Until), etc. *CTL** extends *LTL*, considering time as branching, where there is more than one possibility for the future. Thus it contains the additional modalities of E (exists a path), and A (for all paths). μ-calculus is a temporal logic with extended expressive power. It contains the N, E, A operators along with the least fix point and the greatest fix-point operators (denoted by ν and μ respectively), which allow recursion. By applying the fix-point operators over the basic ones, the implementation of F, U, as well as additional operators not expressible in *CTL**, can be obtained.

Proposition 3.10. *bisBPQL queries are expressible in μ-calculus, with the formula size linear in the size of the query graph.*

First and Second Order logics. The temporal logics considered above fail to capture isoBPQL and homBPQL (see Lemma 3.13 below). The BP patterns used in homBPQL queries test for the (in)existence of certain nodes in the graph, and require the nodes to be (or not) connected by paths whose shape confirm to some regular languages. The isoBPQL variant further requires the nodes to be distinct. As in the case of XML query languages (and the tree pattern used in them), such patterns can be naturally expressed by FO(TC)[2] or MSO,[3] but not in FO. Indeed we will see in the next section that hom/isoBPQL patterns can be expressed by a fairly simple conjunctive subclass of FO(TC), with the size of the FO(TC) formula being linear in the size of the given BP pattern.

The following lemma establishes the connection between modal and non-modal logics.

Lemma 3.11. *(See [23].) μ-calculus is equivalent to the bisimulation-invariant part of MSO.*

From this and Proposition 3.10 above it naturally follows that MSO can also express bisBPQL. However the converse is incorrect, as follows.

Definition 3.12. A query language L over graph structures is defined to be **bisimulation-invariant** if for every formula $f \in L$ and for every two bisimilar graphs G, G', $G \models f \Leftrightarrow G' \models f$.

All temporal logics, and specifically *μ-calculus* and *CTL** are bisimulation-invariant.

Lemma 3.13. *homBPQL and isoBPQL cannot be expressed by any bisimulation-invariant language.*

Proof. The example in Fig. 4 above shows that (sub)graph isomorphism queries distinct between two bisimilar graphs. I.e., there is a query graph Q (the query in the figure) and two bisimilar graphs G, G' (BP1, BP2 in the figure), such that G is isomorphic to Q (and thus the answer for the query "Does a subgraph of G isomorphic to Q exist?" is true), and on the other hand G' is not isomorphic to Q (and thus for G', the query result is false). Naturally, the same holds for homomorphism. Consequently, any query language that allow expressing a subgraph homomorphism/isomorphism query is not bisimulation invariant and cannot be expressed by any bisimulation-invariant logic. □

Recognizable sets. Another formalism that captures hom-, iso-, and bisBPQL patterns is Recognizable Sets [27]. Recognizable Sets are an extension of the notion of regular languages, which are defined for strings, for general structures (domains). It is defined as a mapping from the general domain D into some finite domain S (intuitively the states), including a domain $F \subseteq S$ (intuitively representing the accepting states) such that the recognizable set $R \subseteq D$ is mapped into F. (For a formal definition see [27].)

The relationship between Recognizable Sets and the logics discussed above follows from the following lemma from [10].

Lemma 3.14. *(See [10].) For every graph property expressible by MSO, the set of graphs satisfying the property is a Recognizable Set.*

Note that recognizable does not necessarily mean computable. For instance, interpreted over the set of all graphs there exists a recognizable set that it is uncomputable, as implied by the lemma above and the undecidability results of [34]. However, for graphs generated by a context free graph grammar, it is indeed decidable and computable [10]. A consequent of Lemma 3.14 and the expressibility of BPQL queries by MSO:

Proposition 3.15. *For a BPQL system s, the set of process refinements for which a given Boolean homBPQL/isoBPQL/bisBPQL query returns a positive answer is a Recognizable Set.*

3.3. Query evaluation

To conclude this section, let us consider query evaluation. Ideally, one could derive a query evaluation algorithm for BPQL by adapting an evaluation algorithm for some equivalent or more powerful language on an equivalent or more powerful model. The two most relevant algorithms that we had found in the literature differ in their invariance to bisimulation and their complexity: The first is bisimulation invariant, hence can be used only to evaluate bisBPQL queries. The second can be used to evaluate all types of queries but its complexity is extremely high (non-elementary in the size of the query).

Algorithm 1. We have shown above that BPQL_{spec} can be expressed by SRSMs. We have also shown that BPQL_{query} patterns can be expressed as *CTL** formulas. Research on RSMs [1,6] suggests model checking techniques to evaluate *CTL** formula over SRSMs, with complexity $O(r*2^f)$, where r and f are the size of the SRSM and the formula, resp. Due to the exponential

[2] First Order Logic augmented with a Transitive Closure operator.

[3] Monadic Second Order Logic.

increase in the formula size, upon translation from BPQL_{query} to *CTL**, this yields, for a BPQL system s and a query q, a query evaluation algorithm with complexity $O(|s| * 2^{(2^{|q|})})$.

Algorithm 2. An alternative approach is to rely on the following lemma showed by Courcelle in [10]:

Lemma 3.16. *(See [10].) The intersection of a Recognizable Set with an Equational Set is an Equational Set.*

The intersection algorithm described in [10], coupled with Propositions 3.5 and 3.15 above, and the fact that emptiness is decidable for Equational sets [10], provides an algorithm for evaluating Boolean BPQL queries. This result, however, although of theoretical interest, does not suggest a practical evaluation algorithm for BPQL: The complexity of the construction in [10], though FPT (see Section 5), is non-elementary in the size of the query (i.e. $2^{2^{\cdot^{\cdot^{2^2}}}}$ with the size of the exponent tower being $O(|Q|)$).

Clearly, a more direct (and more efficient) approach is required here. As our query language is weaker than the full MSO, such algorithm is possible, and we propose it next. In particular, our algorithm is applicable in a uniform manner to all the iso/hom/bisBPQL variants, as well as to both Boolean and explanatory queries.

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 by *X-match*, that given a single BP pattern $\hat{p}$ and some BP p, computes 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 [4]. 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. For better readability, we explain the algorithm gradually. We start with an informal presentation – First, we sketch here informally the boolean version of our algorithm, then explain how to obtain its explanatory version. We later proceed to the formal specifics of the algorithm.

Embedding a single pattern. We start by explaining how to find, given a system s, a BP p and a BP pattern $\hat{p}$ (possibly including regular path expressions and negation), 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 this steps below.

Grammar We first construct a graph grammar for the system s, as explained in the previous section. We use the result of [26] 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 correspond precisely to the 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. $Path_R(n, m)$ holds iff there is a path from node n to node m where the sequence of labels on the path forms a word in the regular language R. In general, each pattern $\hat{p}$ can be expressed as a conjunction of three formulas $F_{\hat{p}} = f_1 \wedge f_2 \wedge f_3$ where f_1 is a conjunction of label predicates, f_2 is a conjunction of path predicates, and f_3 is a universally quantified formula containing conjunction of negated node, edge, and path predicates. Thus f_1 and f_2 handle the positive part of the query, where f_3 represents its negative part.

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. [2]. 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.

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 split F into three formulas denoted by 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[4] 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[5] 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 more tricky: their associated regular expressions are split into all possible 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}). The details can be found in Appendix A. Finally, the universally quantified formulas in F are split in a similar manner.

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 correctness of the algorithm appears in Appendix A, following its formal description. 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 [16,21].) A worst case complexity for the oracles is thus $O(p^{\hat{p}})$. However, using Database and Verification optimization techniques, this is typically much lower in practice [25].

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

The following theorem gives an analysis of the algorithm's complexity.

Theorem 5.1. *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 $O(|s|^2 \times c^{|q|} \times O(X\text{-}match(|s|, |q|)))$, where c is a constant.*

Proof. The complexity of the algorithm is computed by counting the number of possible query (formula) decompositions being generated. Note that all decompositions are applied to formulas that represent some sub-patterns of the original query, and in each stage the decomposition is into a constant (denote c_1) number of parts. Thus we have at most $O(c_1^{|q|})$ decompositions to consider at each stage. The number of stages is bounded by the number of the constructed non-terminals. Those are pairs (N, F) where N is a non-terminal in the grammar representing the system s and F is one of the conjunctive sub-formulas. The number of non-terminals N in the grammar of s is at most $|s|^2$ (the power of 2 is caused by the "normalization" of the grammar done to obtain graphs with at most two non-terminals). The number of formulas F is, hence, bounded by $O(c_2^{|q|})$ for some constant c_2. Finally, at each step, for pair (N, F), we apply the X-match oracle. This yields overall $O(|s|^2 \times c^{|q|} \times O(X\text{-}match(|s|, |q|)))$. □

Thus, the algorithm is polynomial in the size of the system s 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

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

[5] 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 $f_1 \wedge f_2$.

is NP-hard in the size of the query, is it evident that testing if the answer to an 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 5.2.

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. *The above holds even if the query does not use negation and regular path expressions.*
3. *For homBPQL and bisBPQL, the above holds even if, furthermore, all the call nodes in the system and the query have only one possible implementation.*[6]

It is open if (3) holds also for isoBPQL.

Proof. We start by giving the proof for parts (1) and (2) of the theorem, obtained through a simple reduction, and then proceed to proving part (3), using a more complicated reduction.

Parts (1) and (2). To prove parts (1) and (2) of the theorem, we define a simple class of graphs, namely graphs that are 'tree-like'. Essentially, these graphs are 'almost' directed trees, but with their leaves all connected to a single node, which in turn may be connected to another single node. The formal definition follows.

Definition 5.3. A directed graph $G = (V, E)$ is 'tree-like' if one of the following holds.

1. There exists a unique node $end(G) \in V$, such that (a) $T = (V - end(G), E - \bigcup_{w \in V, e=(w,end(G))}(e))$ is a directed tree, and (b) for every leaf v of T, $(v, end(G)) \in E$.
2. There exists $end_2(G) \in V$ such that (a) $G' = (V - end_2(G), E - \bigcup_{w \in V, e=(w,end_2(G))}(e))$ satisfies (1), and (b) the only edge in G having $end_2(G)$ as target is $(end(G), end_2(G))$.

For the finite case of trees, sub-tree homomorphism (along with transitive edges) is decided in polynomial time, as the algorithm of [22] for querying Core XPath over XML trees is of complexity $O(|S| * |Q|)$. This algorithm can be easily adapted to tree-like graphs and patterns, as the 'body' of the pattern (all nodes except the end nodes) can be embedded within the body of the graph. To find an extension of these embeddings that also include the end nodes, we only need to make sure that all nodes in the pattern that participate in the embedding are indeed connected to the nodes that relate (through the embedding) to the 'end' node. The node that relate to the 'end' node should be verified to be connected to the node that relate to the 'end_2' node. We shall now consider BP and BP patterns that are all tree-like. Thus, we can use the finite-case oracle within our algorithm to obtain an $O(|S|^2 * 2^Q)$ algorithm. The next lemma shows that the problem is NP-hard, even in this restricted case.

Lemma 5.4. *BPQL is NP-hard even when all BP graphs are restricted to be tree-like.*

Proof. We prove the NP-hardness using a reduction from 3-SAT, as follows.

Reduction. Given a Conjunctive Normal Form formula F, with variables $\{X_1, \ldots X_n\}$ we generate an instance of specification and query (S, Q) as shown in Figs. 5, 6, 7. The idea is to create a non-terminal associated with each variable of the formula (Fig. 5). This non-terminal can derive two different trees, which are the two BPs depicted in Fig. 6. I.e., for all i, the implementations of X_i are BP$_i$*True* and BP$_i$*False*. The former contains all clauses that X_i satisfies, and the latter contains all clauses that $\neg X_i$ satisfies. The query, depicted in Fig. 7, requires all clauses of the formula F to appear. An embedding thus corresponds to a 'correct' choice of either a variable or its negation, i.e. a satisfying assignment to the variables.

To formally prove that the reduction is valid, we give the following lemma.

Lemma 5.5. *There exists a non-empty embedding of the query within the specification if and only if the formula is satisfiable.*

Proof. Let E be an embedding of the query within the specification. So there must be a refinement R_i where all nodes of the query appear. This refinement was generated by choosing a subset of non-terminal generating their 'true' graphs, and another subset generating the 'false' graph. This choices corresponds exactly to a satisfying assignment – for every variable

[6] Recall that, in general, the implementation function allows to map each function name to a set of BPs which represent alternative possible implementations for the function.

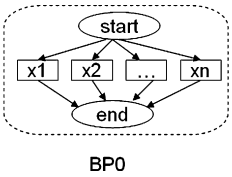

Fig. 5. Specification upper level.

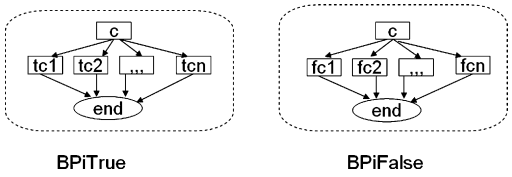

Fig. 6. Specification derivation rules.

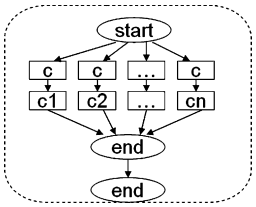

Fig. 7. Query.

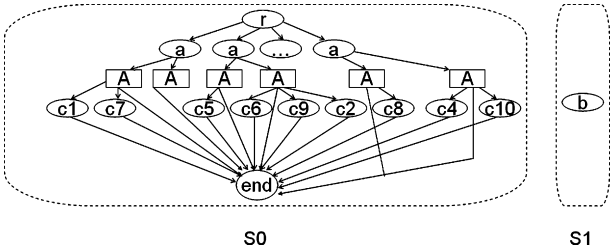

Fig. 8. Specification.

whose non-terminal generated the 'false' ('true') graph, assign 'false' ('true'). This is indeed an assignment, as every non-terminal can only generate exactly one of the 'true' or 'false' graphs, and it is satisfying as every clause node appears in the refinement, i.e. for every clause there was at least one non-terminals deriving it. The truth value assigned to the variable corresponding to this non-terminal thus satisfies this clause.

Conversely, let A be a satisfying assignment. The refinement obtained by deriving each non-terminal through its true graph if A assigns 'true' to it, and through its false graph if A assigns 'false' to it. This is indeed a well-defined refinement, since A is an assignment and thus determines a unique derivation rule for each non-terminal (true or false). There exists a homomorphism from Q to this refinement as every node clause appears at least once. This is due to the fact that the assignment A is satisfying, thus for each clause, there is at least one variable whose truth value causes the clause to be true, and its corresponding non-terminal thus derives the clause node. □

Hence the reduction is valid. As the reduction uses only trees, and no negations or regular path expressions as part of the query, parts (1) and (2) of the theorem are proved. □

Part (3). Note that in the specification used in the reduction above, each label X_i which is marking a function call node may have multiple implementations. A question rises, whether this is essential. Part (3) of the theorem states that, at least for homomorphism and bisimulation, the answer is no. To prove this part we propose another reduction which is a bit more complicated, but where each non-terminal of the specification appears as the left-hand side of a single derivation rule. However, this reduction is not valid for isoBPQL.

The specification graph, depicted in Fig. 8 is interpreted with $\tau(A) = \text{BP1}$, i.e. the implementation of A is BP1. BP0 is constructed as follows. The level directly below the root contains a set of m nodes, all labeled a, where m is the number of variables. Intuitively, each such node corresponds to a single variable. To each such node, two nodes labeled 'A' are

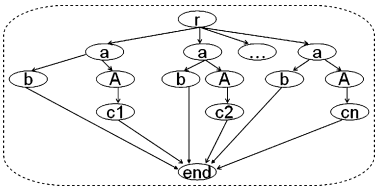

Fig. 9. Query.

connected. These correspond to truth values of the variable – one of which corresponds to the value of 'true' (positive 'A'), and the other one to the value of 'false' (negative 'A'). On the next level, the nodes are labeled by the names of the formula clauses – if a variable X_i satisfies the clause C_j, then a node labeled C_j will appear as a child node of the positive child of the node marked X_i. if $\neg X_i$ satisfies C_j then it will appear as a child node of the negative child of the node marked X_i.

The query graph, depicted in Fig. 9 intuitively requires that every clause is satisfied by some variable or by its negation. (The one whose node will be assigned by the embedding to the 'A' node connected to the node marked by the name of the clause.)

The following lemma shows the correctness of this reduction.

Lemma 5.6. *There exists a non-empty embedding of the query within the specification if and only if the formula is satisfiable.*

Proof. Assume E is an embedding. Observe that for the nodes in the lower level (labeled $C_1, \ldots, C_n$) of the query as well as in the specification, correspond to the clauses of the formula. The part of an embedding that takes care of $C_1, \ldots, C_n$ intuitively assigns each of the clauses to the variables that are responsible for satisfying it. The list of nodes labeled 'a' corresponds to the list of all variables. Each node labeled 'a' in the query graph is assigned a node (some nodes) labeled 'a' in the specification graph. This part of an embedding corresponds to a choice of variable. To determine whether the variable or its negation satisfies the clause, we distinct between the two nodes marked 'A' connected to each 'a'. One of which corresponds to the positive variable (i.e. assigning 'true' to the variable), and the other one corresponds to the negative variable. i.e. the assignment A_E is constructed as follows.

For each node n labeled 'a' in the query Q denote by Var_n the set of nodes in the specification affiliated with it in E. Now for the neighbor of n in Q, marked 'A' (and denoted p), and for each node $m \in Var_n$, p is mapped to exactly one neighbor of m (out of the two) marked by 'A'. (Because the node marked 'b' in Q should be mapped to a refinement of the other 'A'.) I.e. it is either mapped into the positive 'A' or to the negative one. Accordingly, the truth value of the variable corresponding to m is set to be true or false. We next claim that the assignment A_E constructed is indeed a satisfying assignment.

Lemma 5.7. *A_E is a satisfying assignment.*

Proof. The fact that A_E is an assignment is a direct consequence of the construction – exactly one out of 'true' or 'false' truth values is chosen for each variable. It satisfies the formula, since for each clause C_i of the formula, a node m in Q marked by C_i is assigned to some node s in the specification marked by the same C_i. Thus, its 'father' in Q, marked by A, is assigned to some node p marked by A in the specification. In turn, p is connected to s, which means that the corresponding variable (or its negation, depending on whether this is a positive or a negative 'a'), satisfies C_i. □

So far we've shown how to construct a satisfying assignment for the logic formula, given an embedding. Conversely, if A is a satisfying assignment then one can construct an embedding by relating with each node marked 'A' in the query and attached to a node labeled C_i, the node in the specification that correspond to the variable (or its negation) that was set by the assignment to true (resp. false), and (resp. their negation) appear in C_i. (If more than one such variable exists, choose one arbitrarily.) The nodes marked C_j are assigned nodes marked C_j that are connected to the nodes that their father ('A') was connected to, and similarly for those marked 'a'. Each node marked by 'b' is assigned to the node derived by the corresponding 'A' node (the one not assigned to the query 'A' node). Denote this embedding by E_A. We next show that E_A is a valid embedding for homBPQL.

Lemma 5.8. *E_A is a homBPQL embedding.*

Proof. Since A is a satisfying assignment, for each clause C_i there exists at least one variable whose assignment satisfies C_i. We can assume there exists exactly one such variable (in case there is more than one, the construction above arbitrarily selects one). Thus each node marked 'A' in the query is assigned a node in the specification that represents a variable, in the form that it appears in C_i, and thus the node is connected to the appropriate node in the specification marked with C_i. Note that since it is possible for the same variable to satisfy several clauses, the relation obtained is not an isomorphism.

The nodes marked 'b' are trivially connected as appropriate, and so are the 'start' and 'end' nodes. We obtain a homBPQL embedding. □

That concludes the correctness of Lemma 5.6. □

Thus we have obtained the correctness of Theorem 5.2. The embedding found is not only a homBPQL embedding, but is additionally a bisBPQL embedding, with each node of the query assigned a set of nodes of order 1. However, as stated above, this is not an isoBPQL embedding. It remains an open question whether similar construction (i.e. baring the restrictions described in part (3) of the theorem above) is possible for isoBPQL. □

To give a lower bound we can show that

Theorem 5.9. *The Boolean versions of homBPQL and isoBPQL are in NP (combined complexity).*

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

Lemma 5.10. *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.*

Note that an analogous lemma exists for context free *string* grammars [33]. Similarly to the case of strings, the idea of the proof below is, given a (possibly too long) refinement sequence D', one can remove all refinement steps that are applied in D' but are not essential to the embedding of the query subgraph, and obtain a sufficiently short refinement sequence D. We then show that the resultant refinement sequence is of polynomial size in terms of both the specification and the query. This is also done in a manner analogous to [33].

Proof of Lemma 5.10. For simplicity, the proof consists of several stages, as follows.

1. Looking for graph homomorphism (isomorphism) rather than the subgraph counterpart. Assuming no transitive nodes or edges.
   A graph (generated by the specification) G such that there exists homomorphism (isomorphism) to it from the query graph q, must satisfy $|nodes(G)| \leqslant |nodes(q)|$ ($|nodes(G)| = |nodes(q)|$, respectively). I.e., the size of a solution is less or equal to the size of the query. Note that this claim does not hold for bisimulation.
   Let R be the shortest refinement sequence creating a graph G such that q is homomorphic (isomorphic) to G. We aim at bounding the number of refinements in R. We distinct between two types of refinement steps – *extending* (replacing a node by a graph containing two or more nodes), and *singleton* (replacing a node by a single node). Note that these are the only possible steps. We count the number of steps of both types. The number of extending steps is obviously bounded by the size of G, and thus also by the size of q. Between each two extending steps there can be only $|s|$ singleton steps, in the shortest refinement sequence. This is due to the fact that, if the same replacement step is used twice, than we have a cycle of singleton replacement steps which is redundant, and thus there exists a shorter refinement sequence – the one that does not contain this cycle. Thus, the number of rules applications in the shortest refinement sequence is bounded by $|s| * |q|$.
2. Looking for subgraph homomorphism (isomorphism), still without transitive nodes or edges.
   Let $R' = \{R_1, R_2, \ldots, R_n\}$ be a (perhaps long) refinement sequence of s resulting in G whose subgraph is homomorphic (isomorphic) to the query q. We make the following changes to R':
   (a) Remove every refinement step that does not *contribute* to the query. Namely, if $l \to H$ is used and no node of H appears in q nor a partial refinement sequence starting from any of the nodes of H appears in R', remove $l \to H$.
   (b) If $l \to H$ is used and some nodes of H do not contribute to Q, remove these nodes.[7]
   We can now apply case (1), as we've removed all irrelevant rules, and we look for *graph* homomorphism (isomorphism).
3. Introducing transitive edges. The same principle as in (2) holds, as follows.
   Again, let R' be a refinement sequence as in (2). For each transitive edge $A \Rightarrow B$ in Q, there exist two cases. On the first case, there exist nodes N_1, N_2 in the root specification S_0 such that a (maybe indirect) refinement of N_1 contains A, a (maybe indirect) refinement of N_2 contains B and there is a path in S_0 from N_1 to N_2. In this case the number of the (contributing) refinement steps for both parts are bounded as above. In the other case, there exists a single node N

[7] This changes the refinement sequence this way, and it is now not necessarily a refinement sequence of the original specification. However, these nodes can be inserted back in the end of the sequence and all arguments hold.

in S_0 that contributes to both A, B. Such refinements correspond to the 'singleton' refinements of part (2) as they do not contribute directly to the creation of query but rather through some sequence of non-terminal replacements. Thus, similarly to part (2), there can be only as much such replacements as $|S|$.

4. Introducing transitive nodes. Transitive nodes are a restriction on possible refinement sequences. It just means that some refinement steps must take place, but all the arguments above stay intact. Again, between any two 'contributing' replacement steps, a cycle of so-called 'singleton' replacement steps may occur. This time, this cycle may be necessary because of the transitive node constraint. However, its size is bounded by the size of the specification. □

Using this lemma, the NP algorithm is simple – for each transitive compound activity in the query guess a refinement sequence for the corresponding activity in the system. Then guess a mapping from the query BP patterns to (the obtained refinements of) the system BPs and verify that it satisfies that embedding requirements. □

It is open if the same holds for bisBPQL.

Parameterized complexity. We now concern the parameterized complexity of the problem of BPQL, with the embeddings of homomorphism or bisimulation. We start with a short introduction to the area and its relevance in our case. For a survey of this field, in context of query languages, refer to [30].

The conventional approach to computational complexity refers to the size of the input as one parameter, with respect to which the complexity of the algorithm is analyzed. However, it is not always reasonable to do so. Say that a problem has two inputs, A and B, such that a, the size of A, is typically much smaller then b, the size of B. Thus, an algorithm exponential in a but polynomial in b is much better then an algorithm exponential in b and polynomial in a. In fact, an algorithm that is linear in b, though exponential in a, might be even better than an algorithm quadratic in $a+b$.

A recurring instance of this scenario often appears in databases. The database itself is typically large and may consist of hundreds of thousands of records, where a query over can typically be expressed in such succinctness, that its size can almost be considered as a constant. Thus database researchers analyze separately the complexity in terms of the *data* and of the *query* sizes.

In the BPQL setting, the size of the pattern we are looking for is typically small with respect to the entire specification. Thus parameterized complexity is relevant for BPQL. We shall see that parameterized complexity analysis produces results that are on the one hand analogous to the results obtained above using 'conventional' complexity, but on the other hand are somewhat different.

The basic idea of parameterized complexity is to consider the size of the typically small input as a parameter t, where the size of the more significant input is marked as n. Before presenting our results, we present the basics of the field of parameterized complexity, through the following definitions.

Definition 5.11 *(FPT).* An algorithm is **Fixed Parameter Tractable (FPT)** if its complexity can be expressed in the form $P(n) * f(t)$, where P is a polynomial, f is any function, n is the size of the input and t is a parameter.

Note that this definition of tractability is rather lenient. There are no restrictions on the nature of f. Thus, some algorithms are considered FPT though non-elementary in the size of t ($2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$ where the size of the tower of exponent is t) and clearly unfeasible.

The parameterized complexity parallel to polynomial time reduction is *Fixed Parameter Reduction*, defined as follows.

Definition 5.12. A fixed-parameter reduction is a Turing reduction with time complexity which is at most $f(k) * p(|X|)$, where f is an arbitrary function, p is a polynomial, $|X|$ is the size of the input and k is a parameter.

A class of problems that is often considered as the parameterized-complexity equivalent of the NP class is $W[1]$. Its definition uses the definition of the Weighted 3SAT problem, as follows.

Definition 5.13 *(Weighted 3SAT).* Given a 3SAT formula, does it have a satisfying assignment of Hamming weight that is exactly k (i.e. assigning true to k variables).

Definition 5.14 *($W[1]$).* $W[1]$ [17] is defined as the class of decision problems with input of the form (X,k) (X being the input, k being a parameter), that is fixed-parameter reducible to Weighted 3SAT.

The following lemmas establish the connection between FPT and $W[1]$.

Lemma 5.15. *(See [18].)* $FPT \subseteq W[1]$.

Lemma 5.16. *(See [18].) If* $FPT = W[1]$ *then* $NP \subseteq DTIME(2^{o(n)})$.

Hence the following conjecture,

Conjecture 5.17. *FPT* $\neq$ *W*[1].

And its immediate corollary,

Corollary 5.18. *W*[1]*-hard problems are not FPT.*

In a sense, W[1]-hardness relates to FPT as NP-hardness relates to polynomial time complexity. In a manner similar to the complexity analysis above, the parameterized hardness of BPQL is determined by the embedding chosen, and the hardness of computing this embedding over finite graphs. Below we formalize this dependency.

Lemma 5.19. *For each finite graph embedding class X such that finding if an embedding belonging to E exists, for the finite graph case, is W*[1]*-hard, then so is the corresponding X-BPQL algorithm is W*[1]*-hard as well.*

Proof. The lemma is trivially correct, as a pair of (BP specification, query) both consisting only of a single root graph with no implementations constitutes a restricted case of BPQL, and corresponds exactly to the finite case of embeddings. □

Lemma 5.20. *homBPQL, isoBPQL are W*[1]*-hard.*

Proof. In [20] it is shown that subgraph homomorphism for finite graphs is *W*[1]-hard. For subgraph isomorphism, clique is *W*[1]-hard by [20]. The simple reduction of creating a query graph of the form of a clique proves the hardness of Subgraph Isomorphism. Using Lemma 5.19, we obtain that homBPQL and isoBPQL are *W*[1]-hard. □

More interestingly, a similar result holds for the converse situation – if the finite problem is FPT then so is the corresponding problem for BPQL. This result is proved below, in Lemma 5.21. Note that this is a case where the parameterized complexity analysis differ from its 'conventional' complexity counterpart, as we have shown above that BPQL is NP-hard even for restricted cases where the finite problem of the corresponding embedding is polynomial.

Lemma 5.21. *For each finite graph embedding E such that E has an FPT algorithm O solving the finite graph case, BPQL algorithm used with O as oracle is FPT as well.*

Proof. In Theorem 5.1 above we've shown that the BPQL algorithm requires only $O(2^{|Q|})$ multiplicative factor with respect to the algorithm for the finite case. Thus, a computation of an embedding that is FPT for finite graphs remains FPT for BPs as well. □

Following is an interesting corollary of Lemma 5.21.

Corollary 5.22. *bisBPQL is FPT.*

Proof. In [20] the finite version of subgraph bisimulation is shown to be FPT. Thus, following Lemma 5.21, bisBPQL is FPT. □

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.

To guarantee 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 [8]. It would also be interesting to combine our algorithm with some existing verification techniques, e.g. [25].

Acknowledgments

This research was partially supported by the EU project Mancoosi and the Israel Science Foundation.

Appendix A. Formal algorithm

We present the full and formal description of our algorithm, along with its correctness proof, and complexity analysis. We start by defining formally the representation of BP patterns using formal logic (namely FO(TC) for structural interpretation and μ-calculus for behavioral interpretation). We then proceed to giving the formal and full specifics of the algorithm itself. We conclude by proving the correctness of the algorithm.

Logic representation. We use a slight variation of the common (see e.g. [10]) semantics and interpretation, as follows. The logic includes, as usual, atoms, variables, predicates, connectives and quantifiers, and is interpreted over some model. Following, we present each of these components.

Atoms. The set of nodes, $V = \{V_1, \ldots, V_k\}$, and paths constructed out of these nodes, $P = \{P_1, \ldots, P_m\}$.

Variables. $NodeVars = \{X_1, \ldots, X_k\}$, $PathVars = \{Z_1, \ldots, Z_m\}$ denote variables which correspond to nodes and paths (respectively) in the graphs.

Atomic predicates. We define the following atomic predicates.

1. $L_c(X_i)$, where c is a constant and X_i is a node variable. Intuitively, this predicate is satisfied if the label of the node assigned to X_i is c.
2. $Path_R(X_i, X_j, Z_t)$, where R is a regular expression, X_i and X_j are node variables, and Z_t is a path variable. Intuitively, this predicate is satisfied if Z_t is assigned a path from the node assigned to X_i to the node assigned to X_j. The formal semantics are given below.

Connectives. AND, OR, NOT, with their usual meaning.

Using these components, various logics can be defined. The logics vary in two aspects – (a) their interpretation of variables and (b) their quantifiers, as follows.

1. First Order Logic contains the quantifiers $\forall x$ (for all x) and $\exists x$ (for some x), for variables that represent single nodes, and interpreted as usual. The assignment A assigns atoms (nodes) to the free variables. FO(TC) denotes First Order Logic, augmented with a Transitive Closure operator.
2. μ-calculus includes the least fix point (ν) and greatest fix point (μ) quantifiers, with variables varying over sets of nodes. All concepts are thus interpreted over sets, with the common concept of set inclusion used as a basis for determining the 'least' and 'greatest' fix points with respect to some formula.

In each of these logics, we shall only regard a special kind of formulas, namely conjunctive formulas, which are defined as follows.

Definition A.1. A finite formula f that is written in a logic L over the domain of finite graphs, and has a set of free variables $\{X_1, X_2, \ldots, X_n\}$ which range over graph nodes and a set of free variables $\{Z_1, \ldots, Z_k\}$ ranging over paths[8] in the graph, is *conjunctive* if it can be written as the conjunction of basic formulas $f_1, \ldots, f_m, g_1, \ldots, g_s, h_1, \ldots, h_l$ as follows.

Basic formulas. We next list the structure of the basic formulas.

1. Each f_i has one free variable and is of the form $L_c(x)$. These formulas are called **node formulas**.
2. Each g_k is of the form $Path_R(X_i, X_j, Z_k)$, R being a path regular expression over the labels alphabet. These formulas have three free variables. The first two (X_i, X_j) are node variables. Z_k ranges over the range of regular expressions over the **nodes** and will intuitively contain the path from X_i to X_j. We call these formulas **edge formulas**.
3. Each h_i has no free variables. It can be any atomic formula or its negation. Thus these formulas are the **closed formulas**.
4. A formula that has a free variable may have no quantifiers.
5. A formula with no free variables can either contain only quantifiers of type $\forall$, or contain only quantifiers of type $\exists$, but not both in a single formula. A conjunction of closed formulas may be preceded by a negation.

Notations. We use the following notations for the different components of the formula.

1. Denote $\{f_i\}$ as **Nodes(f)**
2. Denote $\{g_i\}$ as **Edges(f)**
3. Denote $\{h_i\}$ as **Closed(f)**

[8] Single edges are also considered as paths.

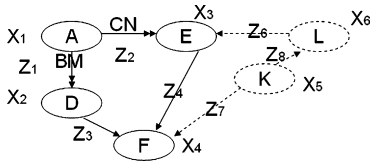

Fig. 10. Query graph.

4. Denote the set $\{f_1, \ldots, f_m, g_1, \ldots, g_s, h_1, \ldots, h_l\}$, the set of basic formulas, as $Basic(f)$
5. Denote the node variables that appear in f as $NodeVars(f)$, and the path variables as $PathVars(f)$

Where formulas are interpreted over graphs, we define an assignment of the components (nodes, edges, paths) of a graph to the variables of a formula, as follows.

Definition A.2. An assignment is an object $A(f, G)$, constructed out of two mappings. $A_n : NodeVars(f) \mapsto N(G)$, $A_E : EdgeVars(f) \mapsto Reg(N(G))$, where $N(G)$ is the set of G's nodes, and $Reg(N(G))$ is the set of all path regular expressions over the nodes of G.

Specifically, we are interested in assignments that satisfy the formula. An assignment A is satisfying for a conjunctive formula f and a finite graph G if it satisfies every clause of f, according to the following semantics.

Formulas semantics. In the following, $string(p)$ denotes the string that is obtained by replacing each node in the path p in G by its label. $Nodes(p)$ denotes the set of all nodes along p. We use here the intuitive concept of path. Its exact definition is given later on.

A node formula $L_c(X)$ is satisfied if the label of the node $A_n(X)$ is c. An edge formula $Path_R(X, Y, Z)$ is satisfied by A if,

1. $A_E(Z)$ is a path in G from the node $A_n(X)$ to the node $A_n(Y)$.
2. $string(A_E(Z)) \in R$.

Example. The query depicted in Fig. 10 contains a positive part and a negative part. The latter is required not to appear, and is marked by dashed lines. The X_i-s near nodes mark their corresponding node variables, where edges are marked by Z_i-s variables. With this allocation of variables in mind, the query translates into the following formula:

$$L_A(X_1) \wedge L_B(X_2) \wedge E^{BM}(X_1, X_2, Z_1) \wedge L_E(X_3) \wedge L_F(X_4) \wedge E^{CN}(X_1, X_3) \wedge E(X_2, X_4, Z_3) \wedge E(X_3, X_4, Z_4)$$
$$\wedge \neg\big(\exists X_5 \exists X_6 \exists Z_6 \exists Z_7 \exists Z_8 . E(X_5, X_6, Z_8) \wedge E(X_6, X_3, Z_6) \wedge E(X_5, X_4, Z_7) \wedge L_k(X_5) \wedge L_L(X_6)\big)$$

The replacement of a variable x with an atom a in a formula f is denoted by $f[X|a]$. Every free occurrence of x in f is replaced by a. Where the order of variables in $f(x_1, \ldots, x_n)$ is well defined, we use the notation $f[a_1, a_2, \ldots, a_n]$ with the meaning of $f[x_1|a_1, \ldots, x_n|a_n]$.

Given an assignment of a partial set of the components of a graph G to the variables of a formula f, we are interested in the subgraph of G that contains only the 'relevant' components. This is formalized using the following definition, which is the logic equivalent of the definition of relevance given in Section 2.

Definition A.3. For a graph G, a formula f and an assignment $A(f, G)$. The subgraph of G induced on A is denoted by $G|_A$, and is the subgraph of G obtained by $G' = (V', E')$ where $V' = (\bigcup_{x \in NodeVars(f)} A_n(x)) \cup (\bigcup_{Z \in PathVars(f)} Nodes(A_E(Z)))$, $E' = \bigcup_{Z \in PathVars(f)} Edges(A_E(Z))$.

Example. Consider the graph G depicted in Fig. 11. Evaluating the formula f corresponding to the query in Fig. 10 over G, we obtain the following assignment: $A(X_1) = N_1$, $A(X_2) = N_6$, $A(X_3) = N_7$, $A(X_4) = N_8$, $A(Z_1) = (E_1, E_2, E_5)$, $A(Z_2) = (E_3, E_4, E_6)$, $A(Z_3) = (E_8)$, $A(Z_4) = (E_7)$.

The induced subgraph $G|_A$ is depicted in Fig. 12.

Algorithm. In the algorithm below we shall consider decomposition of formulas. For that cause, we need to define first some notions that relate to decomposition of regular expressions. The first definition defines the notion of string concatenation.

Definition A.4. Given two strings S_1, S_2 their concatenation $S_1 \circ S_2$ is the string constructed out of the character sequence of S_1 followed by the character sequence of S_2.

We next define the possible operations over regular expressions. For a regular expression R the language it recognizes (i.e. the set of all strings conforming to it) is denoted by $L(R)$.

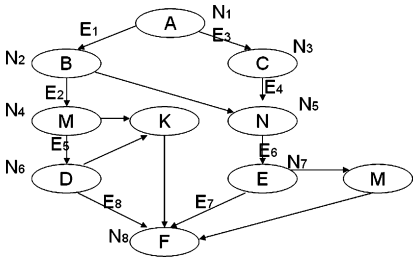

Fig. 11. Graph.

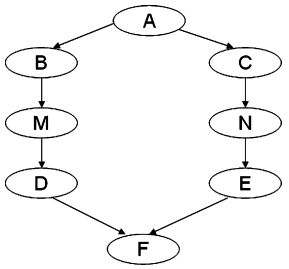

Fig. 12. Induced subgraph.

Definition A.5. Given two regular expressions R_1, R_2,

1. $R_1 \circ R_2$ is a regular expression R such that $L(R) = \{v \circ u \mid v \in L(R_1), u \in L(R_2)\}$.
2. $R_1 + R_2$ is a regular expression R such that $L(R) = L(R_1) \cup L(R_2)$.
3. R_1^* is a regular expression R such that $L(R) = \bigcup_{i=1,\ldots,\infty} L(R^i)$ where $R^i = R_1 \circ R_1 \circ \cdots \circ R_1$ (i times). R is called a starred expression.

We can now define the 3-decomposition of a regular expression R into a set of triplets of regular expressions $\{(R_1^i, R_2^i, R_3^i)\}$ where the concatenation of these triplets, in a sense, 'covers' $L(R)$. This is formalized as follows.

Definition A.6. A 3-decomposition of a regular expression R is a set of triplets of regular expressions $\{(R_1^i, R_2^i, R_3^i)\}$, such that,

1. $\forall i.\ \forall w \in L(R_1^i)\ \forall u \in L(R_2^i)\ \forall v \in L(R_3^i).\ (w \circ u \circ v) \in L(R)$.
2. $\forall r \in L(R),\ w, u, v.(w \circ u \circ v = r) \Longrightarrow (\exists i.w \in L(R_1^i) \wedge u \in L(R_2^i) \wedge v \in L(R_3^i))$.

Lemma A.7. *For every regular expression R over alphabet Σ, there exists a 3-decomposition of R that is a finite set. Such decomposition can be computed in Poly($|R|$) time.*

Proof. The regular expression decomposition algorithm is given in Algorithm 1, named *RegDecomp*. It uses an algorithm *Reg*, that given an automaton[9] A generates the regular expression $Reg(A)$ such that $L(Reg(A)) = L(A)$. I.e., A and $Reg(A)$ recognize the same language. *RegDecomp* generates the 3-decomposition of a regular expression R using the automaton $A(R) = (S, start, \delta, end)$ that corresponds to R. It assumes that $A(R)$ has unique start and end states.[10] *RegDecomp* chooses two states S_1, S_2 out of S, and generates the following automata:

1. $A_{(start,S_1)}(R) = (S, start, \delta, S_1)$
2. $A_{(S_1,S_2)}(R) = (S, S_1, \delta, S_2)$
3. $A_{(S_2,end)}(R) = (S, S_2, \delta, end)$

We claim that the set of regular expressions triplets corresponding to these triplets of automata forms a 3-decomposition. For each triplets of words u, v, w s.t. $u \in L(A_{(start,S_1)}(R))$, $v \in L(A_{(S_1,S_2)}(R))$, $w \in L(A_{(S_2,end)}(R))$, it is clear that the execution of $A(R)$ over u ends in S_1, an execution of $A(R)$, starting with S_1, over v, ends in S_2 and an execution of $A(R)$ over w starting in S_2 ends in *end*. Thus the execution of $A(R)$ over $u \circ v \circ w$ starting in *start* ends in *end*, i.e. $u \circ v \circ w \in L(R)$.

[9] When using the term 'automaton' we mean Non-deterministic Finite Automaton, unless stated otherwise.

[10] In case it has more than one end state, a unique end state can be generated, with the old end states being connected to it through ϵ transitions.

Algorithm 1: Regular expression decomposition.

$D := \phi$;
Generate the finite state automaton $A(R) = (S, start, \delta, end)$ s.t. $L(A(R)) = L(R)$;
/* We generate 3 automata, all with the same states and transition relation as the original one but with different start and end states */
foreach $(S_1, S_2) \in S \times S$ **do**
  /* The start state of the first automaton is the original start state, and its end state is S_1 */
  $A_{(start,S_1)}(R) := (S, start, \delta, S_1)$;
  $R_{(start,S_1)} := Reg(A_{(start,S_1)}(R))$;
  /* The start state of the second automaton is S_1, and its end state is S_2 */
  $A_{(S_1,S_2)}(R) := (S, S_1, \delta, S_2)$;
  $R_{(S_1,S_2)} := Reg(A_{(S_1,S_2)}(R))$;
  /* The start state of the third automaton is S_2, and its end state is the original end state */
  $A_{(S_2,end)}(R) := (S, S_2, \delta, end)$;
  $R_{(S_2,end)} := Reg(A_{(S_2,end)}(R))$;
  if $L(R_{(start,S_1)}) \neq \phi$ and $L(R_{(S_1,S_2)}) \neq \phi$ and $L(R_{(S_2,end)}) \neq \phi$ **then**
    $D := D \cup (R_{(start,S_1)}, R_{(S_1,S_2)}, R_{(S_2,end)})$;
  end
end

Conversely, for each $r \in L(R)$ and its decomposition into u, v, w, the execution of $A(R)$ over u, starting in *start* results in some S'_1, its execution over v starting in S'_1 results in some S'_2, and its execution over w starting in S'_2 results in *end* (as the execution over $u \circ v \circ w$ starting in *start* ends in *end*). Thus, the regular expressions that were generated by choosing S'_1 as its S_1 and S'_2 as its S_2 satisfy u, v and w.

The time complexity of the construction is quadratic in the size of the automaton $A(R)$ and thus in the size of R ($A(R)$ is generated in linear time). The quadratic factor is due to the choice of all pairs of states for (S_1, S_2). □

Algorithm. We can now give the formal description of the grammar, formula and the result construction.

Notation. For two conjunctive formulas f and g we denote the formula h which includes all clauses of f and g (h is the conjunction of both) by $f + g$.

Grammar. The construction of the grammar out of the input BP specification is done in two stages, as follows.

1. For each label A of a function call node, add a unique label L_A and a rule $A \to L_A$.
2. Transform the BP into a VR grammar in a normal form, where each derivation rule contains at most two non-terminals in the right-hand side graph. This transformation is done in 3 stages:
   2.1. Transform the BP into an HR grammar.
   2.2. Transform the HR grammar into a normal form, using the algorithm of [26].
   2.3. Transform the normal form HR grammar into a normal form VR grammar, by replacing each edge with a node and vice versa.

   The size of the resultant grammar is quadratic w.r.t. the size of the original BP, where the transformation to normal form is the cause of this increase of size (the other transformations are clearly linear).

Formula. Given the definitions above, the translation of a BP pattern into a conjunctive formula is straightforward. We generate one basic formula at a time, and the resultant formula is the conjunction of all those basic formulas. For each node of the pattern, a distinct variable is generated. If the node is labeled by c and assigned a variable x, then $L_c(x)$ is generated. For a regular edge between a node labeled x and a node labeled y, we generate a new variable z and the formula $E(x, y, z)$. If the edge is indirect and labeled with a regular expression R, the formula $Path_R(x, y, z)$ is generated. Note that we do not consider composite nodes here, as extension of the algorithm to consider full queries rather than a single BP pattern is discussed separately below.

Algorithm signature. We now present the signature of the algorithm given below. The algorithm uses an X-match oracle, which is an all-SAT(G, f) algorithm solving the 'finite version', i.e. taking as input a finite graph G and a conjunctive formula f as input and producing all satisfying assignments for f over G.

Input. G_p, a graph grammar, f, a conjunctive formula, and an X-match oracle, as described above.

Output. We give two versions of the algorithm. The first version solves the **boolean version** of the BPQL problem – i.e., whether an embedding exists. The second version is the **projection** version, where the result encodes the embeddings of the query in refinements of the specification.

Result construction algorithm. Next, we present the formal algorithm for constructing the result grammar.

Algorithm 2: Generating decompositions.

foreach ***DISTINCT*** *nodes decomposition* f_{N_1}, f_g, f_{N_2} *s.t.* $Nodes(f_{N_1}) \cup Nodes(f_g) \cup Nodes(f_{N_2}) = Nodes(F_j)$ **do**
  foreach $e \in Edges(F_j)$ *s.t. R is the regular expression of e* **do**
    Denote X_e, Y_e, Z_e as the free variables of e;
    $SubDivisions(R) := RegDecomp(R)$;
    foreach $(R_0, R_1, R_2) \in SubDivisions(R)$ **do**
      if X_e *and* Y_e *both appear as free variables in* $Nodes(k)$ *for some unique* $k \in \{f_{N_1}, f_g, f_{N_2}\}$ **then**
        $k := k + e$;
      end
      if X_e *appears in* f_{N_1} *and* Y_e *appears in* f_{N_2} **then**
        $f_g := f_g + e^{R_1}[X_e|N_1, Y_e|N_2, Z_e]$;
        $f_{N_1} := f_{N_1} + e^{R_0}[X_e, End_{N_1}, Z'_e]$;
        $f_{N_2} := f_{N_2} + e^{R_2}[Start_{N_2}, Y_e, Z''_e]$;
      end
      if X_e *appears in* f_g *and* Y_e *appears in* f_{N_1} **then**
        $f_g := f_g + e^{R_0 \circ R_1}[Y_e|N_1]$;
        $f_{N_1} := f_{N_1} + e^{R_2}[Start_{N_1}, Y_e, Z'_e]$;
      end
    end
  end
  $ClosedExistentialSet = \Phi$;
  foreach u *in* $Closed(f_j)$ **do**
    if u *is universal* **then**
      Add u to all formulas;
    end
    if u *is existential* **then**
      Add u to ClosedExistentialSet;
    end
  end
  Find all formulas from ClosedExistentialSet that are satisfied on G_i, remove those from the set;
  Try all divisions of the remaining set into sets S_{N_1}, S_{N_2}, and for each division
  $f_{N_1} := f_{N_1} + S_{N_1}$;
  $f_{N_2} := f_{N_2} + S_{N_2}$;
  Apply the oracle to find the set of satisfying assignments for f_g on g. These are denoted $\{S_1, \ldots, S_k\}$;
  if *none found* **then**
    move on to the next triplet;
    else generate for each l, a new rule, $(N_i, F_j) \to G_l$, s.t. G_l is obtained from $g|_{S_l}$ by replacing non-terminal N_i with (N_1, F_{N_1})
  end
end

Non-terminals. Let $N = \{R, N_1, \ldots, N_n\}$ be the non-terminals of G_p, R marking the initial non-terminal, and let $P(F_{\hat{p}}) = \{F_1, \ldots, F_m\}$ be the set of *conjunctive sub-formulas* of $F_{\hat{p}}$, defined as follows. Each F_i in $P(F_{\hat{p}})$ is the conjunctive formula generated by picking a set from $P(Basic(F_{\hat{p}}))$ (the power set of the set of basic formulas of f) and creating the formula which is the conjunction of all formulas within the set. The non-terminals set of G_r is $\bigcup_{N_i \in N, F_j \in P(F_{\hat{p}})}\{(N_i, F_j)\}$. Its initial (root) non-terminal is $(R, F_{\hat{p}})$.

Derivation rules. For each non-terminal (N_i, F_j), assume $N_i \to G_i$ is a rule of G_p. G_i is a graph that may:

1. Consist only of terminals or,
2. Consist of both terminals and non-terminals.

Case 1. In this case, G_i is just a node-labeled finite graph. Thus, apply the oracle to find all satisfying assignments of F_j on G_i. The new rules are $(N_i, F_j) \to G_i|_{A_l}$ for each satisfying assignment A_l.

If no satisfying assignment exists then in this case no rules are generated that derive the non-terminal (N_i, F_j). We mark (N_i, F_j) by F (failure), to be removed in the clean-up stage (described below).

Case 2. In this case G_i contains two non-terminals N_1, N_2 (as the grammar is in normal form). We thus need to consider decompositions of the query, so that each part of it is assigned for each non-terminal.

The algorithm that generates these decompositions is given in Algorithm 2, which will be referred to as *decomp*. 'Start' and 'End' refer to the Start and End nodes of a BP, respectively.

The cleanup stage of the algorithm removes non-terminals that do not contribute to the generation of any refinements (i.e. lead to dead-end). This stage is given in Algorithm 3.

Algorithm 3: CleanUp.

```
1 /* Remove failed non-terminals */
2 while A ≠ φ do
3 |   A := the set of all non-terminals marked with F or deriving a graph that includes a non-terminal that was removed;
4 |   Remove A;
5 end
6 Remove all non-terminals inaccessible from the root non-terminal;
```

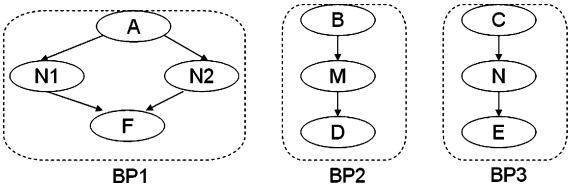

Fig. 13. Specification.

The last part of the algorithm depends upon the type of result we are interested in. For the boolean variant of the algorithm we only check if the resultant grammar is empty. To obtain the set of all results as defined above we use a variation of the decomposition algorithm. We omit the pre-processing part, and conduct the decomposition of the formula to $n+1$ parts in each stage, where n is the number of non-terminals in the right-hand side of a derivation rule handled.

Example. We consider the BP in Fig. 13 (interpreted with BP2 being the implementation of N1 and BP3 being the implementation of N2) and the formula that represents the query in Fig. 10. The algorithm tries all decomposition of nodes into 3 sets. It fails to find an assignment until it chooses the 'correct' decomposition – the nodes marked A and F in one set, the node marked D in the second set, and the node marked E in the third set. In this case we generate two new non-terminals, and turn to find a satisfying assignment within the implementations of N1, N2. In this simple example, we find an assignment for the node marked D and the indirect edge marked BM in the implementation of N1, and for the node marked E and the indirect edge marked CN in the implementation of N2.

Note. Where a BPQL query is viewed as a logic formula, our algorithm can be related to as a (very specific) *theorem-prover* [28]. In the area of automatic theorem proving, it is common to prove theorems that range over several theories (for exact definitions refer to [28]). In such case the formula need to be *decomposed* into its sub-formulas, each sub-formula holding only terms of its own theory. Every sub-formula can then be 'fed' into a theory-specific theorem prover, and the 'connecting parts', which are sub-formulas that connect formulas of different theories, are proven separately. This method is known as the Nelson–Oppen method [28]. Essentially, our algorithm operates very much in the spirit of the Nelson–Oppen method. The query (without call nodes; these are considered later on), ranges over the domain of finite graphs, where the specification is a graph *grammar*. Thus the formula is decomposed into parts (in this case each part is over the same domain – finite graphs), and each part is matched separately. The connecting parts of the formula here are the edges referred to in the algorithm as the 'connecting edges'.

Correctness. We now show the correctness of the algorithm presented above.

Theorem A.8. *Let G_p be the original grammar, let f be the formula, and let G_r be the grammar generated by the algorithm.*

A finite graph G_2 is generated by G_r if and only if $G_2 = G'|_{S_i}$, for some finite graph G' generated by G_p and some satisfying assignment S_i of f on G'. I.e., G_2 consists of the nodes and edges of G' that are relevant.

Proof. We give a lemma that is sufficient in order to show the algorithm correctness. Informally, we show that when decomposing a formula F into three formulas F_{N_1}, F_g, F_{N_2} as done in Algorithm 2, decomposing a graph g into its conjunctive decomposition of size 3 (defined below), and finding a satisfying assignment for each of the three formulas within a subgraph, their combination constitutes a satisfying assignment for F with respect to g. Moreover, the combination of the relevant parts of the subgraphs, with respect to the satisfying assignments, constitutes the relevant part of g with respect to its satisfying assignment. The derivation sequence that generates graphs in the new grammar corresponds exactly to such inductive decomposition. Thus, and as the invariant proved in the lemma holds throughout the grammar construction, the theorem holds. It remains to formally give and prove the lemma described above. We start with the following definition of graph decomposition that will be used in Lemma A.10 below.

Definition A.9. Let G_p be a VR normalized graph grammar as above. Let K be a graph containing two non-terminals N_1 and N_2, and g be a graph generated by K through a single replacement of N_1 and a single replacement of N_2 (using rules of G_p). The *conjunctive decomposition* of g (with respect to K, G_p) is a triplet of graphs (G_{N_1}, G_g, G_{N_2}), defined as follows.

G_{N_1} is defined as all nodes and edges of g that did not appear in K and were added through N_1.
G_g is defined as all nodes and edges of g that also appeared in K.
G_{N_2} is defined as all nodes and edges of g that did not appear in K and were added through N_2.

Lemma A.10. *For every conjunctive formula f and every finite and connected graph G, that is derived directly from some graph K with respect to a VR graph grammar G_p, if f is decomposed into conjunctive formulas f_{N_1}, f_g, f_{N_2} as in Algorithm 2 and there exists a corresponding conjunctive decomposition of G (G_{N_1}, G_g, G_{N_2}) such that each graph G_i satisfies its corresponding f_i, then G satisfies f.*

Conversely, every such decomposition is found.

Moreover, the nodes and edges of G that are relevant to satisfying assignments for f are exactly the nodes and edges of the G_i-s that are relevant to satisfying assignments for the f_i-s.

Proof. Denote the satisfying assignment of G_i over f_i by A_i. We construct an assignment A as follows. For every node predicate $L_c(x)$ of f that appears in f_i, $A(x) = A_i(x)$.

For every edge predicate $Path_R(x, y, z)$ of f such that $L_{c_1}(x)$ appears in f_i and $L_{c_2}(y)$ appears in f_j, the algorithm decomposed R into 3 regular expressions R_0, R_1, R_2. By Lemma A.7, for every $u \in R_0$, $v \in R_1$, $w \in R_2$, $(u \circ v \circ w) \in R$.

We define $A(z) = A_i(z_i) \circ A_j(z_j) \circ A_k(z_k)$, where $\circ$ denotes concatenation. This holds for the 'appropriate' ordering of $0, 1, 2$ into i, j, k, according to the location of x and y in f_g, f_{N_1} or f_{N_2}.

A is clearly an assignment, as for node variables, each node predicate appears in exactly one formula, thus A assigns to it a single value. For path variables, the construction chooses a single set of terms and thus assign a unique value to each variable.

A is a satisfying assignment for f. For node predicates, each predicate appears in one of the f_i's, thus satisfied by A_i and thus by A. For edge predicates $Path_R(x, y, z)$, those are satisfied by construction (using Lemma A.7, as explained above). z is assigned a path from $A(x)$ to $A(y)$, as $A_i(z_i)$ is a path from $A_i(s_i) = A(s_i)$ to $A_i(t_i) = A(t_i)$, $s_0 = x$, $t_0 = y$, and $t_i = s_{i+1}$ (up to identification of start an exit nodes of a graph derived by N_i, with the node N_i itself).[11]

$A(z)$ satisfies the regular expression R, as $A(z) = A_i(z_i) \circ A_j(z_j) \circ A_k(z_k)$, each $A_l(z_l)$ satisfies the appropriate R_i according to its location, and $R = R_0 \circ R_1 \circ R_2$.

Conversely, it is clear that the algorithm generates all decompositions of the node predicates set, and test all possibilities. As for the edges predicates, we showed in Lemma A.7 that all decompositions of the regular expression into 3 parts are generated. Thus every decomposition of a path (if the path corresponds to the regular expression) into 3 sub-paths is tested.

The 'moreover' part of the lemma is obtained through the construction. Denote the set of A_i assignments by S_A. When constructing the assignment A we chose, for every node variable, the node in G_i assigned to it by A_i (for some A_i in S_A). For every path variable z, the construction of $A(z)$ is composed out of components that are all assigned by some assignments in S_A. Thus all edges that appear in the path $A(z)$ are edges that appear in the assignments in S_A, and thus these are all edges that are relevant to some G_i w.r.t. A_i. □

This completes the proof of the algorithm correctness, as the invariant holds at each stage. □

Evaluating a full query Algorithm 2 considered the embedding of a single BP pattern. Its extension to an algorithm that evaluates a full BPQL query is given in Algorithm 4.

Theorem A.11. *Given a BP system s and an X-BPQL query q, the BP system s' generated by Algorithm 4 is empty if and only if there is no X-embedding of q in s.*

Proof. We show that R'_i is a refinement of s' if and only if there exists a refinement R_i of s and an embedding E such that R'_i is isomorphic to the subgraph of R_i which is relevant to the embedding.

To show this, we use Theorem A.8, namely the correctness of the decomposition algorithm. We use a simple induction on t, the number of replacements needed to take place, starting from the root process of s', to obtain a refinement R'_i. For $t = 0$, the algorithm just uses the X-match oracle and so its correctness follows immediately from the correctness of the oracle. Assume correctness for $t = k$. I.e., every refinement R' of s', which is of replacement depth k, is isomorphic to the subgraph of R which is relevant to some embedding E_1, such that R is a refinement of s. We now look at another refinement step. For each call node c in R' and a node n of the query associated with it (according to E_1), if n is direct, then the construction of the BP s' adds to the implementations set of c in s' an X-match oracle embedding of $\tau(n)$ within the implementation of c in s. By the correctness of the X-match oracle, this embedding is an X-embedding. If n is indirect, then the implementation that is generated is an embedding obtained by the decomposition algorithm applied on the specification rooted in c and the implementation of n in q. Following the correctness of the decomposition algorithm, this is indeed an X-embedding. Thus, any implementation chosen to continue the refinement process maintains the invariant. This completes the correctness proof for Algorithm 4. □

[11] Since the grammar is a single-entry single-exit, edges whose target is N_i will become, upon replacement, edges whose target is $Start(N_i)$, and similarly for $End(N_i)$.

Algorithm 4: Handling multiple BP patterns.

```
1  HandledPairs := φ;
2  /* This will hold all pairs of (BP, BP pattern) that were already handled, along with the new root of the embedding BP */
3  WorkSet := {(S0, Q0, transitive, location)};
4  /* The work set contains pairs of (BP, BP pattern) that are to be matched, along with a boolean determining if the match is to be done transitively
   (in case of indirect calls) or not, and the location where the result of the embedding should be put (replacing a node within a higher-level
   specification) */
5  while WorkSet ≠ φ do
6      Pick a pair (s, q, transitive, location) out of WorkSet;
7      if (s, q) ∈ HandledPairs then
8          S' := EmbeddingRoot((s, q));
9      else
10         if transitive then
11             Apply decomp(s, q) to obtain a specification rooted at S';
12             else
13                 Apply X-match-Oracle(s, q) to obtain a specification rooted at S';
14             end
15         end
16         foreach call node n of q do
17             foreach node m of s that was associated with n do
18                 Add (m, n, IsTransitive(n), location) to WorkSet;
19             end
20         end
21     end
22 end
23 Insert S' in 'location';
24 Remove (s, q) out of WorkSet;
25 end
```

References

[1] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, M. Yannakakis, Analysis of recursive state machines, ACM Trans. Program. Lang. Syst. 27 (4) (2005).
[2] R. Alur, S. Chaudhuri, P. Madhusudan, A fixpoint calculus for local and global program flows, in: Proc. of POPL, 2006.
[3] C. Beeri, A. Eyal, S. Kamenkovich, T. Milo, Querying business processes with BP-QL (demo), in: Proc. of VLDB, 2005.
[4] C. Beeri, A. Eyal, S. Kamenkovich, T. Milo, Querying business processes, in: Proc. of VLDB, 2006.
[5] C. Beeri, A. Eyal, S. Kamenkovich, T. Milo, Monitoring business processes with queries, in: Proc. of VLDB, 2007.
[6] M. Benedikt, P. Godefroid, T. Reps, Model checking of unrestricted hierarchical state machines, in: Proc. of ICALP, 2001.
[7] Business Process Execution Language for Web Services, http://www.ibm.com/developerworks/library/ws-bpel/.
[8] P. Buneman, G. Cong, W. Fan, A. Kementsietsidis, Using partial evaluation in distributed query evaluation, in: Proc. of VLDB, 2006.
[9] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, 1999.
[10] B. Courcelle, The monadic second-order logic of graphs, Inform. and Comput. 85 (1) (1990).
[11] B. Courcelle, Recognizable sets of graphs, hypergraphs and relational structures: a survey, in: DLT '04, in: LNCS, vol. 3340, 2004.
[12] D. Deutch, T. Milo, Querying structural and behavioral properties of business processes, in: Proc. of DBPL, 2007.
[13] D. Deutch, T. Milo, Type inference and type checking for queries on execution traces, in: Proc. of VLDB, 2008.
[14] D. Deutch, T. Milo, Evaluating top-k queries over business processes, in: Proc. of ICDE, 2009.
[15] A. Deutsch, L. Sui, V. Vianu, D. Zhou, Verification of communicating data-driven web services, in: Proc. of PODS, 2006.
[16] A. Dovier, C. Piazza, The subgraph bisimulation problem, IEEE Trans. Knowl. Data Eng. 15 (4) (2003).
[17] R.G. Downey, M.R. Fellows, Fixed-parameter tractability and completeness II: On completeness for $W[1]$, Theoret. Comput. Sci. (1995).
[18] R.G. Downey, M.R. Fellows, Parameterized Complexity, Springer, 1999.
[19] E.A. Emerson, Temporal and modal logic, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B: Formal Models and Semantics, Elsevier, 1990.
[20] M. Frick, M. Grohe, The complexity of first-order and monadic second-order logic revisited, in: LICS '02, IEEE Computer Society, Washington, DC, 2002, pp. 215–224.
[21] M.R. Garey, D.S. Johnson, Computer and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, 1979.
[22] G. Gottlob, C. Koch, R. Pichler, Xpath processing in a nutshell, SIGMOD Rec. 32 (2) (2003).
[23] D. Janin, I. Walukiewicz, On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic, in: Proc. of CONCUR, 1996.
[24] D. Janssens, G. Rozenberg, Graph grammars with node-label controlled rewriting and embedding, in: Proc. of COMPUGRAPH, 1983.
[25] M.S. Lam, J. Whaley, V.B. Livshits, M.C. Martin, D. Avots, M. Carbin, C. Unkel, Context-sensitive program analysis as database queries, in: Proc. of PODS, 2005.
[26] T. Lengauer, E. Wanke, Efficient decision procedures for graph properties on context-free graph languages, J. ACM 40 (2) (1993).
[27] J.E. Mezei, J.B. Wright, Algebraic automata and context-free sets, in: Proc. of the Conference on the Algebraic Theory of Machines, Languages, and Semigroups, Information and Control 11 (1967).
[28] G. Nelson, D.C. Oppen, Simplification by cooperating decision procedures, ACM Trans. Program. Lang. Syst. 1 (2) (1979).
[29] Oracle BPEL Process Manager 2.0 Quick Start Tutorial, http://www.oracle.com/technology/products/ias/bpel/index.html.
[30] C.H. Papadimitriou, M. Yannakakis, On the complexity of database queries (extended abstract), in: PODS '97: Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ACM Press, New York, NY, 1997, pp. 12–19.
[31] T. Pavlidis, Linear and context-free graph grammars, J. ACM 19 (1) (1972).
[32] A. Schurr, Logic based programmed structure rewriting systems, Fund. Inform. 26 (3–4) (1996).
[33] M. Sipser, Introduction to the Theory of Computation, PWS Publishing Company, 1997.
[34] B. Trakhtenbrot, Impossibility of an algorithm for the decision problem in finite classes, Dokl. Akad. Nauk SSSR 70 (1950).